FreeLing User Manual

2.2

September 2010

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 tokenizing, sentence splitting, morphological analysis, NE detection and classification, recognition of dates/numbers/physical magnitudes/currency/ratios, PoS tagging, shallow parsing, dependency parsing, WN-based sense annotation, and coreference resolution. Future versions are expected to improve performance in existing functionalities, as well as incorporate new features, such as word sense disambiguation, document classification, etc.

FreeLing is designed to be used as an external library from any application requiring this kind of services. Nevertheless, a simple 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.

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 machine translation system to a syntactic-tree drawing interface) can access those data and process them as needed.

Nevertheless, FreeLing package provides a quite complete application program (`analyzer`) that enables an end user to obtain the analysis of a text. See chapter 5 for details.

This program offers a set of options that cover almost all 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 as described above.

1.3 Supported Languages

The distributed version supports (to different extents) English, Spanish, Catalan, Galician, Italian, Portuguese, Asturian and Welsch.

See the `Linguistic Data` section on FreeLing webpage to find out about the size of the morphological dictionaries and the source of their data.

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.

- The English sense dictionary is straightforwardly extracted from WN 1.6 and therefore is distributed under the terms of WN license, as is all knowledge extracted from WN contained in thos package.
- Spanish sense dictionary is a reduced subset extracted from EuroWordNet, This subset, included in this FreeLing package, is distributed under GPL.
- Catalan sense dictionary is extracted from CREL project, funded by the Catalan government, and is distributed under GPL.

  See http://wordnet.princeton.edu for details on WordNet,
  and http://www.illc.uva.nl/EuroWordNet for more information on EuroWordNet.

1.4 License

FreeLing code is licensed under GNU General Public License (GPL).
The linguistic data 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 was originally written by people in TALP Research Center at Universitat Politecnica de Catalunya (http://talp.upc.edu).

Spanish and Catalan linguistic data were originally developed by people in CLiC, Centre de Llenguatge i Computacio at Universitat de Barcelona (http://clic.ub.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://www.lsi.upc.edu/~nlp/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 (about 400Mb)
- Some external libraries are required to compile FreeLing:
  - `libpcre` (version 4.3 or higher)
    Perl C Regular Expressions. Included in most Linux distributions. Just make sure you have installed both development and runtime packages.
    Orientative package names (check the package manager in your system):
    * Ubuntu/Debian: `libpcre3-dev libpcre3`
    * OpenSuse/Fedora/Mandriva: `libpcre libpcre-devel`
    * Slackware: `pcre`
  - `libdb` (version 4.1.25 or higher)
    Berkeley DB. Included in all Linux distributions. You probably have it already installed. Just make sure you have installed both runtime and development packages, as well as C++ support:
    Orientative package names (check the package manager in your system):
    * Ubuntu/Debian: `libdb4.7 libdb4.7-dev libdb4.7++ libdb4.7++-dev`
    * OpenSuse/Fedora/Mandriva: `libdb-4.7 libdb-4.7-devel libdb_cxx-4.7 libdb_cxx-4.7-devel`
    * Slackware: `db4`
  - `libboost` (version 1.31 or higher)
    Boost library. Included in all Linux distributions. You probably **do not** have it installed. Make sure to installed both runtime and development packages:
    Orientative package names (check the package manager in your system):
    * Ubuntu/Debian: `libboost-dev libboost-filesystem libboost-program-options`
    * OpenSuse/Fedora/Mandriva: `libboost-devel libboost`
    * Slackware: `boost`

– `Omlet & Fries` (omlet v.1.0.1 or higher, fries v.1.2 or higher)
Machine Learning utility libraries, used by Named Entity Classifier. These libraries contain linguistic data structures, so they are required even if you do not plan to use the NEC ability of FreeLing. Available from `http://www.lsi.upc.edu/~nlp/omlet+fries`. Already included in FreeLing binary `.deb` 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` 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.

The provided packages will only work on debian-based distributions. They have been tested in Ubuntu (8.04 Hardy, 8.10 Intrepid, 9.04 Jaunty, 9.10 Karmic, 10.04 Lucid) and Debian (5.0.5 Lenny).

To install the package, just double-click on the package file should launch the debian installer, which will take care of the dependencies.

If that doesn't work, you can install it manually:

```
sudo apt-get install libdb4.6++ libpcre3 libboost-filesystem1.34.1
sudo dpkg -i FreeLing-2.2.deb
```

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

Libraries `libfries`, and `libomlet` are included in the FreeLing `.deb` package, so you do not need to install them.

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
   ```

   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**

   Many of the 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 a similar manager.

   If you have another distribution, use your package manager to locate and install the appropriate library packages (see section 2.1).

- *Install libpcre headers:*
  Installing the headers will pull in the binaries, if not already there:

  ```
  sudo apt-get install libpcre3-dev
  ```

- *Install libdb4 for C++:*
  Version doesn't need to be exactly 4.6. Any 4.x available in your distribution will do. Installing the headers will pull in the binaries:

  ```
  sudo apt-get install libdb4.6++-dev
  ```

- *Install libboost headers and binaries:*
  In Ubuntu Intrepid/Jaunty/Karmic, and Debian Lenny, installing this single package, will pull in all the required others:

  ```
  sudo apt-get install libboost-dev
  ```

  In Ubuntu Hardy packages are organized slightly different:

  ```
  sudo apt-get install libboost-filesystem-dev libboost-program-options-dev libboost-graph-de
  ```

  In Ubuntu Lucid, you need to install:

  ```
  sudo apt-get install  libboost-filesystem1.40-dev libboost-program-options1.40-dev
  ```

NOTE: Alternatively, you can install the libraries above from source. *Don't* do that unless you know what you're doing. Having more than one installation of libdb or libpcre may confuse the compiler and cause errors when compiling FreeLing, unless you master how to appropriately set paths for your compiler, linker, and loader.
However, if you want to do so, follow the installation instructions in the source packages for those libraries. All of them use the standard `./configure && make && make install` procedure. In the case of BerkeleyDB, you have to build C++ support using the `--enable-cxx` option with `./configure`.

3. Install FreeLing

- *Install libfries:*

  ```
  tar xzvf libfries-1.2.tar.gz
  ./configure
  make
  sudo make install
  ```

- *Install libomlet:*

  ```
  tar xzvf libomlet-1.0.1.tar.gz
  ./configure
  make
  sudo make install
  ```

  Omlet&Fries libraries are contained in the files `libomlet.so` and `libfries.so`, installed by default in `/usr/local/lib` (or `/usr/lib` if installed from binary package). If you are a developer, you need to know that `libfries.so` contains the classes storing linguistic information, which are described in section 4.1.1.

- *Install FreeLing itself:*

  ```
  tar xzvf FreeLing-2.2.tar.gz
  ./configure
  make
  sudo make install
  ```

Note that if you (or the library package) install some libraries or headers in non-standard directories (that is, other than `/usr/lib` or `/usr/local/lib` for libraries, or other than `/usr/include` or `/usr/local/include` for headers) you may need to use the `CPPFLAGS` or `LDFLAGS` variables to properly run the `./configure` script when compiling FreeLing.

For instance, if you installed BerkeleyDB from a `rpm` package, the `db_cxx.h` header file may be located at `/usr/include/db4` instead of the default `/usr/include`. So, if `./configure` complains about not finding the library header, you'll have to specify where to find it, with something like:
`./configure CPPFLAGS='-I/usr/include/db4'`

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

Sample program `analyze` is installed in `/usr/local/bin`. See sections 2.3 and 5 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 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 these two steps.

3. Checkout FreeLing sources

```
mkdir mysrc
cd mysrc
svn checkout http://devel.cpl.upc.edu/freeling/svn/latest/freeling
svn checkout http://devel.cpl.upc.edu/freeling/svn/latest/omlet
svn checkout http://devel.cpl.upc.edu/freeling/svn/latest/fries
```

4. Prepare local repositories for compilation

```
cd fries
aclocal; libtoolize; autoconf; automake -a
cd ../omlet
aclocal; libtoolize; autoconf; automake -a
cd ../freeling
aclocal; libtoolize; autoconf; automake -a
cd ..
```

5. Build and install FreeLing

```
cd fries
./configure && make
sudo make install

cd ../omlet
./configure && make
sudo make install

cd ../freeling
./configure && make
sudo make install
```

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

```
cd freeling
svn update
./configure && make
sudo make install
```

Obviously, you can also update `fries` or `omlet` versions in the same way.

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 which may 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 5.

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 can 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 (additional pcre/db libraries porting might be required also...).

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

See the `Installing` section and the Forum in FreeLing webpage for details.

Chapter 3

Analysis Modules

In this chapter, each of the modules in FreeLing are described.

A typical module receives a list of sentences, and enriches its words with new analysis.

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.

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

3.1 Tokenizer Module

The first module in the 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, receives the name of the file with tokenization rules
    tokenizer(const std::string &);

    /// tokenize string with default options
    std::list<word> tokenize(const std::string &);

    /// tokenize string with default options, accumulating byte-offset of words
    std::list<word> tokenize(const std::string &, unsigned long &);
};
```

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

3.1.1 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 Perl regexp.
E.g. `ALPHA [A-Za-z]`

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 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 substrings (up to 9). A token will be created for each substring from 1 to the specified value.
- The third field is the regexp to match against the input. line. Any Perl regexp convention may be used.

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.

3.2 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 sumbit further token lists to be added, or request the splitter to flush the buffer.

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
      /// wants to wait to see what is coming next.
      std::list<sentence> split(const std::list<word> &, bool);
};
```

3.2.1 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 `MaxLines` 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
MaxLines 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`), a sentence end is allowed to be introduced inside such a pair.

`MaxLines` states how many text lines are read 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". Obviously, this option is only effective if `AllowBetweenMarkers` is on.

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>
```

3.3 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 3.4 to 3.11).

At instantiation time, it receives a `maco_options` object, containing information about which submodules have to be created and which files have to 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 and enrich given sentences.
      void analyze(std::list<sentence> &);
};
```

The `maco_options` class has the following API:

```
class maco_options {

 public:
    /// Language analyzed
    std::string Lang;

    /// Submodules to activate
    bool AffixAnalysis,   MultiwordsDetection,
         NumbersDetection, PunctuationDetection,
         DatesDetection,   QuantitiesDetection,
         DictionarySearch, ProbabilityAssignment;
    /// kind of NER wanted (NER_BASIC, NER_BIO, NER_NONE)
    int NERecognition;

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

    /// Extra parameters for Number Detection module
    std::string Decimal, Thousand;
    /// Extra parameters for Probability Assignment module
    double ProbabilityThreshold;

    /// 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 they are defined above.
    void set_active_modules(bool,bool,bool,bool,bool,bool,bool,bool,int,bool);
    void set_data_files(const std::string &,const std::string &,
                        const std::string &,const std::string &,
                        const std::string &,const std::string &,
                        const std::string &,const std::string &);
    void set_nummerical_points(const std::string &,const std::string &);
    void set_threshold(double);
```

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.

3.4 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.

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.

For the reasons described so far, there is no options or 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 &);

      /// Detect number expressions in given sentence
      void annotate(sentence &);
};
```

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.

3.5 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 &);

      /// Detect punctuation in given sentence
      void annotate(sentence &);
};
```

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.

3.5.1 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 tag`.
E.g.

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

One special line may be included for undefined punctuation symbols (any word with no alphanumeric character is considered a punctuation symbol).

This special line has the format: `<Other> tag`. E.g.

```
<Other> Fz
```

3.6 Dates Detection Module

The dates detection module, as the number detection module in section 3.4, 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 &);

      /// Detect date/time expressions in given sentence
      void annotate(sentence &);
};
```

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.

3.7 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 Linguistc Data developer.

The API for this module is the following:

```
class dictionary {
   public:
      /// Constructor
      dictionary(const std::string &, const std::string &,
                 bool, const std::string &);

      /// Get analysis for a given form, and add them
```

```
      /// to given analysis list
      void search_form(const std::string &, std::list<analysis> &);

      /// Analyze words in given sentence
      void annotate(sentence &);
}
```

The parameters of the constructor are:

- The language of the processed text. This is required by the affixation submodule to properly handle graphical accents in latin languages.
- The dictionary file name. This may be a BerkeleyDB–indexed file (with extension `.db`) or either a plain text file (with extension `.src`). 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)

3.7.1 Form Dictionary File

The form dictionary is either a plain text file, or a BerkeleyDB–indexed file. The dictionary module relies on the extension to decide which format to expect (`.src` for plain text, `.db` for indexed files)

The plain text dictionary file (`.src`) format is described below. This file can be directly passed to the constructor of the dictionary search module.

Indexed Berkeley-DB files (`.db`) may be created with the `indexdict` program provided with FreeLing, which is called with the command:

```
indexdict indexed-dict-name.db  <source-dict.src
```

Where `source-dict.src` is a plain text dictionary, and `indexed-dict-name.db` is the resulting indexed file, which can be directly passed to the constructor of the dictionary search module.

See the (very simple) source code in `src/main/utilities/indexdict.cc` if you're interested on how it is indexed.

Format for `.src` dictionary files

Each line in the file must have the lemma–PoS list for one word. That is, each line has the format:
`form lemma1 PoS1 lemma2 PoS2 ....`
E.g.:

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

Whitespaces act as separators, so, make sure not to have extra whitespaces between fields or at the end of the line.

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+....`
For instance:

```
del de+el SPS+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 `SPS` that this entry may have in the dictionary, and *el* will get any tag starting with `DA`.

Note that a contraction 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 SPS+*
```

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

3.7.2 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 parole tag of found dictionary entry (e.g. cruz is NCFS). The condition is a perl RegExp
4. Parole 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)

E.g., prefix rules:

```
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 would receive the `AQ0CN0` tag and the affix (`anti`) plus the lemma as the lemma of the prefixed word. 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.

E.g., 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` 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
```

3.8 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 automat {
 public:
      /// Constructor
      automat();

      /// Detect patterns in given sentence
      void annotate(sentence &);
};

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

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

3.8.1 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 three fields: the multiword form, the multiword lemma, and the multiword PoS tag.

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

For instance:

```
a_buenas_horas a_buenas_horas RG
a_causa_de a_causa_de SPS00
<accidente>_de_trabajo accidente_de_trabajo $1:NC
```

The tag may be specified directly, or as a reference to the tag of some of the multiword components. In the previous example, the last 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".

3.9 Named Entity Recognition Module

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

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 captialization at sentence begginings.

The second module, named `bioner`, is based on machine learning algorithms in Omlet&Fries libraries, and has to be trained from a tagged corpus.

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

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

    /// ("annotate" is inherited from "automat")
    void annotate(sentence &);
};
```

The `bioner` module has a higher precision (over 90%), but is much slower, and adaptation to new languages requires a training corpus, and some feature engineering.

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

    /// Recognize NEs in given sentence
    void annotate ( sentence & );
};
```

3.9.1 Basic NER Options File (module `np`)

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 Espaa" 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 3.5) 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 *Pel* 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
pel
zapatero
china
</Names>
```

- Sections `<RE_NounAdj>` `<RE_Closed>` and `<RE_DateNumPunct>` allow to modify the default regular expressions for PAROLE 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 PAROLE 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>
```

3.9.2 *BIO* NER Options File (module `bioner`)

The machine-learning based NER module requieres a different configuration file. It consists of 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 NER. These features are processed by `libfries`.

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

- Section `<AdaBoostModel>` contains one line with the path to the model file learnt with AdaBoost. These models are learnt and used by `libomlet`.

  ```
  <AdaBoostModel>
  ner.abm
  </AdaBoostModel>
  ```

- 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>
  ```

- 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 `<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>
```

3.10 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 3.4). 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 expression in given sentence
      void annotate(sentence &);
};
```

3.10.1 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 3.8, `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 3.8).

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.

3.11 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 language code, the name of the file
      // containing probabilities, and a threshold.
      probabilities(const std::string &, const std::string &, double);

      /// Assign probabilities to all analysis of each word in sentence
      void annotate(sentence &);
      /// Assign probabilities to all analysis of given word
      void annotate_word(word &);
};
```

The constructor receives:

- The language code: It is used only to decide whether an EAGLES tagset is being used, and to appropriately shorten the tags if that's the case.
- The probabilities file name: The file that contains all needed 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.

3.11.1 Lexical Probabilities File

This file can be generated from a tagged corpus using the script `src/utilitities/TRAIN` provided in FreeLing package. See comments in the script file to find out which format the corpus is expected to have.

The probabilities file has six sections: `<UnknownTags>`, `<Theeta>`, `<Suffixes>`, `<SingleTagFreq>`, `<ClassTagFreq>`, `<FormTagFreq>`. Each section is closed by it corresponding tag `</UnknownTags>`, `</Theeta>`, `</Suffixes>`, `</SingleTagFreq>`, `</ClassTagFreq>`, `</FormTagFreq>`.

- 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 `<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 Parole label. The count is the number of occurrences in a training corpus.

  E.g. `NCMS000 33438`

3.12 Ortographic Correction Module

This module takes into account phonetic similarities and tries to provide analysis for words that would be otherwise treated as unknown words.

For instance, if a mispelled word such as *cavallo* is found, this module will find out that it sounds the same than a correct word in the dictionary (*caballo*), thus providing analysis for that otherwise unknown word.

This module is under development, and it is not operational yet.

3.13 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 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 dictionary file and a boolean.
      senses(const std::string &, bool);

      /// sense annotate selected analysis for each word in given sentences
      void analyze(std::list<sentence> &);
};
```

The constructor of this class receives:

- The name of the sense dictionary file. This file is a Berkely DB indexed file, containing sense codes for each lemma-PoS. Its contents can be build as described in section 3.20.1.
- A boolean stating whether the analysis with more than one sense must be duplicated.

  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, the `duplicate` boolean 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 the boolean 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
```

3.14 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 3.13 is a lighter and faster option.

The module is just a wrapper for UKB algorithm [AS09], which is integrated in FreeLing and distributed as-is under its original GPL license.

UKB algorithm 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 enriches each analysis of each word (for the selected PoS) with a ranked list of senses. The PageRank value is also provided as a result.

The API of the class is the following:

```
class disambiguator {
   public:
      /// Constructor. Receives a relation file for UKB, a sense dictionary,
      /// and two UKB parameters: epsilon and max iteration number.
      disambiguator(const std::string &, const std::string &, double, int);

      /// word sense disambiguation for each word in given sentences
      void analyze(std::list<sentence> &);
};
```

The constructor receives:

- The file name of the semantic relationship graph to load. This is a binary file containing pairs of related senses (WN synsets in this case). Relations are not labelled nor directed.

  This file is created from the plain files (extracted from UKB package) `data/common/wnet30-rels.txt` and `data/common/wnet30g-rels.txt` by FreeLing installation scripts. You can re-create it (or create a new one using a different relation set) with the following commands:

  ```
  compile_ukb -o $FLSHARE/common/wn30-ukb.bin $FLSHARE/common/wnet30-rels.txt
  compile_ukb -o $FLSHARE/common/xwn30-ukb.bin $FLSHARE/common/wnet30-rels.txt $FLSHARE/com
  ```

  (`$FLSHARE` refers to the `share/FreeLing` directory in your FreeLing installation, which defaults to `/usr/local/share/FreeLing` if you installed from source (`/usr/share/FreeLing` if you used a binary `.deb` package). You can change this environment variable to use different linguistic data files.

- The second parameter is a sense dictionary, with the possible senses for each word. Since this is the same information contained in the sense file for the Senses module described in section 3.13, it is generated by FreeLing installation scripts simply converting the default sense file (see sections 3.13 and 3.20.1) to the appropriate format.

You can re-create this file (or use any other sense dictionary with the right format) with the command (paths may differ):

```
convertdict <$FLSHARE/es/senses30.src >$FLSHARE/es/senses30.ukb
```

(`$FLSHARE` refers to the `share/FreeLing` directory in your FreeLing installation, which defaults to `/usr/local/share/FreeLing` if you installed from source, or `/usr/share/FreeLing` if you used a binary `.deb` package. Obviously, if you want to convert the file for a language different than Spanish, you have to use the right path).

- The last two 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.

3.15 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 &, const std::string &, bool, unsigned int);

       /// disambiguate given sentences
       void analyze(std::list<sentence> &);
};
```

The `hmm_tagger` constructor receives the following parameters:

- The language code: Used to determine if the language uses an EAGLES tagset, and to properly shorten the PoS tags in that case.
- 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/utilitities/TRAIN` provided in FreeLing package. See comments in the script file to find out which format the corpus is expected to have.
- 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 `relax_tagger` module can be tuned with hand written constraint, but is about 2 times slower than `hmm_tagger`.

```
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);

       /// disambiguate sentences
       void analyze(std::list<sentence> &);
};
```

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/utilitities/TRAIN` provided in FreeLing package. See comments in the script file to find out which format the corpus is expected to have.
- 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.

3.15.1 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/utilitities/TRAIN` provided in FreeLing package. See comments in the script file to find out which format the corpus is expected to have.

The file has seven sections: `<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 `<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".

3.15.2 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` 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*);
```

3.16 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/nec`. See the README there and the comments int the script for the details.

The API of the class is the following:

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

      /// Classify NEs in given sentence
      void analyze(std::list<sentence> &) const;
};
```

The constructor receives two parameters. The first is the tag that the NER module assigned to Named Entities, so the NEC can know which words to classify. The second parameter is the prefix of the configuration files for the model, as described below.

3.16.1 NEC Data File

The Named Entity Classification module requires three configuration files, with the same path and name, with suffixes `.rgf`, `.lex`, and `.abm`. Only the basename must be given as a parameter at instantiation time, file extensions are automatically added.

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

Both `.abm` and `.lex` files may be generated from an annotated corpus using the training programs in `libomlet` package.
(see `http://www.lsi.upc.edu/~ nlp/omlet+fries`)

The important file in the set is the `.rgf` file. This contains a definition of the context features that must be extracted for each named entity. The feature extraction language is that of [RCSY04] with some useful extensions.

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

3.17 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;
   /// Parse sentences in list
```

```
   void analyze(std::list<sentence> &);
};
```

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).

3.17.1 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(comi)`, `VMI*(comi)` 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...)

3.18 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 &);

   /// Enrich all sentences in given list with a depenceny tree.
   void analyze(std::list<sentence> &);
};
```

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 3.18.1.

3.18.1 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 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.

```
<SEMDB>
SenseFile ../senses16.db
WNFile    ../../common/wn16.db
</SEMDB>
```

The SenseFile must be a BerkeleyDB indexed file as described in the 6.5 section. The WNFile must be a BerkeleyDB indexed file, containing ontology information (hyperonymy, Top Ontology, WN semantic File, etc. The contents of this file are described in section 3.20.2.

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"
```

3.19 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:

3.19.1 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`, which use the libomlet package. (see `http://www.lsi.upc.edu/~nlp/omlet+fries`)

  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 ../senses16.db
  WNFile    ../../common/wn16.db
  </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.

3.20 Semantic Database Module

This module is not a component in the default analysis chain, but it can be used by the applications to enrich or post process the results of the analysis.

Moreover, this module is used by the other modules that need access to the semantic database: The sense annotator `senses`, the dependency parser `dep_txala`, and the coreference solver `coref`.

The API for this module is

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

    /// get list of words for a sense+pos
    std::list<std::string> get_sense_words(const std::string &, const std::string &);

    /// get list of senses for a lemma+pos
    std::list<std::string> get_word_senses(const std::string &, const std::string &);

    /// get sense info for a sensecode+pos
    sense_info get_sense_info(const std::string &, const std::string &);
};
```

3.20.1 Sense Dictionary File

The sense dictionary file is a Berkeley DB indexed file.

It can be created with the `indexdict` program provided with FreeLing, which is called with the command:

```
indexdict indexed-dict-name  <source-dict
```

See the (very simple) source code in `src/main/utilities/indexdict.cc` if you're interested on how it is indexed.

The source file (e.g. `senses16.src` provided with FreeLing) must contain the sense list of each lemma–PoS, one entry per line.

Each line has format: `type:lemma:PoS synset1 synset2 ....`
E.g.

```
W:cebolla:N 05760066 08734429 08734702
S:07389783:N chaval chico joven mozo muchacho
```

The *type* field may be either `W` (for Word) or `S` (for Sense), and indicates whether the rest of the line contains either a word and all its sense codes, or a sense code and all its synonym words.

For `W` entries, the sense code list is assumed to be ordered from most to least frequent sense for that lemma–PoS by the sense annotation module. This is used when value `msf` is selected for the `SenseAnnotation` option.

Type `S` entries are used by dependency parsing rules.

Sense codes can be anything (assuming your later processes know what to do with them). The provided files contain WordNet 1.6 synset codes.

3.20.2 WordNet file

The WordNet file is a Berkeley DB indexed file.

It can be created with the `indexdict` program provided with FreeLing, which is called with the command:

```
indexdict indexed-wn-name  <source-wn
```

See the (very simple) source code in `src/main/utilities/indexdict.cc` if you're interested on how it is indexed.

The source file (e.g. `wn16.src` provided with FreeLing) must contain at each line the information relative to a sense, with the following format:

```
synset:PoS  hypern:hypern:...:hypern  semfile  TopOnto:TopOnto:...:TopOnto
```

That is: the first field is the synset code plus its PoS, separated by a colon. The second field is a colon-separated list of its hypernym synsets. The third field is the WN semantic file the synset belongs to, and the last field is a colon-separated list of EuroWN TopOntology codes valid for the synset.

Note that the only 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.

Chapter 4

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.

4.1 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.

4.1.1 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.

4.1.2 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.
- `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).

4.2 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 recongnition system, or from a XML parser, or from any source suiting the application goals. Similarly, the obtained analysis, instead of being output, could be used in a translation system, or sent to a dialogue control module, etc.

```
int main() {
  string text;
  list<word> lw;
  list<sentence> ls;

  string path="/usr/local/share/FreeLing/es/";

  // create analyzers
  tokenizer tk(path+"tokenizer.dat");
  splitter sp(path+"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("es");
  // then, set required options on/off
  opt.QuantitiesDetection = false;  //deactivate ratio/currency/magnitudes detection
  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 = NER_BASIC;
  // alternatively, you can set active modules in a single call:
  //     opt.set_active_modules(true, true, true, true, true, false, true, true, NER_BASIC, false);

  // and provide files for morphological submodules. Note that it is not necessary
  // to set opt.QuantitiesFile, since Quantities module was deactivated.
  opt.LocutionsFile=path+"locucions.dat"; opt.AffixFile=path+"afixos.dat";
  opt.ProbabilityFile=path+"probabilitats.dat"; opt.DictionaryFile=path+"maco.db";
  opt.NPdataFile=path+"np.dat"; opt.PunctuationFile=path+"../common/punct.dat";
```

```
  // alternatively, you can set the files in a single call:
  //  opt.set_data_files(path+"locucions.dat", "", path+"afixos.dat",
  //                     path+"probabilitats.dat", path+"maco.db",
  //                     path+"np.dat", path+"../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("es", path+"tagger.dat", true, true);
  // create chunker
  chart_parser parser(path+"grammar-dep.dat");
  // create dependency parser
  dep_txala dep(path+"dep/dependences.dat", parser.get_start_symbol());

  // get plain text input lines while not EOF.
  while (getline(cin,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 and output morphosyntactic analysis and disambiguation
    morfo.analyze(ls);
    tagger.analyze(ls);

    // Do whatever our application does with the analyzed sentences
    ProcessResults(ls);

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

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

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

  // Process last sentence(s)
  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 s;
  word::const_iterator a;   //iterator over all analysis of a word
  sentence::const_iterator w;

  // for each sentence in list
  for (s=ls.begin(); s!=ls.end(); s++) {

    // print sentence XML tag
    cout<<"<SENT>"<<endl;
```

```
    // for each word in sentence
    for (w=s->begin(); w!=s->end(); w++) {

      // print word form, with PoS and lemma chosen by the tagger
      cout<<"  <WORD form=\""<<w->get_form();
      cout<<"\" lemma=\""<<w->get_lemma();
      cout<<"\" pos=\""<<w->get_parole();
      cout<<"\">"<<endl;

      // for each possible analysis in word, output lemma, parole and probability
      for (a=w->analysis_begin(); a!=w->analysis_end(); ++a) {

        // print analysis info
        cout<<"    <ANALYSIS lemma=\""<<a->get_lemma();
        cout<<"\" pos=\""<<a->get_parole();
        cout<<"\" prob=\""<<a->get_prob();
        cout<<"\"/>"<<endl;
      }

      // close word XML tag after list of analysis
      cout<<"</WORD>"<<endl;
    }

    // close sentence XML tag
    cout<<"</SENT>"<<endl;
  }
}
```

The above sample program may be found in `/src/main/simple_examples/sample.cc` in FreeLing tarball.

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 -lmorfo -ldb_cxx -lpcre -lomlet -fries -lboost_filesystem
```

Check the README file in the directory to learn more about compiling and using the sample programs.

Option `-lmorfo` links with libmorfo library, which is the final result of the FreeLing compilation process. The oher options refer to other libraries required by FreeLing.

You may have 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.

Chapter 5

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 4.2

FreeLing also provides a couple of programs `analyzer_server` and `analyzer_client` that perform the same task, but the server remains loaded after analyzing each client's request, thus reducing the starting-up overhead if many small files have to be processed. Server-client pairs communicate via sockets.

Both `analyzer` and `analyzer_server` are usually called with an option `-f config-file` (if ommitted, they 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`).

The `analyze` script described below handles all these default paths and variables and makes everything easier if you want to use the defaults

5.1 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 decide whether you want the client-server or the straight version.

The sample main program is called with the command:

```
analyze [port-number] [-f config-file] [options]
```

If `port-number` is ommited, the analyzer is started in straight mode. If a port number is provided, a server will be launched to accept socket connections on that port.

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` 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 5.3.1 for details.

5.1.1 Straight mode

When `port-number` is ommited, a stand-alone analyzer will be launched. It 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.

5.1.2 Client/server mode

If `port-number` is specified, a server is initiated and starts listening for incoming requests.
E.g.:

```
analyze 50005 -f en.cfg  &
```

Then, clients can request analysis to the server, with:

```
analyzer_client localhost 50005  <myinput  >myoutput
```

or, from a remote machine:

```
analyzer_client xx.xx.xx.xx 50005  <myinput  >myoutput
```

(where `xx.xx.xx.xx` should be the IP address or hostname for the machine where the server was started)

Any client trying to connect to the server while it is attending another request, will be blocked and attended when the current request is finished.

5.2 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 5.3.2. 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 l PP3CSD00 1
```

```
gustan gustar VMIP3P0 1
los el DA0MP0 0.975719 lo NCMP000 0.00019425 l 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 l PP3CSD00
gustan gustar VMIP3P0
los el DA0MP0
gatos gato NCMP000
. . Fp
```

We can also ask for the synsets 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 l 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:

```
analyzer -f myconfig.cfg --inpf morfo --outf tagged <mytext.mrf >mytext.tag
```

See options InputFormat and OutputFormat in section 5.3.1 for details on which are valid input and output formats.

5.3 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 5.3.1, both in their command-line and configuration file notations. Configuration file follows the usual linux standards, a sample file may be seen in section 5.3.2.

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.

5.3.1 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`	`N/A`

Prints to stdout a help screen with valid options and exits.

- **Configuration file**

Command line	Configuration file
`-f <filename>`	`N/A`

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

- **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:

Module	Mask
Splitter	0x00000001
Tokenizer	0x00000002
Morphological analyzer	0x00000004
Options management	0x00000008
Number detection	0x00000010
Date identification	0x00000020
Punctuation detection	0x00000040
Dictionary search	0x00000080
Suffixation rules	0x00000100
Multiword detection	0x00000200
Named entity detection	0x00000400
Probability assignment	0x00000800
Quantities detection	0x00001000
Named entity classification	0x00002000
Automata (abstract)	0x00004000
PoS Tagger (abstract)	0x00008000
HMM tagger	0x00010000
Relaxation labelling	0x00020000
RL tagger	0x00040000
RL tagger constr. grammar	0x00080000
Sense annotation	0x00100000
Chart parser	0x00200000
Parser grammar	0x00400000
Dependency parser	0x00800000
Correference resolution	0x01000000
Utilities	0x02000000

- **Language of input text**

Command line	Configuration file
`--lang <language>`	`Lang=<language>`

Language of input text (es: Spanish, ca: Catalan, en: English, cy: Welsh, it: Italian, gl: Galician, pt: Portuguese).

Other languages may be added to the library. See chapter 6 for details.

- **Splitter Buffer Flushing**

Command line	Configuration file
`--flush`, `--noflush`	`AlwaysFlush=(yes|y|on|no|n|off)`

When inactive (most usual choice) sentence splitter buffers lines until a sentence marker is found. Then, it outputs a complete sentence. When active, the splitter never buffers any token, and considers each newline as 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, 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`$_N$
  - parsed: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged, optionally sense–annotated, and parsed text.
  - dep: tokenized, sentence-splitted, morphologically analyzed, PoS-tagged, optionally sense–annotated, and dependency-parsed text.

- **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/utilitities/TRAIN` provided in FreeLing package. See comments in the script file for details about how to use it.

- **Use UTF8 encoding**

Command line	Configuration file
`--utf`	`UseUTF=(yes|y|on|no|n|off)`

FreeLing internally uses `latin1` ISO encoding. When this option is active, the input is assumed to be in `utf8`, and converted to `latin1` before processing. The results are converted back to `utf8` before being output.

Note that this will *not* work with `utf8` encodings that cannot be converted to `latin1`.

When using the server version (see chapter 5), you can choose whether it will be the server or the client who will perform the conversion.

Specifying `--utf` at the server command line means that the server expects `utf8` input, so the client must send so (thus, no conversion must be carried out by the client).

Specifying `--utf` at the client command line means that the client will convert `utf8` input to `latin1` before sending the request to the server. So, the server has to be expecting `latin1` input (i.e. no conversion in the server side).

- **Tokenizer File**

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

File of tokenization rules. See section 3.1 for details.

- **Splitter File**

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

File of splitter options rules. See section 3.2 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 3.7.2 for details.

- **Multiword Detection**

Command line	Configuration file
`--loc, --noloc`	`MultiwordsDetection=(yes|y|on|no|n|off)`

Whether to perform multiword detection. Multiwords may be detected if a multiword file is provided. Multiword File option, below).

- **Multiword File**

Command line	Configuration file
`-L <filename>, --floc <filename>`	`LocutionsFile=<filename>`

Multiword definition file. See section 3.8 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
`-M <filename>`, `--fpunct <filename>`	`PunctuationFile=<filename>`

Punctuation symbols file. See section 3.5 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 3.10 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. Must be a Berkeley DB indexed file. See section 3.7 and chapter 6 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 3.11 for details.

- **Unknown Words Probability Threshold.**

Command line	Configuration file
`--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 Recognizer File**

Command line	Configuration file
`--ner [bio|basic|none]`, `--fnp <filename>`	`NPDataFile=<filename>`

Configuration data file for active NE recognizer (either "bio" or "basic"). See section 3.9 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 Prefix**

Command line	Configuration file
`--fnec <filename>`	`NECFilePrefix=<filename>`

Prefix to find files for Named Entity Classifier configuration.

The searched files will be the given prefix with the following extensions:

  - `.rfg`: Feature extractor rule file.
  - `.lex`: Feature dictionary.
  - `.abm`: AdaBoost model for NEC.

See section 3.16 for details.

- **Sense Annotation**

Command line	Configuration file
`--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 contents of the "Sense Dictionary File" described below.

- **Sense Dictionary File**

Command line	Configuration file
`--fsense <filename>`	`SenseFile=<filename>`

Word sense data file. It is a Berkeley DB indexed file. See section 3.20.1 for details.

- **Duplicate Analysis for each Sense**

Command line	Configuration file
`--dup`, `--nodup`	`DuplicateAnalysis=(yes|y|on|no|n|off)`

When this option is set, the senses annotator will duplicate the analysis once for each of its possible senses. See section 3.13 for details.

This may be useful if one wants to perform WSD, or to use the *sense* field in the `analysis` in the constraint grammar (see section 3.15.2).

- **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 3.15.1 for details.

- **Relaxation labelling tagger constraints file**

Command line	Configuration file
`-R <filename>`	`TaggerRelaxFile=<filename>`

File containing the constraints to apply to solve the PoS tagging. See section 3.15.2 for details.

- **Relaxation labelling tagger iteration limit**

Command line	Configuration file
`--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
`--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 after tagging**

Command line	Configuration file
`--retk`, `--noretk`	`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 3.7.2 and 3.15 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 3.15 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 3.17.1 for details.

- **Dependency Parser Rule File**

Command line	Configuration file
`-J <filename>`, `--dep <filename>`	`DepRulesFile==<filename>`

Rules to be used to perform dependency analysis. See section 3.18.1 for details.

5.3.2 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.

```
# ---- sample configuration file for Spanish analyzer

#### General options
Lang=es

#### Trace options. Only effective if we have compiled with -DVERBOSE
TraceLevel=0
TraceModule=0x0000

## Options to control the applied modules. The input may be partially
## processed, or not a full analysis may me wanted. The specific
## formats are a choice of the main program using the library, as well
## as the responsability of calling only the required modules.
InputFormat=plain
OutputFormat=morfo

# 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
DecimalPoint=","
ThousandPoint="."
LocutionsFile=$FREELINGSHARE/es/locucions.dat
QuantitiesFile=$FREELINGSHARE/es/quantities.dat
AffixFile=$FREELINGSHARE/es/afixos.dat
ProbabilityFile=$FREELINGSHARE/es/probabilitats.dat
DictionaryFile=$FREELINGSHARE/es/maco.db
PunctuationFile=$FREELINGSHARE/common/punct.dat
```

```
ProbabilityThreshold=0.001
#NER options
NERecognition=basic
NPDataFile=$FREELINGSHARE/es/np.dat
## --- comment lines above and uncomment those below, if you want
## --- a better NE recognizer (higer accuracy, lower speed)
#NERecognition=bio
#NPDataFile=$FREELINGSHARE/es/ner/ner.dat

## NEC options
NEClassification=no
NECFilePrefix=$FREELINGSHARE/es/nec/nec

## Sense annotation options (none,all,mfs)
SenseAnnotation=none
SenseFile=$FREELINGSHARE/es/senses16.db
DuplicateAnalysis=false
UKBRelations=$FREELINGSHARE/common/wn16-ukb.bin
UKBDictionary=$FREELINGSHARE/es/senses16.ukb
UKBEpsilon=0.03
UKBMaxIter=10

#### Tagger options
Tagger=hmm
#Tagger=relax
TaggerHMMFile=$FREELINGSHARE/es/tagger.dat
TaggerRelaxFile=$FREELINGSHARE/es/constr_gram.dat
TaggerRelaxMaxIter=500
TaggerRelaxScaleFactor=670.0
TaggerRelaxEpsilon=0.001
TaggerRetokenize=yes
TaggerForceSelect=retok

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

#### Dependence Parser options
DepParser=txala
DepTxalaFile=$FREELINGSHARE/es/dep/dependences.dat
DepMaltFile=$FREELINGSHARE/es/malt/malt.dat

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

Chapter 6

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 5.3.2).

6.1 Tokenizer

The first module in the processing chain is the tokenizer. As described in section 5.3.1, 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.

6.2 Morphological analyzer

The morphological analyzer module consists of several sub-modules that may require language customization. See sections 3.3 to 3.11 for details on data file formats for each module:

6.2.1 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.

6.2.2 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`.

6.2.3 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`.

6.2.4 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.

6.2.5 Dictionary search

The lexical forms for each language are sought in a Berkeley 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.

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

6.2.6 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 3.7.2 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.

6.2.7 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 creted using a tagged corpus and the script provided in `src/utilities/TRAIN`.

See section 3.11 for format details.

6.3 HMM PoS Tagger

The HMM PoS tagger only requires an appropriate HMM parameters file, given by the TaggerHMMFile option. See section 3.15.1 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` provided in FreeLing package.

6.4 Relaxation Labelling PoS Tagger

The Relaxation Labelling PoS tagger only requires an appropriate pseudo- constraint grammar file, given by the RelaxTaggerFile option. See section 3.15.2 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` provided in FreeLing 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`

6.5 Sense annotation

The sense annotation module uses a BerkeleyDB indexed file. This file may also be used by the dependency labeling module (see section 3.18.1).

It may be created for a new language (or or form a new knowledge source) with the program `src/utilities/indexdict` provided with FreeLing.

The source file must have the sense list for one lemma–PoS at each line.

Each line has format: `W:lemma:PoS synset1 synset2 ....`

E.g.

```
W:cebolla:N 05760066 08734429 08734702
```

The first sense code in the list is assumed to be the most frequent sense for that lemma–PoS by the sense annotation module. This only takes effect when value `msf` is selected for the `SenseAnnotation` option.

The file may also contain the same information indexed by synset (that is, the list of synonyms for a given synset). This is needed if you are using the `synon` function in your dependency rules (see section 3.18.1), or if you want to access that information for your own purposes. The lines with this information have the format:

Each line has format: `S:synset:PoS lemma1 lemma2 ....` E.g.
`S:07389783:N chaval chico joven mozo muchacho nio`

To create a sense file for a new language, just list the sense codes for each lemma-PoS combination in a text file (e.g. `sensefile.txt`), with lines in the format described above, and then issue:
`indexdict sense.db < sensefile.txt`

This will produce an indexed file `sense.db` which is to be given to the analyzer via the `SenseFile` option in configuration file, or via the `--fsense` option at command line. It can also be referred to in the entry `WNFile` of the `<SEMDB>` section of a file of dependency labeling rules (section 3.18.1).

6.6 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 3.17.1 for format details.

6.7 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 3.18.1 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.

[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.

[RCSY04] Dan Roth, Chad Cumby, Mark Sammons, and Wen-Tau Yih. A relational feature extraction language (fex). Technical report, University of Illinois at Urbana Champaign, `http://l2r.cs.uiuc.edu/~cogcomp/software.php`, 2004.

[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.