Outilex platform - user guide

Olivier Blanc and Matthieu Constant

Jan. 29, 2006

Contents

Chapter 1

Introduction

Outilex is a 4-year-research project funded by the French Industry Ministery. It gathers 4 academic institutions and 6 industrial organizations:

- Institut Gaspard Monge (IGM), Universite de Marne-la-Vallee (coordinator)
- Thales R&D
- Systran
- Lingway
- Thales Com
- LORIA
- Laboratoire d'Informatique de Paris 6 (LIP6)
- Universite de Rouen
- Centre de l'Energie Atomique (CEA)
- Langage Communication Information (LCI)

Started in October 2002, this project aims at developing a platform devoted to Natural Language Processing.

Chapter 2

Getting started

2.1 System Requirements

To compile the C++ programs, you need

- tool `jam` (de chez Perforcce)
- library `C++ Boost` (www.boost.org)
- library `libxml` (standard library)
- library `ICU` from IBM

To compile the Systran C++ tokenization module, you need an old version of `flex` 2.5.4 (package flex-old). Newer versions do not work.

To run the User Interface, you need the `Java Run time Environment` (1.5).

To vizualize text automata generated by the programs, you need to install `AT&T Graphviz` programs (especially, the program `dot`).

2.2 Outilex directory

Outilex directory includes the following files and directories:

- file `README.txt` (to get started)
- file `install-outilex` (script to compile and install C++ programs)
- file `outilexUI.jar` (to run user interface)
- file `clean-outilex` (scipt to clean compiled programs)

- directory `bin` (C++ compiled programs)
- directory `data` (some linguistic data provided with the platform)
- directory `docs` (documentation)
- directory `lingdef` (linguistic definitions of the set of tags used in dictionaries and graphs)

This directory also contains a log file (`outilex.log`) that includes all commands that have been launched from the interface. This can help users getting used with the syntax of the commands of the different programs.

2.3 Installation procedure

To install Outilex platform, you need to follow the steps described below:

- Go to Outilex directory;
- Run compilation by typing:

  `./install-outilex`

  Warning: You might have to change some environment parameters depending on you local system.

- Set `LINGDEF` environment variable by typing:

  ```
  LINGDEF=${OUTILEX_HOME}/lingdef/french/lingdef.xml
  export LINGDEF
  ```

  with `${OUTILEX_HOME}` the path of Outilex directory

  To avoid doing this operation everytime you want to run Outilex-platform, you should put these commands in your `.bashrc` file.

  Note: this operation is temporary and should be removed in the next versions of the Outilex platform.

2.4 Launch Outilex

2.4.1 Starting command

Outilex platform User Interface (UI) can be launched by typing:

```
java -jar outilexUI.jar
```

2.4.2 UI general description

The UI is composed of:

- a menu (on top), (Tool bar, coming soon...)
- a process/resource panel (on the left): to create personal processing chain with available linguistic resources,
- a content panel (on the right): to display linguistic resources and processing results

Important note:
Many functionalities can be run via popup menus (right-click on the mouse). Double-clicking on a resource (in the left panel), makes it display on the rightpanel.

2.4.3 Projects

Outilex platform works with a system of project. Each project is composed of a set of resources (texts, dictionaries and grammars). The Menu `Project` allows users to create, open and save projects. A project is associated with one language. This language selects a linguistic definition file. Presently, language is forced to "French". For example, if 'french' is the project language, the set of linguistic tags that will be used in the processings is defined in the file `lingdef/french/lingdef.xml`.

Chapter 3

Dictionaries

Dictionaries are sets of lexical entries associated with morphological, syntactic and semantic information. Lexical entries are either simple words or multiword units. Outilex platform allows users to edit their own dictionaries. There are several formats:

- an editable textual format: DELA format encoded in UTF-8 (extension `.dic`)
- an exchange format in XML (extension `.dic.xml.gz`)
- a binary format used by programs (extension `.idx`)

The different operations on dictionaries are gathered in the `Dictionary` menu of the platform.

3.1 DELA format

The DELA format has been defined in [**?**, **?**].

Syntax of an entry

An entry is defined on a single line as it is shown in the following examples:

```
car,.N+Conc:s/this is an example
eats,eat.V:P3s
Tony Blair,.N+Npr+Hum:ms
sincerely,.ADV
```

- The first element is the inflected form and is **obligatory** (`car` and `eats`).
- The second element (between symbols ',' and '.') is the lemma and is optional (e.g. `eat`). If it is not present, the lemma is considered to be the same as the inflected form (e.g. `car`).
- The third element is the part-of-speech and is **obligatory** (e.g. `ADV` for adverb, `N` for noun).
- The elements following symbol '+' are syntactic and semantic information and are optional (e.g. `Conc` for concrete, `Npr` for proper name, `Hum` for human).
- The character sequence following symbol ':' is a set of morphological information and are optional; each character stands for a piece of information (e.g. `P3s` stands for present [`P`] at the third person [`3`] of singular [`s`]).
- The sequence following symbol '/' is an optional comment (e.g. `this is an example`).

The tagset is free, as long as the writer follows the syntax defined above. This editable dictionaries are encoded in UTF-8 and their file extension is `.dic`.

3.2 XML format

3.2.1 Description

Outilex uses an UTF-8 XML exchange format (file extension `.dic.xml`). Tagset is defined in the lingdef (cf. section 3.2.2. All tags used must be defined in the lindef file.

Hereby is an example of an entry :

```
<entry>
 <lemma>abaissable</lemma>
  <pos name='adj'/>
  <inflected>
    <form>abaissable</form>
    <feat name='gender' value='masculine'/>
    <feat name='number' value='singular'/>
```

```
  </inflected>
  <inflected>
    <form>abaissable</form>
    <feat name='gender' value='feminine'/>
    <feat name='number' value='singular'/>
  </inflected>
  <inflected>
    <form>abaissables</form>
    <feat name='gender' value='masculine'/>
    <feat name='number' value='plural'/>
  </inflected>
  <inflected>
    <form>abaissables</form>
    <feat name='gender' value='feminine'/>
    <feat name='number' value='plural'/>
  </inflected>
</entry>
```

To avoid memory space feeding, XML dictionaries are compressed and their file extension is `.dic.xml.gz`.

3.2.2 Lingdef

The Lingdef file defines the tagset that can be used in Outilex XML dictionaries. It is also encoded in XML. This file is located in the directory `<language>` (e.g. `french`) of the directory `lingdef`.

Hereby is an example of the definition of the tagset used for nouns:

```
<!-- nouns -->

<attrtype name='nounsubcat' type='enum'>
  <value name='pred' alias='Pred'/>
  <value name='conc' alias='Conc,concret'/>
  <value name='abst' alias='Abst,abstract,abs'/>
  <value name='hum'  alias='Hum,human'/>
  <value name='anl'  alias='Anl,animal'/>
  <value name='tps'  alias='Tps,temporal'/>
  <value name='top'  alias='Top,toponym'/>
  <value name='unit' alias='Unit'/>
  <value name='num'  alias='Nnum,numeral'/>
  <value name='dnom' alias='Dnom,detnom'/>
```

```
</attrtype>

<attrtype name='collective' type='bool'>
  <true alias='Coll'/>
</attrtype>

<attrtype name='proper' type='bool'>
  <true alias='pr,Pr,Prenom'/>
</attrtype>

<pos name='noun' cutename='N'>
  <attribute name='subcat' type='nounsubcat' shortcut='yes'/>
  <attribute name='gender' type='gender' shortcut='yes'/>
  <attribute name='number' type='number' shortcut='yes'/>
  <attribute name='proper' type='proper' default='false' shortcut='yes'/>
  <attribute name='coll'   type='collective' shortcut='yes'/>
</pos>
```

3.3 Operations

3.3.1 Editing a dictionary

For editing a new or existing dictionary, you need to click on items `new` or `open` in the menu `Dictionary`. Then a text editor containing the dictionary will appear in the content part of the UI. After having edited the dictionary, you can save it by clicking on `save` or `save as` in the same menu. Dictionaries are saved in UTF-8.

3.3.2 Converting DELA in XML

Converting a DELA dictionary into an XML one requires the definition of the file `delaf-corresp` that defines the correpondance between tags used in the DELA dictionary and tags used in the XML dictionary. This file is located in the directory `<language>` (e.g. `french`) of the directory `lingdef`.

Below is a sample of this file:

```
POS A        adj
POS ADV      adverb
POS DET      det
POS N        noun
POS PREP     prep
```

```
POS PREPADJ  prepadj
POS PREPDET  prepdet
POS PREPPRO  preppro
POS PRO      pronoun
POS V        verb

flex f PREPDET,DET,PRO,A,N,V form gender e feminine
flex m PREPDET,DET,PRO,A,N,V form gender e masculine
flex p PREPDET,DET,PRO,A,N,V form number e plural
flex s PREPDET,DET,PRO,A,N,V form number e singular

flex 1 PRO,V form person e 1
flex 2 PRO,V form person e 2
flex 3 PRO,V form person e 3

....
```

For converting a DELA dictionary into an XML dictionary, you need to click on item `Transcode/DELA -> XML` in the menu `Dictionary`. The selected DELA dictionary will then be converted in a compressed XML format and be displayed in the content part of the UI. (Program used: `delaf2xml.sh`, cf. section 7.5).

3.3.3 Indexing dictionaries

To be used in the Outilex text processes, the dictionaries must be indexed into binary files that represent the dictionaries in the form of minimized finite-state transducers. These files have the extension `.idx`.

To do so, you need to click on item `indexing` in the menu `Dictionary`. The C++ program that is launched is either `dic-index` for compressed XML dictionaries (cf. section 7.6) or `dela-index` for DELA dictionaries (cf. section 7.4).

This process through the interface will produce a file `<name>.idx` if input dictionaries are named `<name>.dic` or `<name>.dic.xml.gz`.

3.3.4 Add selected dictionary to project

You can add the selected dictionary to your current project by clicking on item `Add to project` in the menu `Dictionary`, ONLY IF IT HAS ALREADY BEEN INDEXED.

Chapter 4

Grammars

Grammars used in Outilex are equivalent to Recursive Transition Networks [**?**]. They are in the form of graphs. Their symbols used can be lexical values, lexical masks or call to sub-graphs. They can also contain outputs and weights. Outilex platform includes a graph editor developed from Unitex sources [**?**].

Temporary, for more details on graph edition, see the Unitex manual (file `manuelunitex.pdf`)

4.1 A simple graph

4.2 Subgraphs

4.3 Ouputs and weights

If you want to associate an output with a box (input), you need to use the `output` text field. As in the input, symbol '+' can play the role of a "line breaker". Weights can also be added with the following syntax in the the `output` text field:

```
<output>/<weight>
```

where `<output>` is a string defining the output associated with an input, `<input>` is a positive real number. For instance,

```
Noun/3.0
```

The weight of a path of a graph is the sum of all its weights. By default, if the weight of an input label is missing, its value is 0.

For instance, the graph in figure 4.1 recognizes sequences `noun-adj`. But when a compound noun (`noun+comp`) is also recognized on the same sequence of text, a priority is given to this last analysis because it is assigned a weight of 1 (0 for the other).

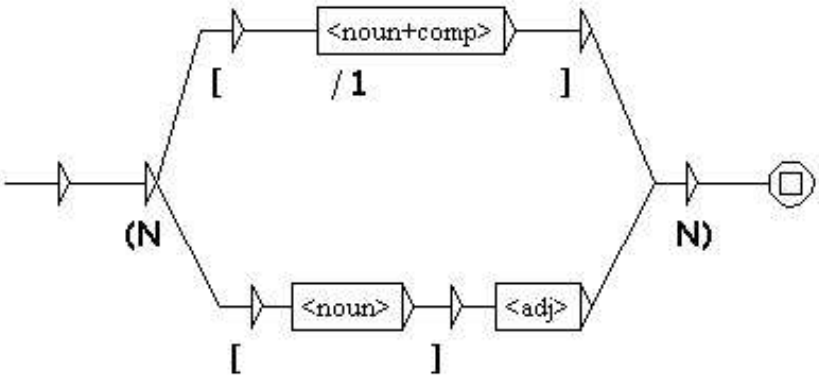

Figure 4.1: Use of weights

4.4 Normalization graphs

A normalization graph is a graph such that when applied to a text-automaton it normalizes some sequences like `de` as shown in the following example:

4.5 Decoration graphs

A decoration graph is a graph such that when applied to a text automaton, new transitions corresponding to new analyses are added to the initial text automaton.

Below are two examples:

These graphs have a special format. Linguistic entities that have to be analysed must be delimited in the graph by square brackets in the output like in the graphs shown above. The part-of-speech (e.g. `N` for "noun" in figure 4.3, `CV` for "verbal complex" in figure 4.4 `PPV` for "preverbal pronoun" in figure 4.4) to be assigned to these entities must be put just after the opening square brackets. Attributes to this part-of-speech can also be added by adding outputs with the following syntax `+<attribute>` (e.g. +Npr

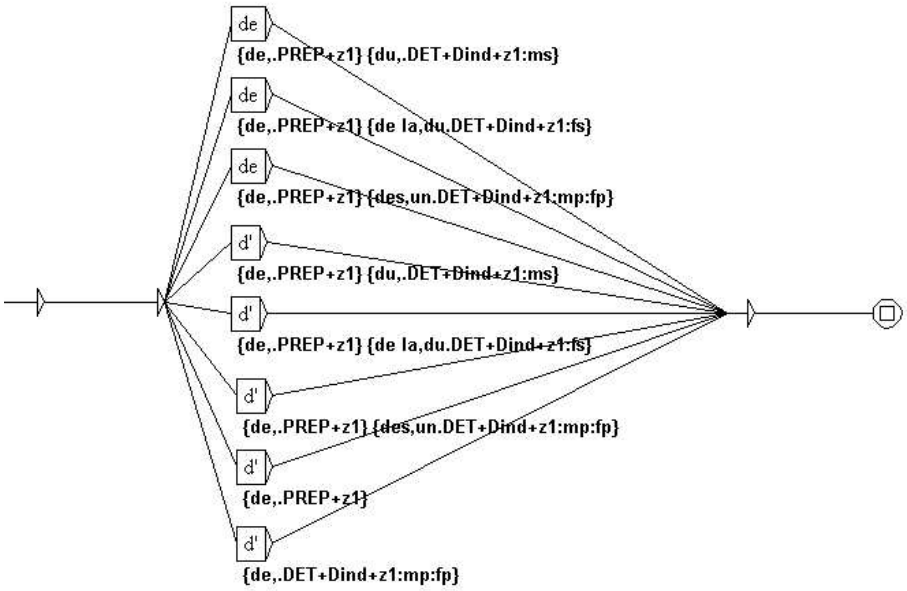

Figure 4.2: Normalization graph

means "proper name" in figure 4.3). For instance, the graph in figure 4.3 recognizes named entities which are tagged `N+Npr`.

A complex entity can inherit from attributes from its elements (e.g. lemma, mode, gender, and so on.). This can be indicated in the decoration graph as an output with the following syntax `+^<attribute>`. Such an example is shown in figure 4.4: when recognized in a text, the pattern `<avoir.verb> <verb+ppast>`[1] is analysed as a `CV` (verbal complex) whose mode is the mode of `avoir` and whose lemma is the one of the verb at the past participle. For instance, the sequence `a lu` would be analysed as a `CV` whose mode is `ind` (for indicative) and whose lemma is `lire`.

[1] the verb `avoir` followed by a verb at the past participle

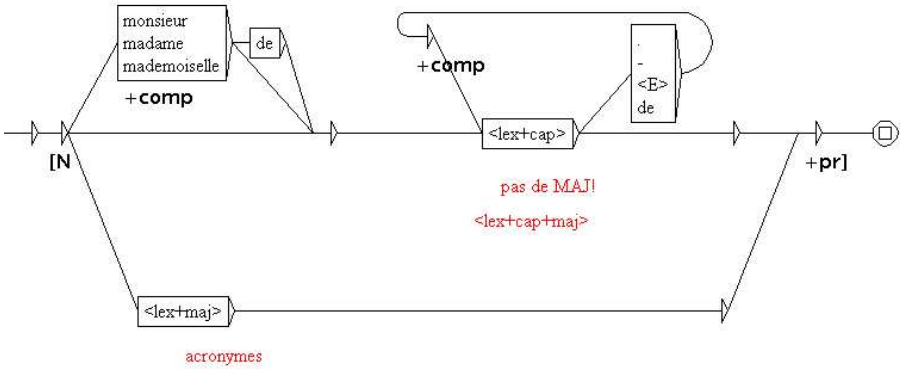

Figure 4.3: Recognition of named entities

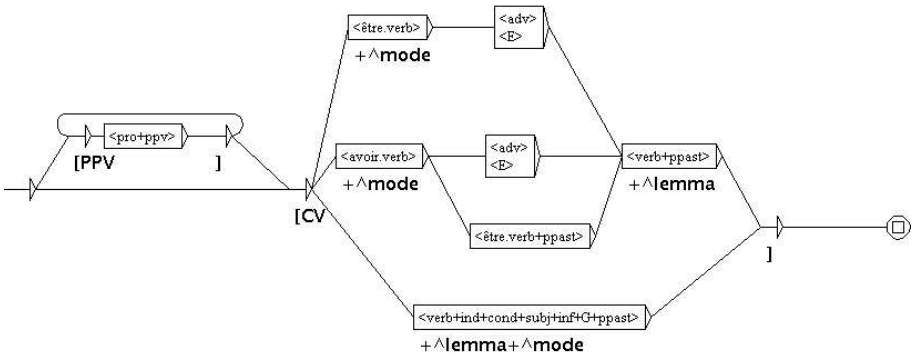

Figure 4.4: Recognition of verbal complexes

Chapter 5

Text FSA

The Text FSA is used to represent text ambiguity. For each sentence, there is an automaton that represents its possible analyses. Grammar application programs all use it as input.

Chapter 6

Text processing

Outilex platform allows users to process texts using linguistic resources such as dictionaries and grammars. The left part of the UI permits to create your own desired chain.

Processing a text is:

- segment text in tokens and sentences (check box `segmentation`),
- applying dictionaries on segmented text and obtaining a text automaton representing the possible analyses for each sentence (check box `apply dictionaries`),
- normalizing text-fsa by applying normalization graphs (check box `normalize`).
- apply a cascadus of grammars in the form of graphs on the text automaton, resulting to a new text automaton (check box `apply graph cascadus`),
- applying a grammar to the text automaton to obtain a concordance or a modified text (e.g. an annotated text) (check box `locate pattern`).

6.1 Text segmentation

The text segmentation process creates a directory associated to the text (`<text>.dir`) and outputs a segmented text `<text>.segmentation` put in this directory.

6.2 Dictionary application

You must insert and select dictionaries you need (click on button `more`, click on button `less` to close). The process will generate a text automaton,`<text>.fsa`, in the text directory. A copy of it is also made (file `<text>-0.fsa`)

6.3 Normalize text automaton

This operation must be run after dictionary application and text-automaton construction. It applies the graph `Norm.xgrf` in the directory `lingdef/<language>` where `<language>` is the current language. This process generates a new "normalized" text-automaton (file `<text>-norm.fsa`).

6.4 Grammar application

You need to define a list of graphs that will be applied in cascadus on `<text>.fsa`. Each iteration `j` will generate a new text automaton `<text>-j.fsa`. The final automaton is `<text>-final.fsa`. A copy of it is made in file `<text>.fsa`;

`<text>.fsa` is actually the current text automaton to be processed.

6.5 Locate pattern

You need to select a graph to be applied and the type of result you want.

Chapter 7

C++ Programs

The Outilex platform is made of a set of independant C++ programs. This chapter defines their different prototypes.

7.1 apply-dic

```
apply-dic -dic <dic1> [<prio1>] [-dic <dic2> [<prio2>] ...]
   [-imaj][-icase] [-imark][-l <lingdef>][-o <out>] <tokfile>
```

This program applies a set of dictionaries `<dicj>` (extension `.idx`) with different priorities `<prioj>` (real numbers, by default, `10`) to a segmented text `<tokfile>`. It outputs a text-fsa `<out>` (by default, the name of the text file with the extension `.fsa`). It uses a lingdef file `<lingdef>`. Options could be:

- -imaj: ignore case in texts but not in dictionaries;
- -icase: ignore case in dictionaries and in texts;
- -imark: ignore diacritics in texts and in dictionaries.

7.2 concordancer

```
concordancer -l <lingdef> -gram <gram> [-v][-longest-match]
   [-tags][-tree][-w][-m][-ipath][-iout][-o <outputres>] <txtfsa>
```

with options :

`<txtfsa>` : input text fsa

-gram `<gram>` : wrtn grammar to apply

-o <concord> : name of the resulting concordance index file (default to concord.idx)

-longest-match : keep only longest matching sequences

-tags : display morpho-syntactic tags

-tree : display syntactic tree

-w : display weights of matching sequences

-m : merge grammar's outputs into concordances

-all : shortcut for : -tags -tree -w -m

-ipath : keep only one concordance for the same text segment (can be a lot faster for ambigous grammars)

-v : verbose mode (for debugging)

It applies a compiled wrtn grammar <gram> (extension .wrtn) to a text fsa <txtfsa> and saves the matching sequences index into a file <outputres> (default to concord.idx), which can be processed by make-concord-html. There exist different options that are described above.

7.3 decore-fsa

```
decore-fsa -l <lingdef> -rtn <fst> [-v][-ipath][-iout][-o <outputres>]
<txtfsa>
```

This program applies a compiled decoration grammar fst (extension .wrtn) to the text-fsa <txtfsa>. It outputs a new version of txtfsa,<outputres>, with new transitions whenever new analyses have been found by applying grammar fst. It requires the lingdef file <lingdef>.

7.4 dela-index

```
dela-index <dela> -corresp <corresp> [-r <ratio>][-o <index>]
```

This program compresses an UTF-8 DELA dictionary <dela> into into an IDX dictionary using the tag correspondance file <corresp>. The output file is <index>.

7.5 delaf2xml.sh

```
delaf2xml.sh -c <corresp> <dela>
```

This program converts an UTF-8 DELA dictionary `<dela>` (extension `.dic`) into a compressed XML dictionary (`<dela>.xml`) using the tag correspondance file `<corresp>`.

7.6 dic-index

```
dic-index [-validate] [-ratio <r>] <dicofile>
```

This program compresses the dictionary `<dicofile>` (extension `.dic.xml.gz`) into an IDX dictionary. The output file is the name of `<dicofile>` with the extension (`.idx`). For instance, if `<dicofile>` is `dico.dic.xml.gz`, the output would be `dico.dic.idx`.

7.7 make-concord-html

```
make-concord-html <concordidx>
   [-left <left-size>][-right <right-size>][ -o <res>][-dontsort]
```

It constructs an html concordance from the index concordance file `<concordidx>` with a left context of `<left-size>` characters (default: 50) and a right context of `<right-size>` characters (default: 80). Optionally, the concordance can be put in the text order (option `-dontsort`); by default, it is sorted. The result is put in the file `<res>` (default: `concord.html`).

7.8 make-wrtn

```
make-wrtn [-l <lingdef>] <axiom>
```

This program compiles the grammar defined by the main XGRF graph `<axiom>` and its sub-graphs into a unique XML file representing a Weighted Recursive Transition Network (WRTN) with the extension `.wrtn`. It uses the lingdef file `<lingdef>` to interpret semantically the symbols of the graphs. For instance, if `<axiom>` is `main.xgrf`, the output would be `main.wrtn`.

7.9 tfsa2dot

```
tfsa2dot -l <lingdef> [-o <output>] <txtfsa> -n <sentenceno>
```

This program converts the <sentenceno>-th sentence of text-fsa txtfsa (extension .fsa) into the file <output> describing an automaton with the DOT format, using the lingdef file <lingdef>. By default, the output file is named sentence-<sentenceno>.dot.

7.10 tokenization

```
tokenization <text>
```

This program is used to segment a text <text> in tokens, sentences and paragraphs. <text> is the input text file name and can be either in TXT format or HTML format. Outputs are <text>.segmentation, <text>.tokenization and <text>.postfilter.

7.11 transduct-fsa

```
transduct-fsa -l <lingdef> -gram <fst>
   [-lgst][-ipath][-iout][-dontsurf][-o <outputres>] <txtfsa>
```

This program applies a normalization wrtn transducer <fst> on a text-fsa <txtfsa>. It produces a new text-fsa <outputres>.

7.12 wrtn-flatten

```
wrtn-flatten <rtn> [-maxdepth <N>]
```

This program flattens a compiled grammar <rtn> into a finite state automaton (or transducer). Whenever not possible, it makes an approximation by limiting the maximum depth (<N>).

7.13 wrtn-txt-transduct

```
wrtn-txt-transduct -l <lingdef> -gram <fst>
   [-ipath|-iout][-html|-txt][-m|-r|-i][-o <outputres>] <txtfsa>
```

This program applies wrtn transducer <fst> to a text fsa <txtfsa> and generates a new text from the orignal one depending on the chosen option:

- -m for merging outputs in the text when finding matching sequences;

- -r for replacing matching sequences by the associated outputs in the new text;
- -i for ignoring outputs: new text is the original one (USELESS!!!).

The output text `<outputres>` can be either in HTML (`-html`) or in TXT (`-txt`). It requires the lingdef file `<lingdef>`.

Chapter 8

API library