FreeLing User Manual

2.0

November 2007

Contents

Chapter 1

FreeLing: Natural language analysis libraries

The FreeLing package consists of a library providing language analysis services (such as morfological analysis, date recognition, PoS tagging, etc.)

The current version (2.0) of the package provides tokenizing, sentence splitting, morphological analysis, NE detection and classification, recognition of dates/numbers/physical magnitudes/currency/ratios, PoS tagging, shallow parsing, dependency parsing, and WN-based sense annotation. Future versions are expected to improve performance in existing functionalities, as well as incorporate new features, such as word sense disambiguation, document classification, anaphora resolution, 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 Supported Languages

The distributed version includes morphological dictionaries for covered languages (English, Spanish, Catalan, Galician, and Italian):

- The Spanish dictionary was obtained from the Spanish Resource Grammar project developed at the Universitat Pompeu Fabra, and contains over 550,000 forms corresponding to more than 76.000 lemma-PoS combinations. These data are distributed under their original Lesser General Public License For Linguistic Resources (LGPL-LR) license. See THANKS and COPYING files for further information.

- The Catalan dictionary is hand build and contains near 67,000 forms corresponding to more than 7,400 different combinations lemma-PoS.

- The Galician dictionary was obtained from OpenTrad project (a nice open source Machine Translation project at `www.opentrad.org`), and contains over 90,000 forms corresponding to near 7,500 lemma-PoS combinations. These data are distributed under their original Creative Commons license, see THANKS and COPYING files for further information.

- The English dictionary was automatically extracted from WSJ, with accurate manual post-edition and completion. It contains over 65,000 forms corresponding to some 40,000 different combinations lemma-PoS.

- The Italian dictionary is extracted from Morph-it! lexicon developed the University of Bologna, and contains over 360,000 forms corresponding to more than 40,000 lemma-PoS

combinations. These data are distributed under their original Creative Commons license, see THANKS and COPYING files for further information.

Smaller dictionaries (Catalan and Galician) are expected to cover over 80% of open-category tokens in a text. Larger dictionaries are expected to cover between 90-95% of open-category tokens in a text. For words not found in the dictionary, all open categories are assumed, with a probability distribution based on word suffixes, which includes the right tag for 99% of the words, and allow the tagger to make the most suitable choice based on tag sequence probability.

This version also includes WordNet-based sense dictionaries for 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. You'll find a copy in the LICENSES/WN.license file in the distribution tarball.
- Catalan and Spanish sense dictionaries are extracted from EuroWordNet, and the reduced subsets included in this FreeLing package are distibuted under Gnu GPL, as the rest of the code and data in this package. Find a copy of the license in the LICENSES/GPL.license file.

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

1.2 Contributions

Many people contributed, directly or indirectly, to enlarge and enhance this software. See the THANKS file in the distribution package or the THANKS section in FreeLing webpage to learn more.

1.3 Requirements

To install FreeLing you'll need:

- A typical Linux box with usual development tools:
  - bash
  - make
  - C++ compiler with basic STL support (e.g. g++ version 3.x)
- Enough hard disk space (about 50Mb)
- Some external libraries are required to compile FreeLing:

  pcre (version 4.3 or higher) Perl C Regular Expressions. Included in most usual Linux distributions. Just make sure you have it installed.
  Also available from `http://www.pcre.org`

  db (version 4.1.25 or higher) Berkeley DB. Included in all usual Linux distributions. You probably have it already installed. Make sure of it, and that C++ support is also installed (may come in a separate package).
  Also available from `http://www.sleepycat.com`. Do not install it twice unless you know what you are doing.

  libcfg+ (version 0.6.1 or higher) Configuration file and command-line options management. May not be in your linux distribution.
  Available from `http://www.platon.sk/projects/libcfg+/`, follow installation instructions provided in the libcfg+ package.

Omlet & Fries (omlet v.0.96 or higher, fries v.0.93 or higher) Machine Learning utility libraries, used by Named Entity Classifier. Installation scrips are not very clever, so these libraries 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`

Note that you'll need both the binary libraries and their source headers (in some distributions the headers come in a separate package tagged `-devel` or `-dev`, e.g. the libpcre library may be distributed in two packages: the first, say `libpcre-4.3.rpm`, contains the binary libraries, and the second, say `libpcre-devel-4.3.rpm`, provides the source headers)

Note also that if you (or the library package) install those 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 `./configure` script.

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:

```
./configure CPPFLAGS='-I/usr/include/db4'
```

The BerkeleyDB package is probably installed in your system, but you may need to install C++ support, which (depending on your distribution) may be found in a separate package (such as db4-cxx.rpm, db4-cxx-devel.rpm, or the like).

See next section and INSTALL file for further details.

1.4 Installation

Installation follows standard GNU autoconfigure installation procedures. See the file INSTALL for further details.

More detailed installation instructions and tricks can be found in FreeLing web page and discussion forums.

The installation consists of a few basic steps:

- Decompress the FreeLing-X.X.tgz package in a temporary subdirectory.
- Issue the commands:

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

The last command may be issued as root.

You may control the installation defaults providing appropriate parametres to the `./configure` script. The command:

```
./configure --help
```

will provide help about installation options (e.g. non-default installation directory, non standard locations for required libraries, etc.)

The INSTALL file provides more information on standard installation procedures.

1.5 Executing

FreeLing is a library, which means that it is a tool to develop new programs which may require linguistic analysis services.

Nevertheless, a simple main program is included in the package for those who just want a text analyzer. This small program may easily be adapted to fit your needs (e.g. customized input/output formats).

Next chapter describes usage of this sample main program.

1.6 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/cfg+ 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/).

Chapter 2

Using the sample main program to process corpora

The simplest way to use the FreeLing libraries is via the provided sample main program, which allows the user to process an input text to obtain several linguistic processings.

The sample main program is called with the command:

```
analyzer [-f <config-file>] [options]
```

If `<config-file>` is not specified, a file named `analyzer.cfg` is searched in the current working directory. Extra options may be specified in the command line to override any config-file contents.

The FreeLing package includes default configuration files for Spanish, Catalan and English. They may be found at `share/FreeLing/config` directory under the FreeLing installation directory (by default, `/usr/local`).

The `analyzer` program reads from standard input and prints results to standard output, with a plain formats. You may adapt the print formats of this program to suit your needs.

2.1 Usage example

Assuming we have the folowing input file `mytext.txt`:

```
El gato come pescado.  Pero a Don
Jaime no le gustan los gatos.
```

we could issue the command:

```
analyzer -f myconfig.cfg <mytext.txt >mytext.mrf
```

Let's assume that `myconfig.cfg` is the file presented in section 2.2.2. 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:

```
analyzer -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:

```
analyzer -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 2.2.1 for details on which are valid input and output formats.

2.2 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 2.2.1, both in their command-line and configuration file notations. Configuration file follows the usual linux standards, a sample file may be seen in section 2.2.2.

2.2.1 Valid options

- **Help**

Command line	Configuration file
`-h, --help`	`N/A`

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

- **Trace Level**

Command line	Configuration file
`-l <int>, --tlevel <int>`	`TraceLevel=<int>`

Set the trace level (0:no trace - 3:maximum trace), for debugging purposes. Only valid if program was compiled with `-DVERBOSE` flag.

- **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
Utilities	0x01000000

- **Configuration file**

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

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

- **Language of input text**

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

Language of input text (es: Spanish, ca: Catalan, en: English). Other languages may be added to the library. See chapter 4 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, parsed, dep).

  - 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 lemma1 tag1 prob1 lemma2 tag2 prob2 ...`
  - 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 lemma1 tag1 prob1 sense11:...:sense1N lemma2 tag2 prob2 sense21:...:sense2N ...`

- **Output Format**

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

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

  - 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 lemma1 tag1 prob1 lemma2 tag2 prob2 ...` or the format `word lemma1 tag1 prob1 sense11:...:sense1N lemma2 tag2 prob2 sense21:...:sense2N ...` if sense tagging has been activated.
  - 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 the format `word lemma tag prob sense1:...:senseN` if sense tagging has been activated.
  - 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.

- **Tokenizer File**

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

File of tokenization rules. See section 2.3 for details.

- **Splitter File**

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

File of splitter options rules. See section 2.4 for details.

- **Suffix Analysis**

Command line	Configuration file
`--sufx, --nosufx`	`SuffixAnalysis=(yes|y|on|no|n|off)`

Whether to perform suffix analysis on unknown words. Suffix analysis applies known suffixation rules to the word to check whether it is a derived form of a known word (see option Suffix Rules File, below).

- **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, (see Multiword File option, below).

- **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.

- **Punctuation Detection**

Command line	Configuration file
`--punt, --nopunt`	`PunctuationDetection=(yes|y|on|no|n|off)`

Whether to assign PoS tag to punctuation signs

- **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.

- **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 SuffixAnalysis option.

- **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.

- **Decimal Point**

Command line	Configuration file
`--dec <string>`	`DecimalPoint=<string>`

Specify decimal point character (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 instance, in English is a comma, but in Spanish is a dot).

- **Multiword File**

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

Multiword definition file. See section 2.5 for details.

- **Quantity Recognition File**

Command line	Configuration file
`-Q <filename>, --fqty <filename>`	`QuantitiesFile=<filename>`

Quantitiy recognition configuration file. See section 2.6 for details.

- **Suffixation Rules File**

Command line	Configuration file
`-S <filename>, --fsuf <filename>`	`SuffixFile=<filename>`

Suffix rules file. See section 2.7 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.

- **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 2.8 for details.

- **Dictionary File**

Command line	Configuration file
`-D <filename>, --fdict <filename>`	`DictionaryFile=<filename>`

Dictionary database. Must be a Berkeley DB indexed file. See section 2.9 and chapter 4 for details.

- **Named Entity Recognition**

Command line	Configuration file
`--ner, --noner`	`NERecognition=(yes|y|on|no|n|off)`

Whether to perform NE recognition. Deactivating this feature will affect the behaviour of the NE Classification module.

- **Named Entity Recognizer File**

Command line	Configuration file
`-N <filename>, --fnp <filename>`	`NPDataFile=<filename>`

Configuration data file for simple heuristic Proper Noun recognizer. See section 2.10 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 2.11 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.

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 WN1.6 synsets.

- **Sense Dictionary File**

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

Word sense data file. It is a Berkeley DB indexed file. See section 2.12 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. For instance, analyzing the sentence *el gato come pescado* with `--senses all` and `--nodup` options, would enrich each analysis of each word with *a list* of all possible senses for that lemma and part–of–speech.

Form	Lemma	Tag	Prob	Senses
el	el	DA0MS0	1.0	-
gato	gato	NCMS000	1.0	01630731:07221232:01631653
come	comer	VMIP3S0	0.75	00794578:00793267
	comer	VMM02S0	0.25	00794578:00793267
pescado	pescado	NCMS000	0.84	05810856:02006311
	pescar	VMP00SM	0.16	00491793:00775186

Alternatively, if we use option `--dup`, each analysis is duplicated as many times as possible senses, so that each analysis has only one sense:

Form	Lemma	Tag	Prob	Senses
el	el	DA0MS0	1.0	-
gato	gato	NCMS000	0.33	01630731
	gato	NCMS000	0.33	07221232
	gato	NCMS000	0.33	01631653
come	comer	VMIP3S0	0.375	00794578
	comer	VMIP3S0	0.375	00793267
	comer	VMM02S0	0.125	00794578
	comer	VMM02S0	0.125	00793267
pescado	pescado	NCMS000	0.42	05810856
	pescado	NCMS000	0.42	02006311
	pescar	VMP00SM	0.08	00491793
	pescar	VMP00SM	0.08	00775186

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 2.15).

- **Punctuation Detection File**

Command line	Configuration file
`-M <filename>, --fpunct <filename>`	`PunctuationFile=<filename>`

Punctuation symbols file. See section 2.13 for details.

- **Tagger algorithm**

Command line	Configuration file
`-T <string>, --tag <string>`	`Tagger=<string>`

Algorithm to use for PoS tagging

  - hmm: Hidden Markov Model tagger, based on [Bra00].
  - relax: Relaxation Labelling tagger, based on [Pad98].

- **HMM Tagger configuration File**

Command line	Configuration file
`-H <filename>, --hmm <filename>`	`TaggerHMMFile=<filename>`

Parameters file for HMM tagger. See section 2.14 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.

- **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 2.15 for details.

- **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 suffix analysis, see section 2.7 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)

- **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 2.16 for details.

- **Dependency Parser Rule File**

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

Heuristic rules used to perform dependency analysis. See section 2.17 for details.

2.2.2 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). You also can use those files as a starting point to customize one config file to suit your needs.

```
#### default configuration file for spanish analyzer

####---------------- Trace options.  Only effective if we have compiled with -DVERBOSE.
####---------------- For development purposes only.
TraceLevel=0
TraceModule=0x0000
####---------------- General options
Lang=es
# Input/output formats.  (plain, token, splitted, morfo, tagged, sense, parsed, dep)
InputFormat=plain
OutputFormat=morfo
# consider each newline as a sentence end
AlwaysFlush=no
####---------------- Tokenizer options
TokenizerFile="/usr/local/share/FreeLing/es/tokenizer.dat"
####---------------- Splitter options
SplitterFile="/usr/local/share/FreeLing/es/splitter.dat" ####---------------- Morfo options
SuffixAnalysis=yes
MultiwordsDetection=yes
NumbersDetection=yes
PunctuationDetection=yes
DatesDetection=yes
QuantitiesDetection=yes
DictionarySearch=yes
ProbabilityAssignment=yes
NERecognition=yes
DecimalPoint=","
ThousandPoint="."
LocutionsFile=/usr/local/share/FreeLing/es/locucions.dat
QuantitiesFile=/usr/local/share/FreeLing/es/quantities.dat
SuffixFile=/usr/local/share/FreeLing/es/sufixos.dat
ProbabilityFile=/usr/local/share/FreeLing/es/probabilitats.dat
DictionaryFile=/usr/local/share/FreeLing/es/maco.db
NPDataFile=/usr/local/share/FreeLing/es/np.dat
PunctuationFile=/usr/local/share/FreeLing/common/punct.dat
ProbabilityThreshold=0.001
TitleLength=0
####----------------Tagger options
Tagger=hmm
TaggerHMMFile=/usr/local/share/FreeLing/es/tagger.dat
TaggerRelaxFile=/usr/local/share/FreeLing/es/constr_gram.dat
TaggerRelaxMaxIter=500
TaggerRelaxScaleFactor=670.0
TaggerRelaxEpsilon=0.001
####---------------- NEC options
NECClassification=no
NECFilePrefix=/usr/local/share/FreeLing/es/nec
####---------------- Sense annotation options
SenseAnnotation=none
SenseFile=/usr/local/share/FreeLing/es/senses16.db
WNFile=/usr/local/share/FreeLing/common/wn16.db
DuplicateAnalysis=no
####---------------- Parser options
GrammarFile=/usr/local/share/FreeLing/es/grammar-dep.dat
####---------------- Dependency Parser options
HeuristicsFile=/usr/local/share/FreeLing/es/dependences.dat
```

2.3 Tokenizer rules file

The 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 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).
- 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.

2.4 Splitter options file

The 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, 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, 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 infinite loops in case the markers are not properly closed in the text). A value of zero means "Never split, I'll risk to infinite loops". 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 endig or not. For instance, in the following example, "?" is an unabiguous 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>
```

2.5 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".

2.6 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'.

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 2.5, `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 2.5).

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.

2.7 Suffixation rules file

One rule per line, each rule has eight fields:

1. Suffix to erase form word form (e.g: crucecita - cecita = cru)
2. Suffix (* 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. Use original form as lemma instead of the lemma in dictionary entry
8. Consider the suffix always, not only for unknown words.
9. Retokenization info, explained below.. (or "-" if the suffix doesn't cause retokenization).

E.g.

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

The first line (`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 (`mela`) 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
```

2.8 Lexical Probabilities file

Define lexical probabilities for each tag of each word.

This file can be generated from a tagged corpus using the script `src/utilitities/make-probs-file.perl` provided in FreeLing package. See comments in the script file to find out in which format the file must be set.

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`

2.9 Word form dictionary file

Berkeley DB indexed file.

It may be created with the `src/utilities/indexdict` program provided with FreeLing. The source file must have the lemma–PoS list for a word form at each line.

Each line has format: `form lemma1 PoS1 lemma2 PoS2 ....` E.g.
`casa casa NCFS000 casar VMIP3S0 casar VMM02S0`

Lines corresponding to word 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`.

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.

Note that a contraction cannot be splitted in two different ways, so only a combination of lemmas and a combination of tags may appear in the dictionary.

2.10 Named entity recognition data file

This file controls the behaviour of the simple NE recognizer. It 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 2.13) 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 lemmas that are no considered to be 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.*

```
<Ignore>
i
english
dutch
spanish
</Ignore>
```

- Sections `<RE_NounAdj>` `<RE_Closed>` and `<RE_DateNumPunct>` allow to modify the default regular expressions for PAROLE Part-of-Speech tags. For instance, if Penn-Treebank-like tags are used for English, 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 INTEREST OF MARKET ANALISTS` 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).

2.11 Named entity classification data files

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 configuration option, suffixes 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 hacker..

Both `.abm` and `.lex` files may be generated from an annotated corpus using the training programs in the Omlet package, a great machine-learning library, available at `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.

2.12 Sense dictionary file

Berkeley DB indexed file.

It may be created with the `src/utilities/indexdict` program provided with FreeLing. The source file must have the sense list for one lemma–PoS per line.

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

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

Currently, only the PoS tag selected by the tagger is annotated, though the library is designed to support sense annotation for all possible tags of each word.

2.13 Punctuation symbol file

One punctuation symbol per line.
Each line has format: `punctuation-symbol tag`. E.g.

```
!  Fat
, Fc
:  Fd
```

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`

2.14 HMM tagger parameter file

Initial probabilities, transition probabilities, lexical probabilities, etc. The file has six sections: `<Tag>`, `<Bigram>`, `<Trigram>`, `<Initial>`, `<Word>`, and `<Smoothing>`. 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. The package includes a perl script that may be used to generate an appropriate config file from a tagged corpus. See the file `src/utilities/hmm_smooth.perl` for details.

- 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 transition probability that after a `0.AQ` sequence, the next word has `NC` tag.
  `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 "pi" parameters of the HMM.

  Each line is an initial probability, with the format `InitialState LogProbability`.

  Each state 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 to compute observation probability toghether with the tag probabilities above.

  Each line is a word probability `P(w)` with format `word LogProbability`. A special line for `<UNOBSERVED\_WORD>` must be included.

  E.g.

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

2.15 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-relax.perl` 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`.

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> <rer> <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];
```

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:
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 sencond word to the right is not a gerund (`VMG`). This rule is intended to solve the different behaviour of the Spanish word *lo* in sentences such as *si, lo estoy* or *lo estoy viendo*.

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

2.16 Chart 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 file name is given inside the angle (or round) brackets (e.g `VMIP3S0<"mylemmas.dat">`, `VMI*<"myforms.dat">`) 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.
- `@FLAT` Subtrees for "flat" non-terminal symbols are flattened when the symbol is recursive. Only the highest occurrence appears in the final parse tree.
- `@HIDEN` Non-teminal symbols specified under this directive will not appear in the final parse tree (their descendant nodes will be attached to their parent).

2.17 Dependency parser heuristic rules file

This file contains a set of heuristic rules to perform dependency parsing.

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

- 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 rigth. 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. Each line contains a rule, with the format:

  ```
  ancestor-label descendant-label label operation priority-spec
  ```

  where:

  - `ancestor-label` and `descendant-label` are the syntactic labels (either assigned by the chunk parser, or a new `label` created by some other completion rule) of two consecutive nodes in the tree.
  - `label` has two meanings, depending on the `operation` field value. For `top_left` and `top_right` operations, it states the label with with the root node of the resulting tree must be relabelled ("-" means no relabelling). For `last_left` and `last_right` operations, it states the label that the node to be considered "last" must have to get the subtree as a new child. If no node with this label is found, the subtree is attached as a new child to the root node.
  - `operation` is the way in which `ancestor-label` and `descendant-label` nodes are to be combined.
  - `priority-spec` is a specification of possible priority values for this rule, as detailed below.

  For instance, the rule:

  ```
  np pp - top_left  20
  ```

  states that if two subtrees labelled `np` and `pp` are found contiguous in the partial tree, the later is added as a new child of the former.

  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.

The `priority-spec` part of a rule defines the priority that will rank the applicable rules. Rules with low priority values will be applied earlier. The `priority-spec` consists of a list of zero or more pairs `context-condition value`, separated by semicolons. The last item in the list is a single integer value, and is required (i.e. the simplest possible `priority-spec` is a single integer value). Each context condition in the list is checked in order, and the priority value for the first matching condition is used for the rule. If no condition in the list matches, the last single value is used.

The context conditions are a sequence of labels separated with underscores, each label must match the label of one chunk in the partial tree. The condition must include a label `$$` which will match the pair of chunks that activated the rule. An `*` label matches any chunk.

For instance, the rule:

```
np pp - top_left vp_$$_adjp 20; $$_*_vp 10; 5
```

will be activated when an adjacent pair `np pp` is found, and will be ranked with priority 20 provided there is a `vp` chunk to the left of the focus pair, and a `adjp` chunk to its right. If not, it will get a priority of 10 if there is a `vp` chunk at the second right position, with any chunk in the first. If none of those patterns are matched, the rule will be assigned a priority of 5.

- Section `<GRLAB>` contains 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` may be `p` for parent or `d` for descendant), and `attribute` is one of the following:

  - `label`: chunk label (or PoS tag) of the node.
  - `side`: (left or right) position of the specified node with respect to the other.
  - `lemma`: lemma of dathe 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 requiered, 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.tonto=Edible  p.lemma=eat
```

states that if a `verb-phr` node has `eat` as lemma, and a descendant with a label starting by `np` and with a `Edible` property in EWN Top ontology, this dependence is to be labeled as `obj`.

- 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`). 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 4.5 section. The *WNFile* must be a BerkeleyDB indexed file, obtained with the same procedure from a source plain text file. This file must contain a sense per line, 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.

- 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 heuristic rules 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"
```

Chapter 3

Using the library from your own application

The library may be used to develop your own NLP application (e.g. a machine translation system, or an intelligent indexation module for a search engine).

To achieve this goal you have to link your application to the library, and access it via the provided API. Currently, the library provides only C++ API.

3.1 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.

3.1.1 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 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.
- `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 syntactyc dependency relationships between sentence constituents.

3.1.2 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, detectiion 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 associates to each of them a `dep_tree` object.

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).

3.2 Sample program

A very simple sample program using the library is depicted below. It reads text from stdin, morphologically analyzes it, and processes the obtained results. 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.

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

  // create analyzers
  tokenizer tk("myTokenizerFile.dat");
  splitter sp(false,0);

  // 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");

  // set required options
  opt.noQuantitiesDetection = true;  // deactivate quantities submodule

  // Data files for morphological submodules. Note that it is not necessary
  // to set opt.CurrencyFile, since quantities module was deactivated.
  opt.LocutionsFile="myMultiwordsFile.dat";       opt.SuffixFile="mySuffixesFile.dat";
  opt.ProbabilityFile="myProbabilitiesFile.dat";  opt.DictionaryFile="myDictionaryFile.dat";
  opt.NPdataFile="myNPdatafile.dat";              opt.PunctuationFile="myPunctuationFile.dat";

  // create the analyzer with the given set of maco_options
  maco morfo(opt);

  // create a hmm tagger
  hmm_tagger tagger("es", "myTaggerFile.dat");

  // get plain text input lines while not EOF.
  while (getline(cin,text)) {

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

    // 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);

    // analyze all words in all sentences of the list, enriching them with lemma and PoS
    // information. Some of the words may be glued in one (e.g. dates, multiwords, etc.)
    morfo.analyze(ls);

    // disambiguate words in each sentence of given sentence list.
    tagger.analyze(ls);

    // Process the enriched/disambiguated objects in the list of sentences
    ProcessResults(ls);
  }

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

  // morphologically enrich and disambiguate last sentence(s)
  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 file *FreeLing-build-dir*`/src/main/sample.cc`

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
```

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

More clues on how to use the freeling library from your own program may be obtained by looking at the source code of the main program provided in the package. The program is quite simple and commented, so it should be easy to understand what it does. The source can be found in file *FreeLing-build-dir*`/src/main/analyzer.cc`

Chapter 4

Extending the library with analizers 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 2.2.2).

4.1 Tokenizer

The first module in the processing chain is the tokenizer. As described in section 2.2.1, 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.

4.2 Morphological analyzer

The morphological analyzer module consists of several sub-modules that may require language customization. See section 2.2.1 for details on data file formats for each option.

4.2.1 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.

4.2.2 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`.

4.2.3 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`.

4.2.4 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.

4.2.5 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
```

4.2.6 Suffixed forms search

Forms not found in dictionary may be submitted to a suffix analysis to devise whether they are derived forms. The valid suffixes and their application contexts are defined in the suffix rule file referred by SuffixFile configuration option. See section 2.7 for details on suffixation 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.

4.2.7 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. See section 2.8 for format details.

4.3 HMM PoS Tagger

The HMM PoS tagger only requires an appropriate HMM parameters file, given by the TaggerHMMFile option. See section 2.14 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/hmm_smooth.perl` provided in FreeLing package.

4.4 Relaxation Labelling PoS Tagger

The Relaxation Labelling PoS tagger only requires an appropriate pseudo- constraint grammar file, given by the RelaxTaggerFile option. See section 2.15 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-relax.perl` 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`

4.5 Sense annotation

The sense annotation module uses a BerkeleyDB indexed file. This file may also be used by the dependency labeling module (see section 2.17).

It may be created for a new language with the `src/utilities/indexdict` program 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 useful if you are using the `synon` function in your dependency rules (see section 2.17). 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 2.17).

4.6 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 2.16 for format details.

4.7 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 HeuristicsFile option (or passed to the parser constructor). See section 2.17 for format details.

Bibliography

[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, Llus Mrquez, and Llus Padr. A simple named entity extractor using adaboost. In *Proceedings of CoNLL-2003 Shared Task*, Edmonton, Canada, June 2003.

[KVHA95] F. Karlsson, Atro Voutilainen, J. Heikkil, 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.es/~padro.

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