Documentation of the Open-Source Shallow-Transfer Machine Translation Platform *Apertium*

AUTHORS:
Mikel L. Forcada
Boyan Ivanov Bonev
Sergio Ortiz Rojas
Juan Antonio Pérez Ortiz
Gema Ramírez Sánchez
Felipe Sánchez Martínez
Carme Armentano-Oller
Marco A. Montava
Francis M. Tyers

EDITOR:
Mireia Ginestí Rosell

Departament de Llenguatges i Sistemes Informàtics
Universitat d'Alacant

March 10, 2010

Contents

Introduction

This documentation describes the Apertium platform, one of the open-source machine translation systems which originated within the project "Open-Source Machine Translation for the Languages of Spain" ("Traducción automática de código abierto para las lenguas del estado español"). It is a shallow-transfer machine translation system, initially designed for the translation between related language pairs, although some of its components have been also used in the deep-transfer architecture (*Matxin*) that has been developed in the same project for the pair Spanish-Basque. *Apertium* can translate at present between the pairs Spanish-Galician, Spanish–Catalan[1] Catalan-Occitan, Catalan-French, and can be used to build translators between other related language pairs, such as Danish-Swedish,Czech–Slovak, etc.

Existing machine translation systems available at present for the pairs `es-ca` and `es-gl` are mostly commercial or use proprietary technologies, which makes them very hard to adapt to new usages; furthermore, they use different technologies across language pairs, which makes it very difficult to integrate them in a single multilingual content management system.

One of the main novelties of the architecture described here is that it has been released under open-source licenses (in most cases, GNU GPL; some data still have a Creative Commons license) and is distributed free of charge. This means that anyone having the necessary computational and linguistic skills will be able to adapt or enhance the platform or the language-pair data to create a new machine translation system, even for other pairs of related languages. The licenses chosen make these improvements immediately available to everyone. We therefore expect that the introduction of this of open-source machine translation architecture will solve some of the mentioned problems (having different technologies for different pairs, closed-source architectures being hard to adapt to new

[1] With the name *Catalan* we refer also to the Valencian dialectal variant of this language.

uses, etc.) and promote the exchange of existing linguistic data through the use of the XML-based formats defined in this documentation. On the other hand, we think that it will help shift the current business model from a license-centred one to a services-centred one.

It is worth mentioning that "Open-Source Machine Translation for the Languages of Spain" was the first large open-source machine translation project funded by the central Spanish Government, although the adoption of open-source software by the Spanish governments is not new.

This documentation describes in detail the characteristics of the Apertium platform, and is organized as follows:

- Chapter 1: **general description** of the shallow-transfer machine translation system and of the modules that make it up.
- Chapter 2: description of the **format of the data stream** that circulates from one module to the next one.
- Chapter 3: **specification of the modules** of the system. For each module there is a description of: the *program* and its characteristics, the *format of the data* that the module uses, and the *compilers* used for it. This chapter is divided in the following sections:
  - Section 3.1: *Lexical processing modules*, where the morphological analyser, the lexical transfer module, the morphological generator and the post-generator are described (Section 3.1.1), along with the format of the dictionaries used by these modules (section 3.1.2) and their compilers (section 3.1.3)
  - Section 3.2: *Part-of-speech Tagger*, which describes the tagger (Section 3.2.1) and the format of the linguistic data used by the tagger (section 3.2.2.
  - Section 3.3: *Pre-transfer module*, which describes the module that runs before the structural transfer module to perform some operations on multiword units
  - Section 3.5: *Structural transfer module*, where there is a description of the program (section 3.5.2) and of the format of the structural transfer rules (Section 3.5.4).
  - Section 3.6: *De-formatter and Re-formatter*, which describes these modules (section 3.6.1), the rules for format processing (section 3.6.2) and how these modules are generated (Section 3.6.3)
- Chapter 4: it describes the way to **install the system** and to **run the translator**.

- Chapter 5: here you will find an explanation of how to **modify the linguistic data** used by the translator, that is, the dictionaries, the part-of-speech disambiguation data and the structural transfer rules created in this project for Spanish, Catalan and Galician. Furthermore, it contains a brief description of the characteristics of the available data for these three language pairs.

The files which this documentation refers to can be found at and downloaded from the project web page in Sourceforge: `http://sourceforge.net/projects/apertium/`. From this page you can download the packages needed for installation, as well as view the individual files in the SVN (main) and CVS (residual) repositories of the project. The machine translation systems for the different language pairs can also be tested on the Internet at `http://www.apertium.org/`.

Acknowledgements:

The present work has benefited from the contribution of many people and institutions:

- The Spanish Ministry of Industry, Commerce and Tourism has funded the development of this toolbox through the projects "Open-Source Machine Translation for the Languages of Spain", code FIT-340101-2004-3, and its extension FIT-340001-2005-2, and "EurOpenTrad: Open-Source Advanced Machine Translation for the European Integration of the Languages of Spain", code FIT-350101-2006-5, all of them belonging to the PROFIT program.

- Workers and scholars from other machine translation projects at the Universitat d'Alacant: Míriam Antunes Scalco, Carme Armentano i Oller, Raül Canals i Marote, Alicia Garrido Alenda, Patrícia Gilabert i Zarco, Maribel Guardiola i Savall, Javier Herrero Vicente, Amaia Iturraspe Bellver, Sandra Montserrat i Buendia, Hermínia Pastor Pina, Antonio Pertusa Ibáñez, Francisco Javier Ramos Salas, Marcial Samper Asensio and Miguel Sánchez Molina.

- The companies and institutions that have funded these other machine translation projects: Spanish Ministry of Science and Technology, Caja de Ahorros del Mediterráneo, Universitat d'Alacant and Portal Universia, S.A.

- Iñaki Alegria, from the Ixa group of the Euskal Herriko Unibertsitatea (University of the Basque Country), for his close reading of previous versions of this document.
- Google, who, through the Google Summer of Code programme, funded the development of several new modules.

Chapter 1

The shallow-transfer machine translation engine

This chapter describes briefly the structure of the shallow-transfer machine translation engine, which is largely based on that of the existing systems for Spanish–Catalan interNOSTRUM [2, 5, 4] and for Spanish–Portuguese Traductor Universia [7, 17], both developed by the Transducens group of the Universitat d'Alacant. It is a classical indirect translation system that uses a partial syntactic transfer strategy similar to the one used by some commercial MT systems for personal computers.

The design of the system makes it possible to produce MT systems that are *fast* (translating tens of thousands of words per second on ordinary desktop computers) and that achieve results that are, in spite of the errors, reasonably intelligible and easily correctable. In the case of related languages such as the ones involved in the project (Spanish, Galician, Catalan), a mechanical word-for-word translation (with a fixed equivalent) would produce errors that, in most cases, can be solved with a quite rudimentary analysis (a morphological analysis followed by a superficial, local and partial syntactic analysis) and with an appropriate treatment of lexical ambiguities (mainly due to homography). The design of our system follows this approach with very interesting results. The Apertium architecture uses finite-state transducers for lexical processing, hidden Markov models for part-of-speech tagging and finite-state-based chunking for structural transfer.

The translation engine consists of an 8-module *assembly line*, which is represented in Figure 1.1. To ease diagnosis and independent testing, modules communicate between them using text streams. This way, the input and output of the modules can be checked at any moment and, when an error in the translation process is detected, it is easy to test the output

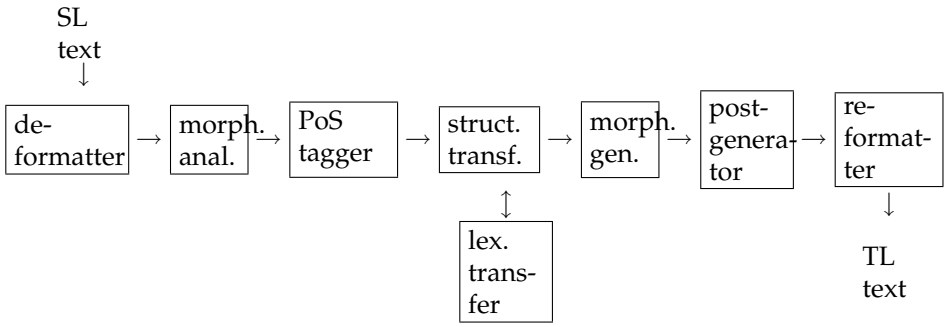


Figure 1.1: The eight modules that build the assembly line of the shallow-transfer machine translation system.

of each module separately to track down the origin of the error. At the same time, communication via text allows for some of the modules to be used in isolation, independently from the rest of the MT system, for other natural-language processing tasks, and enables the construction of prototypes with modified or additional modules.

We decided to encode linguistic data files in XML[1]-based formats due to its interoperability, its independence of the character set and the availability of many tools and libraries that make easy the analysis of data in this format. As stated in [8], XML is the emerging standard for data representation and exchange in Internet. Technologies around XML include very powerful mechanisms for accessing and editing XML documents, which will probably have a significant impact on the development of tools for natural language processing and annotated corpora.

The modules Apertium consists of are the following:

- The *de-formatter*, which separates the text to be translated from the format information (RTF, HTML, etc.); its specification can be found in Section 3.6.1. Format information is encapsulated so that the rest of the modules treat it as blanks between words. For example, for the HTML text in Spanish:

  ```
  es <em>una señal</em>
  ```

  ("it is a sign") the de-formatter encapsulates in brackets the HTML tags and gives the output:

[1] `http://www.w3.org/XML/`

```
es [<em>]una señal[</em>]
```

The character sequences in brackets are treated by the rest of the modules as simple blanks between words.

- The *morphological analyser*, which tokenizes the text in *surface forms* (SF) (lexical units as they appear in texts) and delivers, for each SF, one or more *lexical forms* (LF) consisting of *lemma* (the base form commonly used in classic dictionary entries), the *lexical category* (noun, verb, preposition, etc.) and morphological inflection information (number, gender, person, tense, etc.). Tokenization of a text in SFs is not straightforward due to the existence, on the one hand, of contractions (in Spanish, *del*, *teniéndolo*, *vámonos*; in English, *didn't*, *can't*) and, on the other hand, of lexical units made of more than one word (in Spanish, *a pesar de*, *echó de menos*; in English, *in front of*, *taken into account*). The morphological analyser is able to analyse these complex SFs and treat them appropriately so that they can be processed by the next modules. In the case of contractions, the system reads a single surface form and gives as output a sequence of two or more lexical forms (for instance, the Spanish preposition-article contraction *del* would be analysed into two lexical forms, one for the preposition *de* and another one for the article *el*). Lexical units made of more than one word (multiwords) are treated as single lexical forms and processed specifically according to its type.[2]

  Upon receiving as input the example text from the previous module, the morphological analyser would deliver:

```
^es/ser<vbser><pri><p3><sg>$[ <em>]
^una/un<det><ind><f><sg>/unir<vblex><prs><1><sg>/unir
<vblex><prs><3><sg>$
^señal/señal<n><f><sg>$[</em>]
```

  where each surface form has been analysed into one or more lexical forms: *es* has been analysed as one SF with lemma *ser* ("to be"), whereas *una* receives three analyses: lemma *un* ("one"), determiner, indefinite, feminine, singular; lemma *unir* ("to join"), verb in subjunctive present, 1st person singular, and lemma *unir*, verb in subjunctive present, 3rd person singular.

  This module is generated from a source language (SL) morphological dictionary, the format of which is specified in section 3.1.2.

[2]For more information about the treatment of multiwords, please refer to page 43.

- The *part-of-speech tagger* chooses, using a statistical model (hidden Markov model), one of the analyses of an ambiguous word according to its context; in the previous example, the ambiguous word would be the surface form *una*, which can have three different analyses. A sizeable fraction of surface forms (in Romance languages, for instance, around one out of every three words) are ambiguous, that is, they can be analysed into more than one lemma, more than one part-of-speech or have more than one inflection analysis, and are therefore an important source of translation errors when the wrong equivalent is chosen. The statistical model is trained on representative source-language text corpora.

  The result of processing the example text delivered by the morphological analyser with the part-of-speech tagger would be:

  ```
  ^ser<vbser><pri><p3><sg>$[ <em>]^un<det><ind><f><sg>$
  ^señal<n><f><sg>$[</em>]
  ```

  where the correct lexical form (determiner) has been selected for the word *una*.

  The specification of the part-of-speech tagger is in section 3.2.

- The *lexical transfer module*, that uses a bilingual dictionary and is called by the structural transfer module, reads each LF of the SL and delivers the corresponding target language (TL) lexical form. The dictionary contains a single equivalent for each SL lexical form; that is, no word-sense disambiguation is performed . Multiwords are translated as a single unit. The lexical forms in the running example would be translated into Catalan as follows:

  ```
  ser<vbser> ⟶ ser<vbser>
  un<det> ⟶ un<det>
  señal<n><f> ⟶ senyal<n><m>
  ```

  This module is generated from a bilingual dictionary, which is described in Section 3.1.2.

- The *structural transfer module*, which detects and processes patterns of words (chunks or phrases) that need special processing due to grammatical divergences between the two languages (gender and number changes, word reorderings, changes in prepositions, etc.).

This module is generated from a file containing rules which describe the action to be taken for each pattern. In the running example, the pattern formed by `^un<det><ind><f><sg>$ ^señal<n><f><sg>$` would be detected by a determiner–noun rule, which in this case would change the gender of the determiner so that it agrees with the noun; the result would be:

```
^ser<vbser><pri><p3><sg>$[ <em>]^un<det><ind><m><sg>$
^senyal<n><m><sg>$[</em>]
```

The format of the structural transfer rules file, inspired in the one described in [5], is specified in Section 3.5.

- The *morphological generator*, that, from a lexical form in the target language, generates a suitably inflected surface form. The result for the example phrase would be:

  ```
  és[ <em>]un senyal[</em>]
  ```

  This module is generated from a morphological dictionary, which is described in detail in Section 3.1.2.

- The *post-generator*, that performs some orthographic operations in the TL such as contractions and apostrophations, and which is generated from a transformation rules file the format of which is very similar to the format of the mentioned dictionaries. Its format is specified in Section 3.1.2. In the example text there is no need to perform any contraction or apostrophation.

- The *re-formatter*, which restores the original format information into the translated text; the result for the running example would be the correct conversion of the text into HTML format:

  ```
  és <em>un senyal</em>
  ```

  The specification of the re-formatter is described in Section 3.6.1.

The four lexical processing modules (morphological analyser, lexical transfer module, morphological generator and post-generator) use a single compiler, based on a class of *finite-state transducers* [4], in particular, letter transducers [14, 11]; its characteristics are described in Section 3.1.3.

Chapter 2

Format specification of the data stream between modules

2.1 Introduction

The format of the data that circulate between the engine's modules has to be specified so that document processing is more effective and transparent. The proposed system design (see Section 1) imposes the need to use three different data stream types, as shown in Figure 2.1.

The stream format is text-based to facilitate, among other things, the diagnosis of possible system errors, since it is easy to manipulate the stream in order to reproduce the phenomena that are to be tested, and change it to see the result. Other benefits of using text streams are that it is possible to test independently the output of each module, and that it allows for fast building of prototypes to test the system's global performance, the validity of linguistic data, etc.

The data stream types are:

- *Data stream with format:* It is the text in its original format, with no further marks: XML, ANSI text, RTF, HTML, etc. Since it is the original format of the documents, nothing needs to be specified about it except the name of the format.

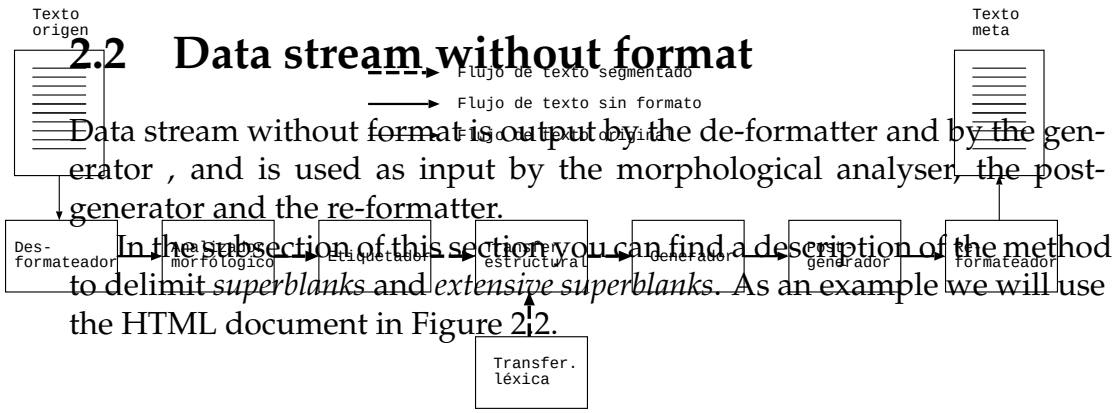
- *Data stream without format:* It is the text with *superblanks*, that is, with special characters that encapsulate the format (see Section 3.6.1); superblanks are treated by the linguistic modules as blanks between words (with some exceptions). This is the format generated by the de-formatter and used by the re-formatter when generating the final translated document.

Figure 2.1: The different data stream types in the machine translation system. See the text for its description.

- *Segmented data stream:* In this format, apart from superblanks, lexical units that are to be translated are delimited also with special characters. These characters are put by the morphological analyser and deleted by the generator, which delivers the final surface forms.

We describe next the characteristics of the data stream used between the modules of the translator, that is, the second and the third stream types. In general terms, it is a plain text format marked with characters that have a special meaning. This format is intended for the processing in servers that translate large volumes of text.

Some of the formats that the engine can process may contain extensive blocks of information in binary format —RTF for instance, that may include bitmap images—. To enable an efficient processing of this type of documents, we designed a way to extract this information and restore it after translation has been performed; see Section 3.6.1 for a complete description.



2.2 Data stream without format

Data stream without format is output by the de-formatter and by the generator , and is used as input by the morphological analyser, the post-generator and the re-formatter.

In the subsection of this section you can find a description of the method to delimit *superblanks* and *extensive superblanks*. As an example we will use the HTML document in Figure 2.2.

```
<html>
  <head>
    <title>Title</title>
  </head>
  <body>
    <p>Divided
       sentence</p>
  </body>
</html>
```

Figure 2.2: Example of HTML document

The structural elements that must include this data stream type are the following:

- *Superblanks*. Blocks that contain segments of format information included in the documents, when these are short.
- *Extensive superblanks*. Marks that are used to specify external documents that include segments of format information for the document being processed, when these segments are long.
- *Text*. The document text that can be translated.
- *Artificial sentence endings*. When the format in the document suggests a sentence separation that is not signalled by any punctuation mark (for instance, titles with no full stop at the end, or the content of cells in a table), the format processing must have a mechanism (invisible for the user) that enables the marking of these sentence endings.
- *Special characters protection (for non-XML stream)*. Characters that must be protected to avoid conflict with the ones used in the data stream format.

2.2.1 Stream format

This format is based on the one used in the machine translation systems interNOSTRUM [2, 5, 4] and Traductor Universia [7, 17].

In this stream type, the characters `[` and `]` are used to indicate *superblanks*, as shown in the following example:

```
[superblank content]
```

In the case of *extensive superblanks*, the file name is specified using the at sign `@`:

```
[@file name]
```

The *text* is outside the superblank marks.

Artificial sentence endings are expressed by a full stop and an empty superblank right after it.

```
.[]
```

The following table shows the **protected characters**:

Name	Character	Protected form	Meaning
At	`@`	`\@`	External superblank
Slash	`/`	`\/`	Divider of meanings
Backslash	`\`	`\\`	Protection character
Caret	`^`	`\^`	Beginning of LF
Opening square bracket	`[`	`\[`	Beginning of blank
Closing square bracket	`]`	`\]`	End of blank
Dollar	`$`	`\$`	End of LF
Greater than	`>`	`\>`	Begin. of morph. symbol
Less than	`<`	`\<`	End of morph. symbol

Figure 2.3 shows the document in Figure 2.2 after encapsulation.

```
[<html>
  <head>
    <title>]Title.[][</title>
  </head>
  <body>
    <p>]Divided[
       ]sentence.[][</p>
  </body>
<html>]
```

Figure 2.3: The document in Figure 2.2 with format encapsulated using square brackets

2.3 Segmented data stream

Segmented data stream is the stream that circulates between the modules that handle linguistic information in the translation engine. In this stream, words are delimited and labelled. There are two types of segmented stream:

- *Ambiguous segmented stream*. Its main characteristic is that words have a surface form and potentially more than one lexical form (lexical multiform). This stream type is the format in which the morphological analyser provides the input data for the part-of-speech tagger (see diagram 3.2 in page 58 for a detailed description of ambiguous segmented stream).

- *Unambiguous segmented stream*. It has only one lexical form for each word and it does not include the surface form. This is the format in which data circulate from the part-of-speech tagger to the transfer module, and from this module to the generator (see diagram 3.3 in page 63 for a detailed description of the format of unambiguous segmented stream).

Furthermore, besides the information already marked in the data stream without format, the new stream has to enable marking of the following information:

- *Lexical units*. A lexical unit is made of a surface form (in the case of ambiguous segmented stream) plus one or more lexical forms (the different possible analyses of the SF) with their grammatical symbols.

- *Surface forms (ambiguous segmented stream)*. The word as it appears in the original text.

- *Lexical forms*. The lemma of the word and its grammatical symbols.

- *Grammatical symbols*. They describe the morphological and grammatical attributes of a surface form.

The symbols '^' for word beginning and '$' for word end are used to delimit *words*, as shown in this example:

^*word*$

```
[<html>
   <head>
     <title>]^Title/Title<n><m><sg>$^./.<sent>$[][</title>
   </head>
   <body>
     <p>]^Divided/Divide<vblex><pp>/Divided<vblex><past>$[
        ]^sentence/sentence<n><sg>/sentence<vblex><inf>$^./.
<sent>$[][</p>
  </body>
<html>]
```

Figure 2.4: Example of segmented stream with format encapsulated in non-XML format, corresponding to the HTML document in Figure 2.2.

To separate the *surface form* and the following *lexical forms*, the symbol / is used. This separator only has sense in the ambiguous segmented stream, since in the unambiguous stream there is only the lexical form. It is used as follows:

```
^surface form/lexical form 1/...$
```

Lexical forms can include symbols (generally located at the end), as shown in the example of Figure 2.4.

Chapter 3

Modules specification

3.1 Lexical processing modules

3.1.1 Module description

One of the most efficient approaches to lexical processing is based on the use of finite-state transducers (FST) [10, 15]. FST are a type of finite-state automata, which may be used as one-pass morphological analysers and generators and may be very efficiently implemented. In this project, we have used a class of FST called letter-transducers [15, 6, 4]; in fact, any finite-state transducer may always be turned into a letter-transducer. Garrido and collaborators [4, 6] give a formal definition of the letter transducers used in this project; describing them informally, a letter-transducer is an idealised machine consisting of:

1. A (finite) set of states, that is, of situations in which the transducer can be while it is reading, from left to right, the input letters or symbols. Among the states of the set, we can distinguish:

   (a) A single initial state: this is the state in which the transducer is before processing the first letter or the first symbol of the input.

   (b) One or more acceptance states, which are only reached after having completely read a valid entry and, therefore, are used to detect valid words.

2. A set (also finite) of state transitions consisting of:

   (a) the origin state

   (b) the destination state

(c) the input letter or symbol

(d) the output letter or symbol

To make possible that input and output have different lengths at any time, it is allowed that there is no input symbol, that there is no output symbol or that there is neither input nor output symbol. This case is generally represented using a special symbol (the empty symbol).

Every time the transducer reads an entry symbol, it creates a list of *live* or *active* states, each one of which has an associated output (a sequence of symbols). The way the letter transducer works is different for each type of lexical processing operation. For example, in the morphological analysis, the transducer tries to read the longest entry recognised by the dictionary ("left-to-right, longest-match" mode).

1. Beginning: the set of live states is given a single live state: the initial state, with the empty word ("") as output associated to the state.

2. When from one of the states in the current set of live states it is possible to reach other states through transitions that do not have input symbol, these states are added to the set of live states, and are associated to the output obtained when extending the associated outputs with the output symbol found in the corresponding transitions. This expansion operation of the set of live states continues until it is not possible to add more states.

3. A symbol from the input word is read.

4. A new set of live states is created, made with the states reached through transitions that have that symbol as input, and this states are associated to the outputs extended by adding the corresponding output symbols found in the transitions.

5. If the current set has any live state, the process continues on step 2.

6. The sets of live states are read backwards until a set is found which contains acceptance states. The morphological analyses will be the outputs associated to these states, and the reading position is set to the position immediately after this set (so that it can be processed again by the transducer in the next pass).

Not all acceptance states have the same characteristics, and this fact adds more conditions to the acceptance process, in order to be able to deal with unknown words or with words that are joined to other words, as will be explained later.

The transducer reads the input word only once on average, from right to left and symbol by symbol, and keeps a tentative list of possible partial outputs that is updated and pruned as the input is being read. When letter transducers are used as morphological analysers or as lemmatizers, they read a surface form and write the resulting lexical form(s). In this case, input symbols are the letters of the surface form, and output symbols are the letters needed to write the lemmas, as well as the letters and special symbols needed to represent the morphological analysis, such as in `<n>`, `<f>`, `<p2>`, etc.

The transducers work in a similar way for other lexical processing tasks.

3.1.1.1 Letter case handling in dictionaries

The same input word in a lexical processing module can be written differently regarding letter case. The most frequent cases are:

- The whole word is in lower case.
- The whole word is in upper case.
- The first letter is capitalised and the rest is in lower case (typical case for proper nouns).

The transductions in the dictionary can also be found in these three states. The way in which one word is written in the dictionary is used to discard possible analysis of the word, according to the following rules:

- If the input letter is upper case and in the current analysis state there are concordant transitions in lower case, these transductions are made.
- If the input letter is lower case and in the current state there are not concordant transitions in lower case, the transductions are not made.

Thanks to this policy, a surface form that is not capitalised can not be analysed as a proper noun.

The case of an input word will be maintained in the output of the translator unless it is decided not to do so. The case can be changed in the structural transfer module; this option is useful, for example, when there is a

reordering of words or when a word is added before a capitalised word at the beginning of a sentence, such as in the translation of the Catalan phrase *Vindran* into English: *They will come*.

3.1.2 Data format: the dictionaries

3.1.2.1 General criteria for dictionary design

The experience of the Transducens group at the Universitat d'Alacant in the creation of machine translation systems between Romance languages (`es`, `ca` and `pt`) already operative and publicly accessible has inspired the main characteristics of the whole shallow-transfer machine translation system described in this document, as well as its application to the Romance languages of Spain (`es`, `ca` and `gl`). In some sense, it could be stated that in the present project the only work was to adapt (rewrite in a standardised and interoperable format) the specifications and programs used in already operative projects.

In particular, the design of the dictionaries has been based in an architecture that pretends to separate, as far as possible, the source language from the target language, even knowing that these dictionaries are translation-oriented and, therefore, that it is not advisable to elaborate them completely separately. The chosen format is used for the specification of both morphological dictionaries (monolingual) and bilingual dictionaries.

The format for dictionaries, as well as for the rest of linguistic data (definition file for part-of-speech tagger and structural transfer rules) is XML[1], an international standard used in numerous natural language processing projects which, thanks to the availability of many utilities and libraries, it is becoming a very powerful tool for linguistic data representation and exchange (see article [8]).

Dictionaries are designed so that they can be compiled into *letter transducers* , for efficiency reasons. For more information on letter transducers as a particular case of finite-state transducers, see Section 3.1.1 or the article [6].

The letter transducers that are generated from the system dictionaries (morphological, bilingual and post-generation dictionaries) process input character strings to produce output strings. According to this, dictionaries are made of entries consisting of string pairs that correspond to the inputs and outputs of the transducer.

[1] `http://www.w3.org/XML/`

The most powerful tool in these dictionaries is the definition and use of *paradigms*. Since in Romance languages a lot of lemmas share the same inflection pattern (there are regularities in their inflection), it is useful and straightforward to group these regularities in inflection paradigms to avoid having to write all the forms of every word. Paradigms allow the representation of dictionary entries compactly and help optimise the speed for building a dictionary. Once the most frequent paradigms in a dictionary are defined, the linguist does not need to bother, in most cases, with the whole inflection of a new term, since entering an inflective word is generally limited to writing the lemma and choosing one inflection pattern among the previously defined paradigms. Furthermore, the use of paradigms reduces the memory requisites, facilitates the construction of efficient letter transducers and speeds up the compilation process [11]. We did not use paradigms in bilingual dictionaries (although it is possible to) because most of the inflection information is processed implicitly in these dictionaries, as explained in page 39.

3.1.2.2 Dictionary types

In our system there are three types of dictionaries: morphological (monolingual) dictionaries for each of the languages involved (Spanish, Catalan and Galician); bilingual dictionaries for the different translation pairs (Spanish–Catalan and Spanish–Galician), and post-generation dictionaries for each of the languages (a post-generation dictionary is not a typical dictionary, with lemmas and morphological information, but is like a little dictionary of the orthographic transformations that words may undergo when they come together). The structure of the three dictionary types is specified by the same DTD (*Document Type Definition*), which can be found in Appendix A.1.

Morphological dictionaries are used both for building morphological analysers —the translation system module used to obtain all the possible lexical forms for a certain surface form in the source language — and morphological generators —the module that generates the surface form in the target language from the lexical form of each word—. These two modules are obtained from a single morphological dictionary, depending on the direction in which it is read by the system: read from left to right, we obtain the analyser, and read from right to left, the generator.

The block structure typical for these dictionaries is the following:

- *An alphabet definition*. This definition is used exclusively for building the morphological analyser; specifically, it enables the morpho-

logical analyser to appropriately tokenize unknown words and the ones in the conditional sections (see the description of the element `<section>` in page 25); the morphological generator does not need this definition.

- *A definition of symbols*. It consists of a declaration of the grammatical symbols that will be used in dictionary entries (you can find in Appendix B a list with the grammatical symbols used in this project).

- *A definition of paradigms*. Paradigms need to be defined here in order to be used in the dictionary sections or in other paradigms.

- *One or more dictionary sections with conditional tokenization*, type `standard`. To include most of the words of the dictionary.

- *One or more dictionary sections with unconditional tokenization*. To include certain words that follow a regular pattern or that are tokenized regardless the text directly after them (see description of the element `<section>` in page 25). In the Catalan morphological dictionaries, words requiring an unconditional tokenization are distributed in two sections: one for the forms that require the introduction of a blank immediately after (due to processing requirements of the lexical forms), like the apostrophized forms *l'* or *d'*, and another one for punctuation marks, numbers and other signs.

Bilingual dictionaries represent in the system the lexical transfer process, that is, the assignment of the TL lexical form that corresponds to each SL lexical form. Two *products* are obtained from each bilingual dictionary, depending on the direction in which it is read by the system: when the dictionary is read from left to right, we obtain the lexical transfer module in one translation direction, and when it is read from right to left, in the other direction. For the bilingual dictionaries of our project, it has been established that Spanish will be put always on the left side of the entries, and the rest of the languages (Catalan and Galician), on the right side. Thus, for example, the bilingual Spanish–Galician dictionary will be read from left to right for the translation `es-gl` and from right to left for the translation `gl-es`. In applications like the ones in this project, these dictionaries do not have paradigms: they are build with generic entries which almost always have no more information than lemma and part of speech, and there is no inflection information.

The block structure used in the bilingual dictionaries of this project is the following:

- *A definition of symbols*. It consists of a declaration of the grammatical symbols that will be used in dictionary entries.

- *A single dictionary section*. Where bilingual correspondences are specified.

Since 2007, bilingual dictionaries allow the specification of more than one TL translation, so that a lexical selection module (see Section 3.4) can choose the most suitable equivalent according to the context. To that end, an attribute has been added to bilingual dictionaries. You can find its description in section 3.1.2.4.

Post-generation dictionaries are used to perform some transformations (orthographic changes, contractions, apostrophation, etc.) required after surface forms in the target language have been generated and come into contact with each other. Since this kind of operation can be expressed as a translation of character strings, it has been decided to use the same type of dictionaries. It is implicitly assumed that the parts of the text whose processing has not been specified are copied just as they arrive. In these dictionaries, the definition of paradigms is useful to express systematic changes in the word contact phenomena. Unlike the other dictionary types, these do not include grammatical symbols, since they process surface forms.

The block structure of post-generation dictionaries is the following:

- *A definition of paradigms*. To use in entries.

- *A dictionary section*. Where the patterns for post-generation operations are specified.

The following table contains an overview of the possible reading directions of dictionaries and their application to the Romance languages in this project:

Dictionary	Reading direction	Function
Morphological	left–right	analysis for `es`, `ca` and `gl`
	right–left	generation for `es`, `ca` and `gl`
Bilingual	left–right	translation for `es-ca` and `es-gl`
	right–left	translation for `ca-es` and `gl-es`
Post-generation	left–right	post-generation for `ca`, `es` and `gl`

```
<?xml version="1.0" encoding="utf-8"?>
<dictionary>
  <alphabet>abcdefghijk ... ABCDEFGH ... çñáéíóú</alphabet>
  <sdefs>
    <!-- ... -->
  </sdefs>
  <pardefs>
    <!-- ... -->
  </pardefs>
  <section ...>
    <!-- ... -->
  </section>
  <!-- ... -->
</dictionary>
```

Figure 3.1: Use of the elements <**dictionary**> and <**alphabet**>

3.1.2.3 Description of the dictionary format

This section presents the main elements of the format in which dictionaries are built. The formal definition (a DTD) can be found in Appendix A.1. Section 3.1.2.4 describes the characteristics of a bilingual dictionary that works in an Apertium system with lexical selection module. Finally, from pages 38 to 41 there is a description of the different particularities of entries for the three dictionary types (morphological, bilingual and post-generation).

Element for dictionary <dictionary>

This is the root element and includes the whole dictionary. It contains an alphabetic character definition, a definition of symbols (which are the morphological tags for the words), a definition of inflection paradigms and one or more dictionary sections, which contain the entries for the lexical forms (consisting of pairs made of surface form–lexical form). Figure 3.1 shows the basic block structure of a generic dictionary.

Element for alphabet <alphabet>

It is used to specify a definition of alphabetic characters. The purpose of this specification is enabling the modules that process the input by means of letter transducers to tokenize it in individual words.

```
<sdefs>
  <sdef n="n"/>
  <sdef n="det"/>
  <sdef n="sg"/>
  <sdef n="pl"/>
  <!-- ... -->
</sdefs>
```

Figure 3.2: Use of the element <**sdefs**>

In the present design, the definition of an alphabet only has sense in morphological dictionaries, since it is needed for the analysis. Figure 3.1 shows a use example for this element.

Element for symbol definition section <sdefs>

It groups all the symbol definitions in a dictionary (<**sdef**>). There is an example of its use in Figure 3.2.

Element for symbol definition <sdef>

It is an empty element (it does not delimit any content): it is used to specify, through the values of the attribute *n*, the names of the grammatical symbols that are used in the dictionary to morphologically label lexical forms. In Figure 3.2 you can find a use example for this element. Refer to Appendix B if you need a list with all the grammatical symbols used in the dictionaries of this project.

Element for dictionary section <section>

It contains the words that will be recognised by the dictionary. The reason to divide a dictionary in sections is that some forms —for example, the ones coming from the identification of certain regular patterns, or some forms that pertain to a specific dialect— may need a different processing.

One of the problems that the definition of sections in a dictionary helps to solve is the tokenization procedure during morphological analysis. Most of the forms are tokenized following a conditional criterion: identifying if the character being processed is followed by a non-alphabetic character —that is, not defined in <**alphabet**>—. However, there are other forms, like the Catalan apostrophized words *l'* or *d'*, that need an unconditional tokenization model: there is no need to analyse what comes after them,

```
<section id="principal" type="standard">
<!-- ... -->
</section>
<section id="patterns" type="inconditional">
<!-- ... -->
</section>
```

Figure 3.3: Use of the element <**section**>

since, if it is an alphabetic character, it will belong to the *next* word. The forms that require unconditional tokenization are included in a specific section of the dictionary. Other kinds of processing can also be solved through these divisions.

The value of the attribute *type* is used to express the kind of string tokenization applied in each dictionary section: the possible values of this attribute are: standard, for almost all the forms of the dictionary (conditional mode), preblank and postblank, for the forms that require an unconditional tokenization and the placing of a blank (before and after, respectively), and inconditional for the rest of forms that require unconditional tokenization.

The attribute *id* is used to assign an identifier (a name) to the dictionary sections.

Element for entries <e>

An entry is the basic unit of a dictionary or of a paradigm definition. Entries consist of a concatenation in any order of string pairs <**p**>, identity transductions <**i**>, references to paradigm <**par**> or regular expressions <**re**>. The structure and meaning of these elements is explained later in this section (in pages 27, 29, 32 and 34 respectively).

Two optional attributes are used with this entry. The first one is *r* (for *restriction*), which specifies if the entry has to be considered only when reading the dictionary from left to right (LR) or when reading it from right to left (RL). If nothing is specified, it is assumed that the entry must be considered in both directions.

In morphological dictionaries, the restriction LR causes that a LF is analysed but not generated (for example, when the LF belongs to a dialectal variant that we wish to recognise but not to generate) and the restriction RL causes that a word is generated but not analysed (needed, for example, for forms with post-generator activation mark, see page 35 for

more details).

In bilingual dictionaries, the restrictions `LR` and `RL` cause that the translation is done only in one direction: for example, in a bilingual `es–ca` dictionary, `LR` indicates that the LF is only translated from Spanish to Catalan, and `RL` only from Catalan to Spanish. Let's illustrate it with an example: the Spanish adverbs *aún* and *todavía* ("still") are translated into Catalan as the same word, *encara*. We can only translate the Catalan adverb *encara* as one of both words into Spanish (there is no difference in meaning); we decide to translate it as *todavía*. In this case, we have to write two entries in the bilingual dictionary: the entry that matches *aún* with *encara* needs to have the restriction `LR` (translation only from `es` to `ca`) and the one that matches *todavía* with *encara* does not need to have any restriction (translation in both directions).

Direction restrictions are also necessary in bilingual dictionaries when we have words with gender to be determined ("GD") or number to be determined ("ND") (consult page 39 for more information).

The other optional attribute in entries is the lemma name *lm*. Due to the employment of paradigms to represent the inflection regularities of lexical units, an entry in morphological dictionaries contains the part of the lemma that is common to all the inflected forms, that is, it contains the lemma cut at the point in which the paradigm regularity begins (for example, the Spanish adjectives *distinto*, *absoluto* and *marino* appear in entries as *distint*, *absolut* and *marin*, since the rest of the inflected forms is common to all of them and specified in a paradigm). This fact can make the dictionary difficult to understand. Therefore entries have this attribute, which contains the whole lemma of the lexical form, so that the dictionary becomes more understandable and linguists can solve problems quickly. In bilingual dictionaries, which normally do not have references to paradigms,[2] this attribute is not used.

Element for string pair <p>

This basic element of dictionaries is used in any kind of entry to indicate the correspondence between two strings; this correspondence specifies a lexical transformation that will be carried out by a state path in the resulting finite-state transducer [4].

It is defined by a pair of internal elements: The left element (<**l**>) and the right element (<**r**>). Its structure is shown in Figure 3.4.

[2]They could have references to paradigms, but we did not judge it necessary for the languages involved .

```
<p>
  <l><!-- ... --></l>
  <r><!-- ... --></r>
</p>
```

Figure 3.4: Use of the element <**p**>

A pair <**p**> must include these two parts although one can be empty, which means deleting (or inserting) a string. The elements <**l**> and <**r**> have the same internal structure and the same requisites. They can contain text and references to grammatical symbols (which, for the languages of the present project, inflected by suffixation, are usually placed at the end in any amount). Outside the tags <**l**> and <**r**> of a string pair there is nothing.

Element for reference to symbol <s>

References to symbols (or tags) are used to specify the morphological information of a LF and are used in any place inside a string pair, that is, inside the elements <**l**> and <**r**>, as if they were individual characters; for the languages of our project, however, they are put at the end of the pairs and always in the same order for the same word type. This order is decided by the linguist according to how he/she wishes to characterise morphologically the LF in the dictionaries, and must be the same in all the dictionaries of a system if we want that the lexical and structural transfer operations work correctly. So, for example, in the Romance language dictionaries of this project, a noun has in the first place the symbol for part of speech (*n*, noun), then for gender (*m*, masculine, *f*, feminine, *mf*, masculine–feminine), and finally for number (*sg*, singular, *pl*, plural, *sp*, singular–plural). The list in Appendix B contains all the grammatical symbols used in the dictionaries of this project and shows the order which has been established for each type of word.

In morphological dictionaries, references to symbols are used in paradigms as well as in entries which do not include any reference to a paradigm. In bilingual dictionaries, usually only the first symbol of each LF is specified, since the rest is automatically copied from the source language LF to the target language LF (in the case they are identical in both languages).

To specify which symbol we are referring to, we use the (mandatory) attribute *n*. The symbol must be defined in the symbol definition section (<**sdefs**>).

```
[1]

<e lm="perro">
  <p>
    <l>perr</l><r>perr</r>
  </p>
  <par n="abuel/o__n"/>
</e>

[2]

<e lm="perro">
  <i>perr</i>
  <par n="abuel/o__n"/>
</e>
```

Figure 3.5: Use of the element <**i**> entries [1] and [2] are equivalent

```
<pardefs>
  <pardef n="abuel/o__n">
    <!-- ... -->
  </pardef>
  <!-- ... -->
</pardefs>
```

Figure 3.6: Use of the element <**pardefs**>

Element for identity transduction <i>

It is a way to write a string pair in which left side and right side are identical. For example, the two entries shown in Figure 3.5 are completely equivalent. The advantage of writing entries with this element is that the result is more compact and more readable.

Element for paradigm definition section <pardefs>

This element includes all the paradigm definitions of a dictionary, each definition in an element <**pardef**>, as shown in Figure 3.6.

Element for paradigm definition <pardef>

It defines an inflection paradigm in the dictionary. A paradigm can be understood as a small dictionary of alternative transformations that can be concatenated to parts of words (or to entries of another paradigm) to specify regularities in the lexical processing of the dictionary entries, such as inflection regularities. To specify these regularities, each paradigm is a list of entries <**e**> like the ones in the dictionary, that is, it has the same structure as a dictionary section <**section**>; therefore, paradigm entries consist of a pair (<**p**>) with left side (<**l**>) and right side (<**r**>). These elements can contain text or grammatical symbols <**s**>.

As in symbol definitions, paradigm definitions have an attribute *n* which specifies the paradigm name, so that it can be referred to inside dictionary entries. In a dictionary entry, therefore, one only needs to indicate the corresponding paradigm name in order that all its possible forms get specified.

The example of paradigm definition pointed out in Figure 3.6 appears developed in Figure 3.7. The following table shows the information expressed by the paradigm:

Root (SF and LF)	Ending (SF)	Analysis (LF)
abuel	o	o<n><m><sg>
abuel	a	o<n><f><sg>
abuel	os	o<n><m><pl>
abuel	as	o<n><f><pl>

This paradigm is assigned to all Spanish nouns (n) that inflect like *abuelo*, such as *alumno*, *amigo* or *gato*, and is designed to be used as a *suffix* in dictionary entries. In general, paradigms can be applied to any position of a dictionary entry (if it makes sense, of course). We can think of paradigms as transducers that are inserted at the point where they are specified. Figure 3.8 shows an example of paradigm defined to be used as a prefix. It is the paradigm used to analyse and generate Spanish words beginning with *ex*, *ex-*, etc., like *ex-presidente*, *exministro*, *ex director*, etc., with all the orthographic variations (*ex* with hyphen, without hyphen and joined, without hyphen and with a blank <**b**/>, see page 3.1.2.3); the output lemma simply adds *ex* without hyphen nor blank to the accompanying lemma. The direction restrictions ("LR") that appear in the example are used to determine which form will the translator generate. The empty identity transduction (<**i**/>) is necessary in this case to analyse and generate the word without the prefix *ex*.

```
<pardef n="abuel/o__n">
  <e>
    <p>
      <l>o</l>
      <r>o<s n="n"/><s n="m"/><s n="sg"/></r>
    </p>
  </e>
  <e>
    <p>
      <l>a</l>
      <r>o<s n="n"/><s n="f"/><s n="sg"/></r>
    </p>
  </e>
  <e>
    <p>
      <l>os</l>
      <r>o<s n="n"/><s n="m"/><s n="pl"/></r>
    </p>
  </e>
  <e>
    <p>
      <l>as</l>
      <r>o<s n="n"/><s n="f"/><s n="pl"/></r>
    </p>
  </e>
</pardef>
```

Figure 3.7: Use of the element <**pardef**> to define the inflective morphology of Spanish nouns with four endings, such as *abuelo, -a, -os, -as* ("grandfather, grandmother")

```
<pardef n="ex">
  <e r="LR"><p><l>ex<b/></l><r>ex</r></p></e>
  <e><i>ex</i></e>
  <e r="LR"><p><l>ex-</l><r>ex</r></p></e>
  <e><i/></e>
</pardef>
```

Figure 3.8: Use of the element <**pardef**> in the paradigm for the prefix *ex*.

```
<e lm="perro">
  <i>perr</i>
  <par n="abuel/o__n"/>
</e>
```

Figure 3.9: Use of the element <**par**>

Entries in a paradigm can contain references to other paradigms provided that these have been defined upper in the file. On the other hand, for the moment a paradigm definition can not include itself neither directly nor indirectly.

Paradigms are used in morphological dictionaries for the analysis and generation of lexical forms. For the language pairs of this project, there is no need to define paradigms in bilingual dictionaries (see page 39).

From Apertium 2 on, there is a new type of paradigm, called meta-paradigm, that allows the definition of paradigms with variations according to the value of an attribute specified in each entry that refers to that paradigm. Section 3.1.2.7 describes the characteristics and use of meta-paradigms.

Element for reference to a paradigm <par>

It is used inside an entry to indicate which inflection paradigm, among the ones defined in <**pardefs**>, follows the entry. Thanks to the references to paradigms there is no need to write all the inflected forms of a lemma in a morphological dictionary entry. The attribute *n* is used to specify the name of the paradigm we want to refer to.

The result of inserting a reference to a paradigm in an entry is the creation of so many string pairs as cases specified in the paradigm. For example, the entry in Figure 3.9, with a reference to the paradigm "abuel/o__n" (defined in Figure 3.7), is equivalent to an entry where each string pair of the paradigm is concatenated to the lemma (that is, an entry with every inflected form of the lemma), as shown in Figure 3.10. In this figure, you can see that the paradigm delivers always in the right string (<**r**>) the lemma (*perro*) with the grammatical symbols that apply to the surface form, since it is from the lemma that transfer operations are carried out.

The appropriate use of paradigms, besides enabling the creation of compact dictionaries, improves compilation speed and reduces memory requirements during this process, since in compilation it is possible to create a single data structure for each one of most paradigms [11].

```
<e>
  <p>
    <l>perro</l>
    <r>perro<s n="n"/><s n="m"/><s n="sg"/></r>
  </p>
</e>
<e>
  <p>
    <l>perra</l>
    <r>perro<s n="n"/><s n="f"/><s n="sg"/></r>
  </p>
</e>
<e>
  <p>
    <l>perros</l>
    <r>perro<s n="n"/><s n="m"/><s n="pl"/></r>
  </p>
</e>
<e>
  <p>
    <l>perras</l>
    <r>perro<s n="n"/><s n="f"/><s n="pl"/></r>
  </p>
</e>
```

Figure 3.10: Entry equivalent to the one in Figure 3.9, that shows the result of inserting the reference to paradigm <**par**> with the paradigm defined in Figure 3.7.

```
<e>
  <re>[0-9]+([.,][0-9]+)?(%)?</re>
  <p><l/><r><s n="num"/></r></p>
</e>
```

Figure 3.11: Us of the element <**re**> in an entry for the detection of Arabic numbers.

Element for regular expression <re>

In natural languages too there are patterns that can be recognized as regular expressions: for example, punctuation marks, numbers (Latin or Roman), e-mail or web page addresses, or any kind of code identifiable through these mechanisms.

For this cases we use the string contained in the tag <**re**>. The compiler reads the regular expression definition and transforms it in a transducer that is inserted in the rest of the dictionary and that translates all the strings that match the expression into identical strings.

The syntax of the present implementation of these regular expressions processes a subgroup of Unix regular expressions, which includes the operators `*`, `?`, `|` and `+`, as well as groupings through parentheses and optional character ranks, for example `[a-zA-zñú]` or its negated versions, like `[^a-z]`.

By analogy, they can be seen as <**i**> elements, with the difference that they can identify strings which may be infinite (like numbers).

Figure 3.11 shows the way to tag quantities expressed as Arabic numbers in the dictionary.

Element for blank block <b>

It is used to express the presence of blanks between the words of a multiword (see page 43 for an explanation on multiwords). It can be inserted in the <**i**>, <**l**> and <**r**> elements. In Figure 3.12 you can see the entry for the Spanish multiword expression *hoy en día* (”nowadays”): the blanks between words are expressed as <**b**/> elements inside the left and right strings.

Blanks can consist of normal space characters or of document format information blocks encapsulated by the de-formatter (*superblanks*, see Section 3.6.1).

```
<e lm="hoy en día">
  <p>
    <l>hoy<b/>en<b/>día</l>
    <r>hoy<b/>en<b/>día<s n="adv"/></r>
  </p>
</e>
```

Figure 3.12: Use of the element <**b**>

```
<e r="RL" lm="de">
   <p>
      <l><a/>de</l>
      <r>de<s n="pr"/></r>
   </p>
</e>
```

Figure 3.13: Use of the element <**a**> in a morphological dictionary

Element for post-generator activation <a>

The element <**a**> for the activation of the post-generator is used to indicate that a word in target language may undergo orthographic transformations due to the contact with other words; for example, being apostrophized, contracted, written without intermediate spaces, etc. These transformations need be carried out after the generation of the target language surface forms, as until then words are isolated and it is not possible to know which words will get in contact . Therefore, these operations must be carried out by the module next to the generator, which is called post-generator. In order to signal which words are to be processed by the post-generator, this element is used in the surface form side of these entries in the morphological dictionary.

The example in Figure 3.13 shows its use, in a Catalan morphological dictionary, for the preposition *de*, which, when appearing before a singular or plural masculine definite article (*el, els*), forms a contraction (*del, dels*). The presence of the tag <**a**/> causes the activation of the post-generator, which checks whether the preposition is followed by one of the words that cause it to contract and, if it is so, makes the contraction (see page 41 for more details). The restriction `RL` indicates that this is an only-generation entry, since it does not make any sense for the analysis.

Element for group marking <g>

This element is used, inside the <**l**> and <**r**> elements, to define groups that require a special treatment beyond the normal word by word processing. It is used in inflective multiwords to signal the beginning and the end of the group of invariable lexical forms (one or more) that are adjacent to the inflected word and that, together with it, build an inseparable unit. In Section 3.1.2.6 you will find a detailed explanation of the different multiword types, and in Figure 3.22 of that section you can see an example of its use.

Element for joining of lexical forms <j>

This element is used only in the right side of an entry (<**r**>) to indicate that the words that form a multiword are treated as individual lexical forms and, therefore, have a grammatical symbol each. This way, this multiword will be processed as a unit by the analyser and by the tagger until it reaches the auxiliary module `pretransfer` (see section 3.3), which is responsible for separating the lexical forms it is made of so that they reach the transfer module as independent forms. If the linguist wants that these forms reach the generator as joined forms, building again a multiword, it is necessary to define a structural transfer rule that groups them in a multiword (see Section 3.5.4). If, on the contrary, these joined forms must be only for the analysis, the entry must have the restriction `LR`.

In Section 3.1.2.6 you will find a more detailed explanation of this element. An example of its use can be found in Figure 3.20 of the mentioned section.

3.1.2.4 Modification of bilingual dictionaries for the new lexical selection module

In 2007, a new module has been added to the Apertium system: the lexical selection module, which is described in section 3.4.

In order for them to work in a lexical selection system, bilingual dictionaries must be slightly modified so that they allow the specification of more than one translation in target language. The only change is the addition of two new attributes to the element <e>. Although these new attributes can be used in all the dictionaries of a system, they only make sense in a bilingual dictionary entry.

In Appendix A.1.1 there is the part of the DTD `dix.dtd` where the element `e` used for dictionary entries is defined. The new attributes are:

slr (*sense from left to right*) is used to specify the *translation mark* when there is more than one translation from left to right for the lemma specified in the left side of an entry. The attribute can receive any value; however, the recommended action is to assign as value the lemma contained in the right part `<r>` (the translation of the lemma).

srl (*sense from right to left*) is used to specify the *translation mark* when there is more than one translation from right to left for the lemma specified in the right side of an entry. As before, the attribute can receive any value, but the recommended action is to assign as value the lemma contained in the left part `<l>` (the translation of the lemma).

Furthermore, in both cases the value of the attribute can end in a white space and the letter "D" to indicate that this is the default translation, that is, the translation that will be chosen when there is not enough information to make a decision. It is compulsory that, for entries that have more than one equivalent in target language, one of the equivalents, and only one, is marked with the letter "D" for *default*.

The following example shows how the new attributes are used. We take as example a bilingual English-Catalan dictionary, with the following entries having more than one translation in the target language:

- *look*: can be translated into Catalan as *mirar* (default) or as *semblar* (according to the English senses *view/seem*),
- *floor*: can be translated into Catalan as *pis* (default) or as *terra* (according to the English senses *level of building/ground*),
- *pis*: can be translated into English as *flat* (default) or as *floor*.

This information is represented by means of the two attributes described:

```
<e srl="flat D">
   <p>
      <l>flat<s n="n"/></l>
      <r>pis<s n="n"/><s n="m"/></r>
    </p>
</e>

<e slr="pis D" srl="floor">
   <p>
      <l>floor<s n="n"/></l>
```

```
      <r>pis<s n="n"/><s n="m"/></r>
   </p>
</e>

<e slr="terra">
   <p>
      <l>floor<s n="n"/></l>
      <r>terra<s n="n"/><s n="m"/></r>
   </p>
</e>

<e slr="mirar D">
   <p>
      <l>look<s n="vblex"/></l>
      <r>mirar<s n="vblex"/></r>
   </p>
</e>

<e slr="semblar">
   <p>
      <l>look<s n="vblex"/></l>
      <r>semblar<s n="vblex"/></r>
   </p>
</e>
```

3.1.2.5 Particularities of the different dictionary types

Dictionary entries have different characteristics depending on the dictionary type. Although some of these characteristics have been presented in the previous sections, we are going to describe them here more exhaustively.

Morphological dictionaries

In these dictionaries, used to generate the system's morphological analysers and generators, it is necessary to mark with <**a**/> those surface forms which, once generated, may need certain orthographic transformations due to the contact with other words; these operations are carried out by the post-generator. As these marks are only generated, the entries containing them must be only for the generation, which means that need to have the restriction *r*="RL" (from right to left). Figure 3.13 shows an entry

```
<e>
  <p>
    <l>pan<s n="n"/></l>
    <r>pa<s n="n"/></r>
  </p>
</e>
```

Figure 3.14: Bilingual dictionary entry for the translation *pan* (es)–*pa* (ca)

containing this element.

Bilingual dictionaries

As explained before, we have not used paradigms in the bilingual dictionaries of our system; these dictionaries are built with generic entries in which, almost always, only part of speech is specified, and which do not have inflection information. For example, in the es-ca dictionary, the entry for the Spanish words *pan*, *panes* ("bread"), translated into Catalan as *pa*, *pans*, would be as shown in Figure 3.14.

As you can see in the figure, only the first grammatical symbol <**s** *n*="**...**"/> of each word is specified, since the unspecified symbols that come after the specified ones in the bilingual dictionary are copied from the source lexical form to the target lexical form. This entry, therefore, works both for *pan* (singular) and for *panes* (plural): the morphological analyser delivers the lemma (*pan*) followed by the grammatical symbols that apply to the analysed surface form (*n m sg* or *n m pl* as applicable), and the symbols that are not specified in the bilingual entry (*m sg* or *m pl*) are copied to the target language. This is valid for both translation directions. The idea is to specify the information indispensable to differentiate the entries, and the rest is *deduced* (copied). It is important to bear this in mind, because, when there are differences between the grammatical symbols of a lexical form from SL to TL, these differences must be specified in the bilingual dictionary. For example, when between source word and translated word there is a gender or number change, one has to specify the grammatical symbols in order (the order in which these symbols appear in the morphological dictionaries)[3] until the symbol that changes between SL and TL is reached.

For example, to translate the Spanish word *cama*, feminine noun, into

[3]To know which grammatical symbols have been used in the dictionaries and in which order, see Appendix B.

```
<e>
  <p>
    <l>cama<s n="n"/><s n="f"/></l>
    <r>llit<s n="n"/><s n="m"/></r>
  </p>
</e>
```

Figure 3.15: Bilingual dictionary entry for the translation *cama* (es)–*llit* (ca)

the Catalan word *llit*, masculine noun, the entry in the bilingual dictionary must be as shown in Figure 3.15. The gender must be specified (*f*, *m*) because, if not, the symbols for gender and number would be copied from the SL lexical form into de TL lexical form. Therefore, when translating from es to ca, we would obtain the lexical form *llit* with the symbols n f sg or n f pl. In both cases, the generator would receive as input a word that is impossible to generate, since the Catalan morphological dictionary does not contain any entry with lemma *llit* and feminine gender.

In this example, the number symbols are not specified; therefore, it works for the correspondence *cama–llit* (singular) as well as for *camas–llits* (plural). However, when there is a number change, the only way is to specify also the gender if the order used in all the dictionary for grammatical symbols is *gender, number*.

By means of a direction restriction r we can indicate which translations are to be done only in one direction and not in the other one (see the description of the restrictions LR and RL in page 26). This is necessary when the correspondence between two lexical forms is not symmetrical; in such case, in the bilingual dictionary two or more entries have to be created and a direction restriction must be applied, like in the example shown in Figure 3.16. In this example, when translating from Spanish to Catalan (LR), we must generate only plural forms, since the word *postres* ("dessert") in Catalan does not have singular form. But, on the other hand, we will translate into Spanish only in plural form (although in Spanish the word has singular and plural forms), since it is not possible to determine, from the Catalan word, whether the number should be singular or plural.

There is another problem due to grammatical divergences between two languages that is resolved with the help of two special symbols, GD (for *gender to be determined*) and ND (for *number to be determined*), symbols which have to be defined in the symbol section of the bilingual dictionary. This problem arises when the grammatical information of a SL lexical form is not enough to determine the gender (masculine or feminine) or

```
<e r="LR">
  <p>
    <l>postre<s n="n"/><s n="m"/><s n="sg"/></l>
    <r>postres<s n="n"/><s n="m"/><s n="pl"/></r>
  </p>
</e>

<e>
  <p>
    <l>postre<s n="n"/><s n="m"/><s n="pl"/></l>
    <r>postres<s n="n"/><s n="m"/><s n="pl"/></r>
  </p>
</e>
```

Figure 3.16: Entries in the Spanish-Catalan bilingual dictionary for the correspondence *postre–postres* ("dessert")

the number (singular or plural) of the TL lexical form. Let's put an example: the Spanish adjective *común* ("common") is masculine and feminine at the same time (and, therefore, masculine–feminine, `mf`), but in Catalan the adjective has different forms for the masculine, *comú/comuns*, and the feminine, *comuna/comunes*. In the bilingual dictionary, the entry should be as shown in Figure 3.17: in the `LR` direction (from Spanish to Catalan), the gender information is not `m`, `f` nor `mf` but `GD`; this *gender to be determined* will be determined next by the structural transfer module, by means of the application of the suitable transfer rules (usually, rules for the agreement between the lexical forms in a pattern; see Section 3.5 to obtain a detailed description of transfer rules). In an analogous way, a similar mechanism exists for singular–plural using the symbol `ND` (for example, in Spanish *análisis* ("analysis") is singular and plural, whereas in Catalan the singular form is *anàlisi* and the plural form *anàlisis*).

Post-generation dictionaries

In the morphological dictionary, the lexical forms which, once generated, may undergo contraction, apostrophation or other transformations, depending of which words are in contact with them in the output text, must have the post-generator activation mark (`<a/>`, see page 35) in the generation entry (`RL` direction). It is essential that the surface forms marked with the post-generator activation mark are identical in the morphological and the post-generation dictionaries of the same translator. In the post-

```
<e r="LR">
  <p>
    <l>común<s n="adj"/><s n="mf"/></l>
    <r>comú<s n="adj"/><s n="GD"/></r>
  </p>
</e>

<e r="RL">
  <p>
    <l>común<s n="adj"/><s n="mf"/></l>
    <r>comú<s n="adj"/><s n="m"/></r>
  </p>
</e>

<e r="RL">
  <p>
    <l>común<s n="adj"/><s n="mf"/></l>
    <r>comú<s n="adj"/><s n="f"/></r>
  </p>
</e>
```

Figure 3.17: Entries in the Spanish–Catalan bilingual dictionary for the correspondence *común–comú* ("common"), the first one for the translation from Spanish to Catalan and the two others for the translation from Catalan to Spanish

generation dictionary, all entries begin with this activation mark.

In Figure 3.18 there is an extract of the Spanish post-generator; the example shows how the contraction for *de* and *el* is done, to form the word *del*. The paradigm `puntuación` not defined in the example contains the non-alphabetic characters that can appear in a text. We can see in the example that the entry for the preposition *de* has the mark <**a**/>. The paradigm assigned to this entry, "`el`", is the one defined just above. According to this entry, when the system receives as input the left string of the entry (the part between <**l**>) concatenated to the left string of the paradigm (that is, when the input is "<a/>**de**<b/>**el**<b/>" or "<a/>**de** <b/>**el**[puntuación]"), the module delivers as output string (the part between <r> elements) the string **"del"** followed by the blanks represented with <b/> or by the symbols represented with [puntuación]. Note that, in the module output, all the marks <**a**/> have been removed.

3.1.2.6 Multiword lexical units

The designed dictionary format allows the creation of *multiword lexical units* —in short, *multiwords*— of different kinds, depending on the problem to be approached.

In this project we have considered three basic types of multiwords:

1. The most simple case are *multiwords without inflection*, which consist of only one lexical form: the lemma is made of two or more invariable orthographic words but it is tagged as a unit. Figure 3.19 shows an example of invariable multiword (the Spanish expression *hoy en día*, "nowadays"): It is made of three words separated by a blank (<**b**/>) and, although it actually consists of an adverb, a preposition and a noun, it is tagged as an adverb as a whole, since it acts as one.

2. A more complicated issue is the case of *compound multiwords*, made of more than one lexical form, each one with its grammatical symbols. The words they are made of are considered not to build a semantic unit like in the previous case, but to appear together building a unit due to contact reasons (phonetic or orthographic reasons). In this category we include *contractions* and *enclitic pronouns* accompanying verbs. To mark this phenomenon we use the tag <**j**> described in page 36. You can see an example in Figure 3.20, in which the analysis of *del* delivers a lexical multiform made of two lexical forms: *de*, preposition, and *el*, singular masculine definite determiner, linked with the <**j**/> element. The analyser and the part-of-speech tagger handle this multiwords as a unit; however, before en-

```
<dictionary>
<pardefs>
  ...
  <pardef n="el">
    <e>
      <p>
        <l>el<b/></l>
        <r>l<b/></r>
      </p>
    </e>
    <e>
      <p>
        <l>el</l>
        <r>l</r>
      </p>
      <par n="puntuación"/>
    </e>
  </pardef>
  ...
</pardefs>
<section id="main" type="standard">
  ...
  <e>
    <p>
      <l><a/>de<b/></l>
      <r>de</r>
    </p>
    <par n="el"/>
  </e>
  ...
</section/>
</ditionary>
```

Figure 3.18: Post-generation dictionary data to perform the contraction for Spanish *de* + *el* = *del* .

```
<e lm="hoy en día">
  <p>
    <l>hoy<b/>en<b/>día</l>
    <r>hoy<b/>en<b/>día<s n="adv"/></r>
  </p>
</e>
```

Figure 3.19: Example of multiword without inflection in the morphological dictionary

```
<e lm="del" r="LR">
  <p>
    <l>del</l>
    <r>de<s n="pr"/><j/>
     el<s n="det"/><s n="def"/>
     <s n="m"/><s n="sg"/></r>
  </p>
</e>
```

Figure 3.20: Entry in the morphological dictionary for the analysis of a contraction (the Spanish contraction *del*)

tering the transfer module, they are processed by an auxiliary module called `pretransfer` (see section 3.3) which is responsible for separating the lexical forms they are made of. This way, they reach the transfer module as independent forms; the linguist has to decide whether they have to be joined again (which must be done in the structural transfer module) or they have to remain as independent forms through the next modules.

In our system, the elements forming a contraction continue as independent forms, and the post-generator is responsible for making the contractions in the target language if it is necessary. On the other hand, enclitic pronouns are joined again to the verb by means of a structural transfer rule (see Section 3.5), so the verb plus its enclitic pronouns get into the generation module as a single lexical multiform, its components joined with a `<j/>`. Therefore, entries containing enclitic pronouns must not have any direction restriction, as can be seen in the example in Figure 3.21, which shows a part of the paradigm for the Spanish verb "dar" ("to give"), specifically the entry for the infinitive form joined to an enclitic pronoun.

```
<e>
  <p>
    <l>ar</l>
    <r>ar<s n="vblex"/><s n="inf"/><j/></r>
  </p>
  <par n="S__cantar"/>
</e>
```

Figure 3.21: A fragment of the inflection paradigm for the Spanish verb *dar* ("to give"), which shows the entry for the infinitive form followed by an enclitic pronoun. Enclitic pronouns are contained in the paradigm `S__cantar`. Note that, unlike in Figure 3.20, this entry is both for analysis and generation.

3. The most complicated case in our system is the case of *multiwords with inner inflection* inside the lemma (or "split lemma" forms), like the example shown in Figure 3.22. The lemma of this kind of multiwords has one part with inflection (the *lemma head*) followed by one invariable part (the *lemma tail*). The invariable part has to be put between <**g**> elements, so that it can be moved to the position immediately after the lemma head to obtain the whole lemma of the multiword. For example, the lemma of the Spanish multiwords *echó de menos* ("he/she missed"), *echándole de menos* ("missing him/her"), etc. has to be *echar de menos* ("to miss"), since this form will be the one searched in the bilingual dictionary to find its translation. This means that the invariable lemma tail (*de menos*) has to be moved after the uninflected lemma head (*echar*). This moving backwards will be done by the auxiliary module `pretransfer` (see section 3.3) which runs before the structural transfer module.

   To understand the example in Figure 3.22, you have to be aware that the paradigm defining the verb *echar* includes, besides the verb inflection, the enclitic pronouns that can appear at the end of the inflected forms of the verb; in the output lexical multiform, this enclitic pronouns are joined using the empty element <**j**/>.

   When the translation is also a *split lemma* (for example, the translation of "to miss" in Catalan is *trobar a faltar*, with forms like *trobem a faltar*, *trobar-lo a faltar*, etc.), it is necessary to place again the lemma tail in its original place, after the inflected form plus the enclitic pronouns (if any), and indicate the correspondence of these invariable parts of the lemma (*de menos*, *a faltar*) at both sides of the transla-

```
<e lm="echar de menos">
  <i>ech</i>
  <par n="aspir/ar__vblex"/> <!-it includes enclitic pronouns -->
  <p>
    <l><b/>de<b/>menos</l>
    <r><g><b/>de<b/>menos</g></r>
  </p>
</e>
```

Figure 3.22: A morphological dictionary entry containing a <**g**> group.

```
<e>
  <p>
    <l>echar<g><b/>de<b/>menos</g><s n="vblex"/></l>
    <r>trobar<g><b/>a<b/>faltar</g><s n="vblex"/></r>
  </p>
</e>
```

Figure 3.23: A bilingual dictionary entry containing two corresponding <**g**> groups.

tion. So, in the example of Figure 3.22, the <**g**> element is used to mark the group '<b/>de<b/>menos' in the morphological dictionary, whereas in the bilingual dictionary (see Figure 3.23), the <**g**> element is used to establish the correspondence between the groups "<b/>de<b/>menos" and "<b/>a<b/>faltar".

If the translation is not a *split lemma*, you do not need to insert any <**g**> element in the target language string.

3.1.2.7 Metaparadigms

When developing the dictionaries for the Occitan translator, we were faced with a new need: we wanted to be able to specify paradigms for verbs that had a same inflection pattern but whose root changed in the different inflected forms. With the existing paradigm system, a new paradigm had to be created for each of these verbs, since it was only possible to specify an inflection regularity pattern for a group of verbs with invariable root. With metaparadigms, it is possible to specify the inflection regularity as well as verb root variations.

At the same time, metaparadigms allow the specification, in a single paradigm, of variations in the grammatical symbols of a lemma. That is, several lemmas can refer to a same metaparadigm even if they have different grammatical symbols. Whereas for Occitan, metaparadigms have allowed having a same paradigm for entries with root variations, for English, these have allowed having a same paradigm for entries with variations in their grammatical symbols.

Related with this, we created the concept of metadictionary: it is a dictionary which contains metaparadigms as well as the normal paradigms used so far. The name of a metadictionary is `apertium-PAIR.`L_1`.metadix` (for example, for the English monolingual dictionary in the Apertium-en-ca system, `apertium-en-ca.en.metadix`). When linguistic data are compiled these dictionaries are pre-processed, so that they have the appropriate format for the dictionary compiler.

Specification of metaparadigms

Metaparadigms are defined in the <**pardefs**> section of the monolingual dictionary, the same section where also the rest of the dictionary paradigms are defined. A metaparadigm, just like a paradigm, has a name specified in the attribute `n`. This name will have the same characteristics as in the other paradigms, with the difference that the variable part of the lemma root will be in brackets and in capital letters, as you can see in this example:

```
<pardef n="m/é[T]er__vblex">
```

This is the definition of a verb paradigm, where the inflection endings have a variable part in the root. The inflection paradigms specified inside this metaparadigm have to present inflection only in the part at the right of the brackets, for example like the one specified in the paradigm:

```
<par n="mét/er__vblex"/>
```

In conclusion, a complete example of metaparadigm definition would be:

```
<pardef n="m/é[T]er__vblex">
  <e>
    <p>
      <l>e</l>
      <r>é</r>
```

```
    </p>
    <i><prm/></i>
    <par n="sent/eria__vblex"/>
  </e>
  <e>
    <i>é<prm/></i>
    <par n="mét/er__vblex"/>
  </e>
</pardef>
```

The tag <**prm**/> is the marker that is used to place the variable text part (the root variation) in the paradigm definition.

Once a metaparadigm is defined, we may want that a verb uses it. To do so, in the verb entry (inside a <**e**> element) we must indicate the suitable metaparadigm and, through the attribute `prm`, define with which letters we want to replace the variable part specified in brackets. For example:

```
<e lm="acuélher">
  <i>acu</i>
  <par n="m/é[T]er__vblex" prm="lh"/>
</e>
```

This entry defines the Occitan verb *acuélher* (”to receive”) and specifies that its inflection paradigm is the one defined by the metaparadigm `m/é[T]er__vblex`, but replacing `T` with `lh`; that is, the letters following *acu* will be *élher* instead of *éter*.

As mentioned before, metaparadigms can also be used for entries which have some variation in their grammatical symbols. The way to specify them is basically the same: the variable part must be specified in the entry with the attribute `sa`, whereas in the paradigm the tag <**sa**> has to be placed where the optional grammatical symbol should appear.

For example, we have the following metaparadigm:

```
<pardef n="house__n">
  <e>
    <p>
      <l/>
      <r><s n="n"/><sa/><s n="sg"/></r>
```

```
    </p>
  </e>
  <e>
    <p>
      <l>s</l>
      <r><s n="n"/><sa/><s n="pl"/></r>
    </p>
  </e>
</pardef>
```

and the following entry:

```
<e lm="time">
  <i>time</i>
  <par n="house__n" sa="unc"/>
</e>
```

where *unc* means that the noun is uncountable.

In the metaparadigm, the tag <**sa**> shows the place where the grammatical symbol is to be placed if an entry contains the attribute `sa` with a value, as happens in the entry for *time*.

A dictionary which contains entries like the ones described here is called metadictionary and must be pre-processed in order to generate a dictionary that follows the DTD for Apertium 2, since the engine does not allow the direct use of metaparadigms. The next section describes how is this pre-processing like.

Pre-processing of the metadictionary

A metadictionary is an XML file to which two XSLT style sheets are applied, in order to pre-process the metaparadigms and obtain a dictionary with all the paradigms derived from the metaparadigms. The first style sheet, `buscaPar.xsl`, produces the list of verbs that use metaparadigms and deletes the possible repetitions of metaparadigms to be expanded. This style sheet generates, in combination with the sheet `principal.xsl`, a second style sheet called `gen.xsl`, which processes the metadictionary with the list of metaparadigms to be expanded and generates a dictionary in Apertium 2 format. Basically, what this generated style sheet does is:

1. In verb entries, if a verb uses a metaparadigm, this metaparadigm is replaced by the corresponding expanded and deparametrized paradigm. Thus, the previous example entry:

```
<e lm="acuélher">
  <i>acu</i>
  <par n="m/é[T]er__vblex" prm="lh"/>
</e>
```

would be deparametrized and expanded into:

```
<e lm="acuélher">
  <i>acu</i>
  <par n="m/élher__vblex"/>
</e>
```

2. On the other hand, since from the first pass the system knows which paradigms have to be created from metaparadigms, these are created. In the previous example, from the metaparadigm:

```
<pardef n="m/é[T]er__vblex">
  <e>
    <p>
      <l>e</l>
      <r>é</r>
    </p>
    <i><prm/></i>
    <par n="sent/eria__vblex"/>
  </e>
  <e>
    <i>é<prm/></i>
    <par n="mét/er__vblex"/>
  </e>
</pardef>
```

the system would generate the paradigm `"m/élher__vblex"` :

```
<pardef n="m/élher__vblex">
  <e>
    <p>
      <l>e</l>
      <r>é</r>
    </p>
    <i>lh/></i>
```

```
        <par n="sent/eria__vblex"/>
      </e>
      <e>
        <i>élh</i>
        <par n="mét/er__vblex"/>
      </e>
    </pardef>
```

After the metadictionary has been processed according to these steps, a .dix dictionary is generated which follows the DTD for Apertium 2 and which can already be compiled.

In the case of our second example, where the variable part was the sequence of grammatical symbols in the paradigm, the style sheets would be applied and, from the value *unc* specified in the attribute `sa`, the following paradigm would be generated:

```
<pardef n="house__n__unc">
  <e>
    <p>
      <l/>
      <r><s n="n"/><s n="unc"/><s n="sg"/></r>
    </p>
 </e>
 <e>
   <p>
     <l>s</l>
     <r><s n="n"/><s n="unc"/><s n="pl"/></r>
   </p>
 </e>
</pardef>
```

for nouns the morphological analysis of which should be (in data stream format):

```
time<n><unc><sg>
```

In this case, metaparadigms allows the use of the same paradigm for entries with the same inflection but with a slightly different morphological analysis.

It is important to note that, when a dictionary uses metaparadigms and, accordingly, its name has the extension `.metadix`, this will be the

```
<?xml version="1.0"?>
<analysis-chars>
  <char value="’">
    <equiv-char value="&#x2019;"/>
    <equiv-char value="&#x2BC;"/>
  </char>
  <char value="&#183;">
    <equiv-char value="."/>
  </char>
</analysis-chars>
```

Figure 3.24: Analysis character specification file

file where dictionary changes have to be made (adding, changing or deleting entries or paradigms), since the file `.dix` is automatically generated from this one every time linguistic data are compiled and, therefore, any changes made in the latter will be overwritten during compilation.

3.1.2.8 Analysis characters

Version 3 of the Apertium platform includes Unicode support; however, this lead to a new problem: alternate characters. Unicode supports several character sets, which include several characters that look identical or almost identical, but which have a different numeric value.

As a solution, equivalent characters can be specified in a file that complements the morphological dictionary. As the morphological dictionary is compiled, whenever a character mentioned in the analysis character specification is encountered, its equivalents are included as though they had been specified using entries specified with the `LR` restriction within the dictionary.

A sample analysis characters specification file can be seen in Figure 3.24. It's worth noting that the analysis characters file can only be used when there is a 1:1 mapping between individual characters; in the case of multiple characters, it would be better to use the example given earlier, in Figure 3.8

3.1.3 Automatic generation of the lexical processing modules

The four lexical processing modules (morphological analyser, lexical transfer, morphological generator and post-generator) are compiled from dictionaries by means of a single compiler based on letter transducers [14]. This compiler is much faster than the ones used in the systems interNOSTRUM [2, 5, 4] and Traductor Universia [7, 17], thanks to the use of new compiler building strategies and the minimization of partial transducers during the building process [11].

The division of dictionary entries into lemma and paradigm enables the effective construction of minimal letter transducers. The compiler makes the most of the factorization allowed by paradigms in order to speed up the construction. Taking into account that, in most European languages, word variations occur at the end or the beginning of words, we took advantage of this fact to improve the construction speed of the minimal transducer.

Paradigms are also minimized before being inserted in the big transducer in order to reduce the size of the big transducer before its minimization. Since, before minimizing, the paradigms of the dictionaries for the languages we have dealt with usually have just a few hundreds of states, the minimization of these paradigms is a very fast process.

If we assume that an entry can have at any point a reference to a paradigm, we could decide to copy at this point the transducer calculated in the paradigm definition. The method used in *Apertium* is based on the idea that it is not always necessary to copy, because in certain cases it is possible to reuse a paradigm that was already copied. In particular, two or more entries that share a paradigm as a suffix can reuse the same copy of this paradigm; the same can be said when it is as a prefix. However, generally it is not possible to reuse paradigms if they are located in intermediate positions of different entries, since new suffixes (or prefixes) can be added to existing entries, which causes the information inserted in the transducer not to be consistent with the dictionary, and therefore the generated transducer would be incorrect (it would add string pairs that are not present in the formal language defined by dictionaries).

Minimal letter transducers are built as explained next. From a string transduction it is possible to build a *sequence of letter transductions* $S(s:t)$ with length $N = \max(|s|, |t|)$ which is defined as follows for each element $1 \leq i \leq N$:

Figure 3.25: Building of the dictionary as prefix acceptor and link to paradigms through transitions $(\theta : \theta)$.

Figure 3.26: Minimized paradigm "-es **n m**" used in Figure 3.25.

$$S_i(s:t) = \begin{cases} (s_i : \theta) & \text{if } i \leq |s| \wedge i > |t| \\ (\theta : t_i) & \text{if } i \leq |t| \wedge i > |s| \\ (s_i : t_i) & \text{in other cases} \end{cases} \tag{3.1}$$

It should be emphasized that the construction design forbids the existence of a $(s : t)$ that is equal to $(\epsilon : \epsilon)$, which is crucial for the consistence of the building method.

The building method uses two procedures: the *assembly* procedure inferred from equation 3.1, and the minimization procedure, which is executed by a conventional minimization algorithm [16] for deterministic finite state automata, which consists of inverting, determining, inverting again and determining again, taking as the alphabet of the automaton to be minimized the Cartesian product of L and as empty transition the $(\theta : \theta)$.

Figure 3.25 shows a simplified example of the assembly process. Transductions, composed as in the equation 3.1, are inserted one by one in a transducer in the form of a *prefix acceptor* or *trie*, that is, in a way that there is only one node for each common prefix of the group of transductions

Figure 3.27: Minimized paradigm "z/-ces **n m**" used in Figure 3.25.

that form the dictionary. With the suffixes of the transductions (that are not shared) new states are created. In the point where there is a reference to a paradigm, a replica of this paradigm is created and a link is created to the dictionary entry which is being inserted in the transducer by means of a null transduction $(\theta : \theta)$.

Each paradigm, as it can be seen as a little dictionary, has been built according to this same procedure and been minimized to reduce the size of the content when building the big dictionary. In Figures 3.26 and 3.27 you can see the state of the paradigms used in Figure 3.25 after its minimization.

3.2 Part-of-speech tagger

3.2.1 Module description

The part-of-speech tagger is based on first-order hidden Markov models [13], that is, on statistical data. The states of the Markov model represent parts of speech, and the observable parameters are ambiguity classes [3], formed by groups of parts of speech.

In spite of working with statistical information, the training and behaviour of the tagger improve with the application of restrictions that forbid certain sequences of parts of speech (in the first-order models, these sequences can only include two parts of speech). For example, in Spanish or Catalan a preposition can never be followed by a verb in personal form; this restriction is of great help when the word after a preposition is ambiguous and one of its possible analyses is a verb in personal form (e.g., *de trabajo*, *en libertad*, etc.). Restrictions are explicitly declared in the tagger definition file, sometimes in the form of *prohibitions* and sometimes

of *obligations*.

The morphological tags which the tagger works with are not the same as the ones used in the morphological analyser. Usually, the information delivered by the analyser is too detailed for the purposes of the part-of-speech disambiguation (for example, for most purposes, it suffices to group in the same category all common nouns, regardless of their gender and number). The use of finer-grained tags does not improve the results, whereas it increases the number of parameters to be estimated and intensifies the problem of lack of linguistic resources such as manually disambiguated texts. For this reason, in the tagger file one has to specify how to group the *fine-grained* tags delivered by the morphological analyser into more general *coarse* tags —which we will call *categories*— that will be used in the part-of-speech disambiguation. Apart from coarse categories, one can also define lexicalized tags. Basically there are two types of lexicalizations described in bibliography: one type adds new observables and the other one, in addition, adds new states to the Markov model [12]; the tagger in Apertium uses the latter lexicalization type.

It is important to note that, in spite of working with *coarse* categories, the tagger outputs fine-grained tags like the ones from the morphological analyser. Sometimes it may occur that the morphological analyser delivers, for a certain word, two or more fine-grained tags that can be grouped under the same tagger category: e.g. in Spanish *cante* can be the 1st or the 3rd singular person of the subjunctive present of the verb *cantar* ("to sing"); both fine-grained tags, `<vblex><prs><p1><sg>` and `<vblex><prs><p3><sg>`, are grouped under the tagger category `VLEXSUBJ` (*subjunctive verb*). In this case, one of both fine tags is discarded; in the tagger definition file it is possible to define which fine-grained tag, among the ones that compose a coarse tag, will be delivered after disambiguation.

3.2.2 Data for the part-of-speech tagger

3.2.2.1 Introduction

We describe next the format of the files that specify how to group the *fine-grained* tags delivered by the morphological analyser into more general *coarse* tags. In this files, moreover, one can specify *restrictions* that help in the estimation of the statistical model underlying the process of lexical disambiguation, as well as preference rules to be applied when two fine-grained tags belong to the same category.

The tagger assumes that, in the input stream, lexical forms will be appropriately delimited, as described in the format specification for the data

stream between modules (Section 2). In brief, the format of the data delivered by the morphological analyser is the following:

$$\begin{array}{rcl} \text{analysedform} & \rightarrow & \text{lexicalmultiform } [\text{ lexicalmultiform }]^* \\ \text{lexicalmultiform} & \rightarrow & \text{lexicalform } [\text{ lexicalform }]^* \text{ lemma-queue?} \\ \text{lexicalform} & \rightarrow & \text{lemma finetag} \\ \text{lemma-queue} & \rightarrow & \text{lemma} \\ \text{finetag} & \rightarrow & \text{morphsymbol } [\text{ morphsymbol }]^* \end{array} \quad (3.2)$$

where:

- *analysedform* is all the information delivered for each surface form in the output of the morphological analyser
- *lexicalmultiform* is a sequence of one or more lexical forms followed, optionally, by an invariable queue as happens in some multiwords (like the Spanish expression *cántale las cuarenta*).
- *lexicalforms*[4] are units made of one lemma and one or more grammatical symbols (which compose the fine-grained tag) with the output information of the analyser
- *lemma-queue* is made of one or more lemmas [5] that are the invariable part of a multiword. The queue of a multiword is made of the lemma or lemmas with no inflection that follow the lemmas with inflection. For example, the Spanish multiword *cantar las cuarenta* ("to lecture", "to reproach") can take the forms *cántale las cuarenta*, *(le) cantaré las cuarenta*, *cantándole las cuarenta*, etc. In this case, the queue would be *las cuarenta* (see page 43 for more information).
- *finetag* is made of one or more grammatical symbols (*símbolo*gram).

For example, the entry for the Spanish ambiguous surface form *correos* would have two lexical multiforms; the first lexical multiform would have one single lexical form, with lemma *correo* ("post office") and a fine tag made of the grammatical symbols *common noun*, *masculine*, *plural*; the second lexical multiform would be a sequence of two lexical forms, one with lemma *correr* ("to move") and a fine tag made of the grammatical symbols *lexical verb*, *imperative*, *second person*, *plural*, and the other one with lemma *vosotros* ("you") and fine tag made of the grammatical symbols *pronoun*, *enclitic*, *second person*, *masculine-feminine*, *plural*.

[4] Separated from each other by a delimiter which corresponds to the `<j/>` element (see page 36).

[5] Separated from each other by the `<b/>` element (see page 34).

3.2.2.2 Format specification

The format of the file (encoded in XML) is specified by the DTD that can be found in Appendix A.2.

The meaning of the different tags is the following:

`tagger` : is the root element; its mandatory attribute `name` is used to specify the name of the tagger generated from the file.

`tagset` : defines the *coarse* tagset or categories with which the tagger works. Categories are defined by the fine-grained tags output by the morphological analyser.

`def-label` : defines a category or coarse tag (whose name is specified in the mandatory attribute `name`) by means of a list of fine tags defined with one or more `tags-item` elements; an optional attribute `closed` indicates whether this is a closed category; if this is the case, it is assumed that an unknown word can never belong to this category.[6]

The more specific categories *must* be defined before the more general ones. When the definition of a general category implicitly includes that of a specific category defined before, it is understood that it refers to all cases *except* the ones defined by the more specific category.

`tags-item` : is used to define a fine-grained tag by means of a sequence of grammatical symbols. The sequence of grammatical symbols that make up the fine tag is specified in the mandatory attribute `tags`. In this sequence, symbols are separated by a dot, and the asterisk "`*`" is used to express that any sequence of symbols may appear in its place. It is also possible to define lexicalized categories, specifying the lemma of the word in the attribute `lemma`.

`def-mult` : defines special categories (*multicategories*) made of more than one category, in order to deal with entries with more than one lexical form, like in the example given in the previous section. Each category is defined as a set of valid sequences (`sequence`) of previously defined categories or of fine-grained tags. It is designed for contractions, verbs with enclitic pronouns, etc.

[6] Closed categories are those that do not grow when new words are created: prepositions, determiners, conjunctions, etc.

sequence : defines a sequence of elements, which can be categories (`label-item`) or fine-grained tags (`tags-item`). Using fine-grained tags directly is useful if one wishes to use a sequence of grammatical symbols that is not part of any previously defined fine tag or that represents a greater specialization of a defined fine tag .

label-item : is used to refer to a category or coarse tag previously defined, to be specified in the mandatory attribute `label`.

forbid : this (optional) section is aimed to define restrictions as sequences of categories `label-sequence` that can not occur in the language involved. In the current version, due to the fact that the tagger is based on first-order hidden Markov models, sequences can only be made of *two* `label-items`.

label-sequence : defines a sequence of categories (`label-item`).

enforce-rules : this (optional) section allows defining restrictions in the form of obligations.

enforce-after : defines a restriction that forces that a certain category can only be followed by the categories belonging to the set of categories defined in `label-set`. Note that this kind of restrictions is equivalent to defining several forbidden (`forbid`) sequences (`label-sequence`) with the category defined in the mandatory attribute `label` and the rest of categories that do not belong to the set defined in `label-set`. For this reason, this kind of restriction must be used very cautiously.

label-set : defines a set of categories (`label-items`).

preferences : used to define priorities in terms of which fine-grained tag must be delivered in the tagger output when two or more fine tags are assigned to the same category.

prefer : specifies that, in case of conflict between different fine-grained tags assigned to the same category, the tagger must output the tag specified in the mandatory attribute `tags`. If a category contains more than one of the fine tags included in these `prefer` elements, the tag defined in the first place will be the selected one.

Figures 3.28 and 3.29 contain an example with the most significant parts of a tagger specification file defined by the DTD just described.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE tagger SYSTEM "tagger.dtd">
<tagger name="es-ca">
<tagset>
   <def-label name="adv">
      <tags-item tags="adv"/>
   </def-label>
   <def-label name="detnt" closed="true">
      <tags-item tags="detnt"/>
   </def-label>
   <def-label name="detm" closed="true">
      <tags-item tags="det.*.m"/>
   </def-label>
   <def-label name="vlexpfci">
      <tags-item tags="vblex.pri"/>
      <tags-item tags="vblex.fti"/>
      <tags-item tags="vblex.cni"/>
   </def-label>
   <def-mult name="infserprnenc" closed="true">
      <sequence>
         <label-item label="vserinf"/>
         <label-item label="prnenc"/>
      </sequence>
      <sequence>
         <label-item label="vserinf"/>
         <label-item label="prnenc"/>
         <label-item label="prnenc"/>
      </sequence>
   </def-mult>
   <def-mult name="prepdet" closed="true">
      <sequence>
         <label-item label="prep"/>
         <tags-item tags="det.def.m.sg"/>
      </sequence>
   </def-mult>
</tagset>
<!-- ... -->
```

Figure 3.28: Example of a tagger definition file (continues in Figure 3.29).

```
<!-- ... -->
<forbid>
   <label-sequence>
      <label-item label=="prep"/>
      <label-item label=="vlexpfci"/>
   </label-sequence>
   <!-- ... -->
</forbid>
<enforce-rules>
   <enforce-after label=="prnpro">
      <label-set>
         <label-item label=="prnpro"/>
         <label-item label=="vlexpfci"/>
         <!-- ... -->
      </label-set>
   </enforce-after>
   <!-- ... -->
</enforce-rules>
<preferences>
   <prefer tags="vblex.pii.p3.sg"/>
   <prefer tags="vbser.pii.p3.sg"/>
   <!-- ... -->
</preferences>
</tagger>
```

Figure 3.29: Example of a tagger definition file (comes from Figure 3.28).

3.2.3 Some questions about the training of the part-of-speech tagger

The training of the part-of-speech tagger can be made both in a supervised manner, using manually disambiguated texts, and a unsupervised manner, using ambiguous texts.

When the training is made with ambiguous texts (unsupervised), the format of the required text can be automatically obtained from a plain text corpus in the chosen language using the system's morphological analyser; in this case, the format of the text forms will be like the one defined in the figure 3.3 (its description can be found in page 58). As the chart shows, each analysed surface form can have more than one analysis (an *analysedform* can give as a result more than one *lexicalmultiform*).

$$\begin{aligned}
\text{analysedform} &\rightarrow \text{lexicalmultiform } [\text{ lexicalmultiform }]^* \\
\text{lexicalmultiform} &\rightarrow \text{lexicalform } [\text{ lexicalform }]^* \text{ lemma-queue?} \\
\text{lexicalform} &\rightarrow \text{lemma finetag} \\
\text{lemma-queue} &\rightarrow \text{lemma} \\
\text{finetag} &\rightarrow \text{morphsymbol } [\text{ morphsymbol }]^*
\end{aligned} \tag{3.3}$$

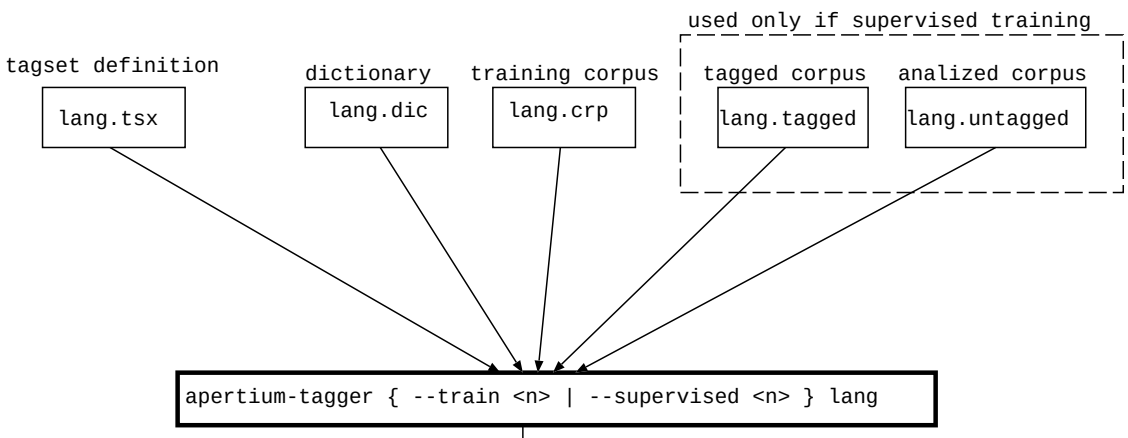
For the supervised training we need manually disambiguated text. The format of the text forms in this case will be like the format delivered by the morphological analyser (see Section 2) except that, being the text already disambiguated, a surface form can never produce more than one lexical form, as shown in Figure 3.4 (a *disambiguatedform* will consist always of a single *lexicalmultiform*).

$$\begin{aligned}
\text{disambiguatedform} &\rightarrow \text{lexicalmultiform} \\
\text{lexicalmultiform} &\rightarrow \text{lexicalform } [\text{ lexicalform }]^* \text{ lemma-queue?} \\
\text{lexicalform} &\rightarrow \text{lemma finetag} \\
\text{lemma-queue} &\rightarrow \text{lemma} \\
\text{finetag} &\rightarrow \text{morphsymbol } [\text{ morphsymbol }]^*
\end{aligned} \tag{3.4}$$

Finally, we need also the dictionary of the involved language to train the tagger. This dictionary is used to determine, in combination with the tagset specification, the different ambiguity classes with which the tagger will work.

Figure 3.30 shows the dependency diagram for the training and the use of the tagger.

Figure 3.30: Dependency diagram for the part-of-speech tagger.



3.3 Auxiliary module: transfer pre-processing module

3.3.1 Justification

The transfer pre-processing module `pretransfer` is in charge of separating compound multiwords (see page 43) and shifting certain parts of multiwords with inner inflection or *split lemma* forms. This module processes the tagger output and generates an entry suitable for the transfer module. The processing performed by this module is necessary for different reasons:

- So that the transfer module can process these units separately in order to deal with, for example, the movement of clitic pronouns when changing from enclitic to proclitic and vice versa.

- So that the bilingual dictionary only has to store information about the lemmas to be translated. If the particles that make up a multiword are included jointly in the bilingual dictionary, the dictionary would have to store an entry for each of the different combinations. By separating compound multiwords and processing multiwords with inner inflection, we can avoid having entries including inflection variations in the bilingual dictionary.

3.3.2 Behaviour and example

The program replaces each `<j/>` in the dictionary, that is, each `+` in the data stream, by a symbol for word end, a blank and a symbol for word beginning. Moreover, if the form is a multiword with split lemma, the queue is moved to the position between the first word of the multiword and its first grammatical symbol.

The task of generating an output which has the original order accepted by the generator, is left to the rules of the transfer module, which are also responsible for creating the compound multiwords which may be required in the target language. In general, the generator works with the same multiwords as the morphological analyser, and with the elements in the same order; that is the reason why this task has to be done in the transfer module.

We show below the result of applying this process to the compound multiword *darlo* ("give it" in Spanish):

```
$ pretransfer
^dar<vblex><inf>+lo<prn><enc><p3><m><sg>$      ⟵ input
^dar<vblex><inf>$ ^lo<prn><enc><p3><m><sg>$    ⟵ output
```

As can be seen, it consists only in dividing the lexical forms of a compound multiword into individual lexical forms.

When the input is a multiword with split lemma, the process is as shown in the following example for the Spanish multiword *echarte de menos* (”to miss you”):

```
$ pretransfer
^echar<vblex><inf>+te<prn><enc><p2><m><sg># de menos$
^echar# de menos<vblex><inf>$ ^te<prn><enc><p2><m><sg>$
```

Here, besides dividing into lexical forms, the module moves the invariable lemma queue into the mentioned position. As you can see, semantic units are maintained after the movement of the invariable queue, since we can consider *echar de menos* a verbal unit with own meaning.

3.4 Lexical selection module

3.4.1 Introduction

When the Apertium system is used to translate between less related languages than the ones dealt with in the first stages of the engine, the question of lexical selection becomes significant, because there are more cases, and more critical, in which a source language word can have more than one different translation in the target language. For this reason we created a new module, the lexical selection module, which deals with this problem.

Before going into its characteristics, we will see how the problems of *multiple equivalence* (the fact of existing more than one possible translation in target language for a source language lexical form) are tackled in Apertium in two ways.

On the one hand, we have the situation where there is no big difference in meaning between the multiple equivalents in the target language, and the fact of choosing one or the other can not lead to any translation error. We could say that between these equivalents there is a synonymy or quasi-synonymy relation. In such a case, the linguist chooses one of the lemmas as a translation (generally the most frequent or usual), and adds a direction restriction to the other lemmas (with the attributes `LR` or `RL`) so that they

are translated in the opposite direction but not in the direction where there are multiple equivalents.

On the other hand, we have the case where there is a clear difference in meaning between the multiple equivalents, which can lead to translation errors if the inappropriate lemma is chosen. These are the cases dealt with the new lexical selection module. The linguist has to encode entries with the attributes `slr` or `srl` described in the next section, thus identifying the different translation options; then, the lexical selection module, by means of statistical methods, chooses the translation which is most suitable in a given context.

Sometimes it is not easy to decide whether a multiple equivalence situation should be solved in one way or the other. For example, if there is difference in the meaning of two or more lemmas in the target language, but we think that the lexical selection module will not be capable of choosing the right translation by means of the context, we will follow the first method: choose a fixed translation (the most general, the most suitable in the maximum number of situations) and add a direction restriction to the rest of translations. In the other cases, we will encode the entries so that the decision is left to the lexical selection module.

When we use an Apertium system without lexical selection module, the only way to add entries with different possible translations is the first one, that is, choosing an only translation and marking the other equivalences with a direction restriction. In the event that we use bilingual dictionaries with multiple translations, encoded with the attributes `slr` or `srl`, in a system that does not have any lexical selection module, a style sheet will convert these entries designed for a lexical selection module into entries with direction restrictions `LR` or `RL`, so that one of the multiple equivalents (the one chosen as default entry by the linguist) becomes the fixed translation of the source language lemma.

As examples of bilingual equivalencies that should have a direction restriction, we can give the translation pairs `ca-es` *encara – aún/todavía* (”still”) and *sobtat – súbito/repentino* (”sudden”), the first one of which could be encoded like this:

```
<e r="LR">
   <p>
      <l>aún<s n="adv"/></l>
      <r>encara<s n="adv"/></r>
   </p>
</e>
```

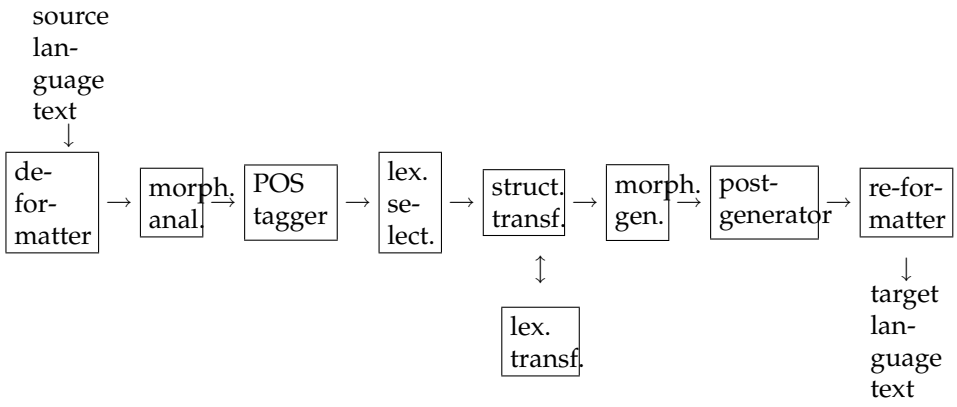


Figure 3.31: The nine modules that build the assembly line in the version 2 of the machine translation system Apertium.

```
<e>
    <p>
      <l>todavía<s n="adv"/></l>
      <r>encara<s n="adv"/></r>
    </p>
</e>
```

As examples of the second case (multiple equivalents with big difference in meaning) we have the pairs `es-ca` *hoja – full/fulla* (”sheet/leaf”) and *muñeca – nina/canell* (”doll/wrist”), as well as the `en-ca` examples shown in page 37, where it is described how to specify these multiple equivalents in the bilingual dictionary.

Figure 3.31 shows the new assembly line of the version 2 of Apertium.[7] The module in charge of the lexical selection (lexical selector) runs after the part-of-speech tagger and before the structural transfer module; therefore, this new module works only with source language information.

Section 3.4.2 next describes the pre-processing that must be done on a bilingual dictionary containing more than one translation per entry (whether the system uses a lexical selector or not), and Section 3.4.3 describes how the lexical selector works and how it has to be trained.

[7] This figure substitutes the figure 1.1 in page 6 which represents the version 1 of Apertium.

3.4.2 Pre-processing of the bilingual dictionaries

Bilingual dictionaries have been modified to allow the specification of more than one translation per entry (refer to Section 3.1.2.4 to learn how to write such dictionary entries); this fact makes it necessary to pre-process these dictionaries, since the Apertium engine works with compiled dictionaries in which there is only one possible translation for each word.

The pre-processing of dictionaries is done automatically during compilation, therefore the final user does not need to perform any specific action.

3.4.2.1 Pre-processing without lexical selection module

When bilingual dictionaries with multiple equivalents are used in a system where there is no lexical selection module, the pre-processing is done by the application of the style sheet `translate-to-default-equivalent.xsl`. This style sheet turns dictionaries with multiple translations per entry into dictionaries with only one translation per entry; to do this, it chooses as translation the entry marked as default, and adds a direction restriction (`LR` or `RL` as applicable) to the other entries, so that they are only translated in the translation direction where there is no equivalent multiplicity. The style sheet is called from the `Makefile`.

To put an example, the result of applying the style sheet on the first three entries shown in page 37 is the following:

```
<e>
   <p>
      <l>flat<s n="n"/></l>
      <r>pis<s n="n"/><s n="m"/></r>
   </p>
</e>

<e r="LR">
   <p>
      <l>floor<s n="n"/></l>
      <r>pis<s n="n"/><s n="m"/></r>
   </p>
</e>

<e r="RL">
   <p>
      <l>floor<s n="n"/></l>
      <r>terra<s n="n"/><s n="m"/></r>
```

```
   </p>
</e>
```

3.4.2.2 Preprocessing with lexical selection module

If the Apertium system works with a lexical selection module, the bilingual dictionary must be pre-processed in order to obtain:

- a monolingual dictionary that, for each source language word (for example *look*) delivers all the possible translation marks or equivalents (`look__mirar D` and `look__semblar`); this dictionary will be used by the lexical selection module; and

- a new bilingual dictionary that, given a word with the lexical selection already done (for example `look__semblar`) delivers the translation (*semblar*); this will be the bilingual dictionary to be used in the lexical transfer.

This pre-processing is automatically done by means of the following software during dictionary compilation:

- `apertium-gen-lextormono`, that receives three parameters:
  - the translation direction for which you want to generate the monolingual dictionary used in the lexical selection; `lr` for the translation left to right, and `rl` for the translation right to left;
  - the monolingual dictionary to be pre-processed; and
  - the file where the output monolingual dictionary has to be written.

- `apertium-gen-lextorbil`, that receives three parameters:
  - the translation direction (`lr` or `rl`) for which you want to generate the bilingual dictionary to be used by the lexical transfer module;
  - the bilingual dictionary to be pre-processed; and
  - the file where the output bilingual dictionary has to be written.

3.4.3 Execution of the lexical selection module

The module responsible for the lexical selection runs after the part-of-speech tagger and before the structural transfer (see Figure 3.31 in page 68); therefore, it uses only information from the source language. However, during the training of the module, target language information is also used.

3.4.3.1 Training

To train the lexical selection module, a corpus in the source language and another one in the target language are required; they do not need to be related. Both corpora must be pre-processed before the training. This pre-processing, consisting in analysing the corpora and performing the POS disambiguation, can be done with `apertium-preprocess-corpus-lextor`.

The training of the module that performs the lexical selection consists of the following tasks:[8]

1. Obtain the list of words that will be ignored when performing lexical selection (*stopwords*). This list can be done manually or using `apertium-gen-stopwords-lextor`;

2. Obtain the list of (source language) words that have more than one translation in the target language, using `apertium-gen-wlist-lextor`;

3. Translate to the target language all the words obtained in the previous step, using `apertium-gen-wlist-lextor-translation`;

4. Running `apertium-lextor --trainwrd` and using the target language pre-processed corpus, train a word co-occurrence model for the words obtained in the previous step;

5. Running `apertium-lextor --trainlch` and using the source language pre-processed corpus, the dictionaries generated by the programs mentioned in Section 3.4.2 and the word co-occurrence models calculated in the previous step, train a co-occurrence model for each of the translation marks of those words that can have more than one translation in the target language.

[8]The training of the models used for the lexical selection has been automated in all the packages using it. Furthermore, all the software mentioned has its UNIX manual page

3.4.3.2 Use

The word co-occurrence models calculated for each translation mark as described in the previous section provide the information required to perform lexical selection with information from the context.

Lexical selection is done by `apertium-lextor --lextor`; the formats used to communicate with the rest of the modules of the translation engine are:

Input: text in the same format as the input for the structural transfer module, that is, text analysed and disambiguated, with invariable queues of multiwords moved before morphological tags.

Output: text in the same format, but with the translation mark to be used when executing lexical transfer.

The following example illustrates the input/output formats used by the lexical selector (we have assumed in the example that only the English verb *get* has more than one translation equivalent in the dictionaries):

- Source language text (English): *To get to the city centre*
- Lexical selector input: `^To<pr>$ ^get<vblex><inf>$ ^to<pr>$ ^the<det><def><sp>$ ^city<n><sg>$ ^centre<n><sg>$`
- Translation marks in the en-ca bilingual dictionary for the verb *get*: `rebre`, `agafar`, `arribar`, `aconseguir D`
- Lexical selector output: `^To<pr>$ ^get__arribar<vblex><inf>$ ^to<pr>$ ^the<det><def><sp>$ ^city<n><sg>$ ^centre<n><sg>$`

3.5 Structural transfer module

3.5.1 Introduction

In 2007, Apertium incorporated a more advanced structural transfer system than the one used until then; it became necessary when we started developing machine translators for less related language pairs in comparison with the ones dealt with before, such as the *English–Catalan* translator.

This enhanced transfer system is made of three modules, the first one of which can be used in isolation in order to run a **shallow-transfer** system (which is the transfer system used so far for related language pairs such as *Spanish–Catalan* or *Spanish–Galician*). When the system is used for less related language pairs and, therefore, an **advanced transfer** becomes necessary, the three transfer modules will be executed.

The two transfer systems differ in the number of passes over the input text. The shallow-transfer system makes structural transformations with a single pass of the rules, which detect sequences or *patterns* of lexical forms and perform on them the required verifications and changes. On the other hand, the advanced transfer system works with a new architecture that allows to detect *patterns of patterns* of lexical forms with three passes, done by its three modules.

We describe next the characteristics of the structural transfer system. Section 3.5.2 describes the shallow-transfer system and Section 3.5.3, the advanced transfer system. The description of the shallow-transfer system is also applicable to the first module of the advanced transfer system, with the differences mentioned in that section. Section 3.5.4 describes the format used to create rules in both systems. In Section 3.5.5 there is a detailed description of how the three modules of the advanced transfer system work, and finally, Section 3.5.6 describes the pre-processing required by the modules.

3.5.2 Shallow-transfer

In this system, only the first of the three modules that compose the advanced transfer system is used. This module is called *chunker*.

The design of the language and the compiler used to generate the structural transfer module is largely based upon the MorphTrans language described in [5] and used by the MT systems interNOSTRUM [2, 5, 4] (Spanish–Catalan) and Traductor Universia [7, 17] (Spanish–Portuguese), developed by the Transducens group at the Universitat d'Alacant.

The transfer process is organized around patterns representing fixed-length sequences of source language lexical forms (SLLFs) (see page 7 for a description of lexical form (LF)); a sequence follows a certain pattern if it contains the sequence of lexical forms of the pattern. Patterns do not need to be constituents or phrases in the syntactic sense: they are mere concatenations of lexical forms that may need a conjoint processing additional to the simple word-for-word translation, due to the grammatical divergences between SL and TL (gender and number changes, reorderings, prepositional changes, etc). The catalogue of patterns defined for a certain language is selected with a view to covering the most common structural transformations. When source language and target language are syntactically similar, as is the case between Spanish, Catalan and Galician, simple rules based on sequences of lexical categories achieve a reasonable translation quality.

The transfer module detects, in the SL, sequences of lexical forms that match one of the patterns previously defined in the pattern catalogue, and processes them applying the corresponding structural transfer rule, doing at the same time the lexical transfer by reading the bilingual dictionary.

The *pattern detection* phase occurs as follows: if the transfer module starts to process the i-th SLLF of the text, l_i, it tries to match the sequence of SLLFs $l_i, l_{i+1}, \ldots$ with all of the patterns in its pattern catalogue: the longest matching pattern is chosen, the matching sequence is processed (see below), and processing continues at SLLF l_{i+k}, where k is the length of the pattern just processed. If no pattern matches the sequence starting at SLLF l_i, it is translated as an isolated word an processing restarts at SLLF l_{i+1} (when no patterns are applicable, the system resorts to word-for-word translation). Note that each SLLF is processed only once: patterns do not overlap; hence, processing occurs left to right and in distinct "chunks".

In the *pattern processing* phase, the system takes the detected sequence of SLLFs and builds (using a program to consult the bilingual dictionary) a sequence of TL lexical forms (TLLFs) obtained after the application of the operations described in the rule associated to the detected pattern (reordering, addition, replacement or deleting of words, inflection changes, etc.). The information that does not change is automatically copied from SL to TL. The resulting data, that is, the lemmas with their associated morphological tags, are sent to the generator, which creates the inflected forms.

For instance, the Spanish sequence *una señal inequívoca* ("an unmistakable signal"), that would go from the tagger to the transfer module in the following format [9]:

[9]The example has been presented in a way that it does not contain superblanks with

```
^uno<det><ind><f><sg>$
^señal<n><f><sg>$
^inequívoco<adj><f><sg>$
```

would be detected as a pattern by a rule for determiner–noun–adjective. The transfer module would consult the bilingual dictionary to get the Catalan equivalents and, as it would detect a gender change in the word *señal* (its Catalan translation *senyal* is masculine), it would propagate this change to the determiner and the adjective to deliver the output sequence:

```
^un<det><ind><m><sg>$
^senyal<n><m><sg>$
^inequívoc<adj><m><sg>$
```

which the generation module would turn into the Catalan inflected sequence: *un senyal inequívoc*.

The task of most rules is to ensure gender and number agreement in simple noun phrases (determiner–noun, determiner–noun–adjective, determiner–adjective–noun, determiner–adjective, etc.), provided that there is agreement between the SLLFs of the detected pattern. These rules are required either because the noun changes its gender or number between SL and TL (as in the previous example) or because gender or number in the TL have to be determined due to the fact that it was ambiguous in SL for some of the words (for example, the Catalan determiner *cap* can be translated into Spanish as *ningún* (masc.) or *ninguna* (fem.) depending on the accompanying noun: *cap cotxe* (ca) → *ningún coche* (es) and *cap casa* (ca) → *ninguna casa* (es)). Furthermore, there other rules defined to solve frequent transfer problems between Spanish, Catalan and Galician, such as, among others:

- rules to change prepositions in certain constructions: *in Barcelona* (es) → *a Barcelona* (ca); *consiste en hacer* (es) → *consisteix a fer* (ca);

format information, so that the linguistic side of the transformation is clearer. See Chapter 2.

- rules to add/remove the preposition *a* in certain Galician modal constructions with the verbs *ir* and *vir*: *vai comprar* (`gl`) → *va a comprar* (`es`);

- rules for articles before proper nouns: *ve la Marta* (`ca`) → *viene Marta* (`es`);

- lexical rules, for instance, to decide the correct translation of the adverb *molt* (`ca`) into Spanish (*muy, mucho*) or of the adjective *primeiro* (`gl`) or *primer* (`ca`) into Spanish (*primer, primero*);

- rules to displace atonic or clitic pronouns, whose position in Galician is different to that in Spanish (proclitic in Galician and enclitic in Spanish or vice versa): *envioume* (`gl`) → *me envió* (`es`); *para nos dicir* (`gl`) → *para decirnos* (`es`).

Multiwords (its different types are described in page 43) are processed in a special way in this module:

- *Multiwords without inflection*, made of only one lexical form, do not need any special processing, since they are treated like other LFs.

- In the case of *compound multiwords*, that is, multiwords formed by more than one *lexical form*, each one with its own grammatical symbols and joined to each other with the element `<j>` in the dictionary entry (which corresponds to the symbol '+' in the data stream), the auxiliary module `pretransfer` (see 3.3), located before this module, separates the different lexical forms so that they reach the transfer module as independent LFs. If we want to join them again so that they reach the generator as multiwords (as is the case of enclitic pronouns in our system), it has to be done by means of a transfer rule, using the <**mlu**> element (described later, in section 3.5.4.42). In page 145 you can find an example of a rule for joining enclitic pronouns to the verb.

- As for *multiwords with inner inflection*, the `pretransfer` module moves the lemma queue (the invariable part) to place it after the lemma head (the inflective form), thus making possible to find the multiword in the bilingual dictionary. This kind of multiwords must be processed by a structural transfer rule which replaces the lemma queue in its proper position. This is done by using, in the output of the rule, the attributes `lemh` "lemma head" and `lemq` "lemma queue") of the <**clip**> element. See page 101 for a more detailed

description of the use of this element, and page 145 to see two rules where these attributes are used.

3.5.3 Advanced transfer

The shallow-transfer architecture described in the previous section is based, as we have seen, in the automatic handling of word co-occurrence patterns by means of rules defined by the user. This model considers two levels from the point of view of the nature of data: a basic level we call *lexical level*, which handles words and the tasks of consulting and changing its characteristics (lemma and tags), besides translating individual lemmas by asking the bilingual dictionary; and another level we call *word pattern level*, which is in charge of doing, when applicable, reorderings of the words that build these patterns, as well as changes in the properties of words that depend on the specific pattern that has been detected. All this process of detection and manipulation of words and patterns is carried out in a single pass.

In contrast, the new advanced transfer architecture is defined as a transfer system in three levels and three passes. The first two levels, lexical and pattern level, are the same ones of the shallow-transfer system. The new added level is a level of *patterns of patterns* of words. The aim of this new processing level is to allow the handling and interaction of patterns of words in a similar way as words are handled in the patterns of the shallow system. With this new structure we intend to achieve a more appropriate handling of all transformations that may be required when translating from one language to another. We want to emphasize that the definition of word patterns in the shallow-transfer system does not need to be the same as the definition of word patterns in the advanced system: we pretend that, in the latter, patterns have a *spirit* of phrases that does not exist in the previous system. Therefore we will use the term *chunk* to refer to word sequences in the advanced transfer system.

The advanced transfer system is organized in three passes. According to the Apertium processing mode, these three passes are carried out by three different modules (programs):

- `chunker`: identifies chunks, translates word for word, and carries out required reorderings and morphosyntactic data propagation inside the chunk (for example, to maintain agreement). Besides, it creates the chunks that will be processed by the next module. The `chunker` has the option of running as a single module in a shallow-transfer system. This is controlled by an attribute in the `<transfer>`

element.

- `interchunk`: this module receives the chunks generated by the `chunker` and is able to reorder them, modify the "syntactic information" associated to each chunk and, finally, output the chunks in the new order and with the new properties, creating new chunks if needed.

- `postchunk`: it receives the chunks modified by the interchunk and carries out final tasks concerning modification of the words contained in each chunk and printing of the text contained in chunks in the format accepted by the generator.

In the following lines we specify the format of the chunks that circulate between the modules of the transfer system (Section 3.5.3.1) and the letter case handling in chunks (Section 3.5.3.2), which is different from case handling of individual lexical forms in a shallow-transfer system.

The following section, 3.5.4, describes the format of transfer rules, which is the same for the three modules and the two transfer modes, with little differences. Finally, after this description, in 3.5.5 you will find a more detailed explanation of the three modules that make up an advanced transfer system.

3.5.3.1 Chunk format

Communication between `chunker` and `interchunk`, as well as between `interchunk` and `postchunk`, is performed through sequences of chunks. We define C as a *sequence of chunks*, that has the form:

$$C = b_0 c_1 b_1 c_2 b_2 \ldots b_{k-1} c_k b_k$$

where each b_i is a *superblank*, and each c is a *chunk*. A chunk c is defined as a string `^`F{W}`$` that contains the following information:

- F is the *lexical pseudoform*; it is a string that has the form fE, where f is the *pseudolemma* of the chunk, and $E = e_1 e_2 \ldots$ is a sequence of grammatical symbols called *chunk symbols*. Changing these symbols will cause the changing of the morphological information of words in the chunk, if this information is linked to these parameters.

- $W = b_0 w_1 b_1 w_2 b_2 \ldots w_k b_k$ is the sequence of words w_i sent by the chunker with the intermediate *superblanks* b_i. These words have the same format in both transfer systems, that is, an individual word

w_i =^$l_i E_i$$ contains lemma l_i and grammatical symbols E_i, some of which can be *references or links to the symbols* of the chunk and are identified with natural numbers <1>, <2>, <3>, etc. These references to symbols correspond, in the specified order, to the symbols of E.

The following is a use example of the described format, with the text *el gat* ("the cat"):

```
^det_nom<SN><m><pl>{^el<det><def><2><3>$[
<a href="http://www.ua.es">]^gat<n><2><3>$}$[</a>]
```

The characters { and }, if present in the original text, must be escaped with a backslash \.

3.5.3.2 Letter case handling

For each chunk, the case of words is determined by the case of the pseudolemma of the chunk, taking into account the following rules:

- When all the letters of the pseudolemma are in lower case: the case state of words is not modified.
- When the first letter of the pseudolemma is in upper case and the rest are in lower case: in the module postchunk, when words are printed, the letter that is the first of the chunk after all the possible word reorderings will be put in upper case .
- When all the letters of the pseudolemma are in upper case: all the words will remain upper case.

It is required that the words in the chunk are not capitalized unless they are proper nouns, so as to avoid the postchunk module having to look for the word that has to lose capitalization, if this is the case. This task belongs to the chunker module and is done with a macro or similar mechanism.

3.5.4 Format specification for structural transfer rules

This section describes the format in which structural transfer rules are written. In the Appendix, in sections A.3, A.4 and A.5, there is the formal definition (DTD).

Structural transfer rules files have two well-differentiated parts: one for the declaration of the elements to be used in rules, and another one for

the rules themselves.

In the **declaration** part we find:

- A series of declarations of *lexical categories*, which specify those lexical forms that will be treated as a particular category and will be detected by patterns. The linguist may include any data about the lexical form to define a category; categories can be very generic (i.e. all the nouns) or very specific (i.e. only those determiners that are demonstrative feminine plural).

- A series of declarations of the *attributes* we want to detect in lexical forms (like *gender*, *number*, *person* or *tense*), to perform with them the required transformation operations and send the resulting data in the output of the rules. The declaration of an attribute contains the name of the attribute and the possible values it can take in a lexical form (in general they correspond to the morphological attributes that characterize the form): for example, the attribute *number* can take the values *singular*, *plural*, *singular-plural* (for invariable lexical forms, like *crisis* in Spanish) and *number to be determined* (for TL lexical forms with different forms for *singular–plural*, but whose number can not be determined in the translation due to the fact that the SL lexical form is invariable in number, see explanation in page 40). If inside the rule, outside of the pattern, one wishes to refer to any of the lexical categories defined in the previous point (to perform tests or actions on them), it will be also necessary to define attributes for them.

- A series of declarations of *global variables*, which are used to transfer values of active attributes inside a rule, or from one rule to the ones applied subsequently.

- A section for the *definition of string lists*, generally lists of lemmas, which will be used to make searches on them for a certain value to perform a specific transformation.

- A series of declarations of *macro-instructions*; macro-instructions contain sequences of frequently used instructions, and can be included in different rules (for example, a macro-instruction to ensure gender and number agreement between two lexical forms of a pattern).

In the **structural transfer rules** we find:

- The definition of the pattern that will be detected, specified as a sequence of lexical categories as they have been defined in the declaration part. It must be noted that, if a sequence of lexical forms matches two different rules, firstly, the longest is chosen, and secondly, for rules of the same length, the one defined before is chosen.

- The process part of the rules, where actions to be performed on SLLF are specified, and the TL pattern is built.

In the following pages we describe in detail the characteristics of all the elements used in rules.

3.5.4.1 Element `<transfer>`

(*Only in the chunker module*)

This is the root element of the `chunker` module and contains all the rest of the elements of the structural transfer rules file of this module.

Its attribute `default` can take two values:

- `lu`: it means that it will run in shallow mode, that is, as only transfer module in a shallow-transfer system and, therefore, no special action will be done on words not detected by any pattern

- `chunk`: it means that it will run in advanced mode and, therefore, when a word is not recognized by any rule, a chunk will be created to encapsulate it, so that it can be processed by the next transfer modules of an advanced transfer system.

The default value is `lu`.

3.5.4.2 Element `<interchunk>`

(*Only in interchunk*)

This is the root element of the `interchunk` module and contains all the rest of the elements of the structural transfer rules file of this module.

3.5.4.3 Element `<postchunk>`

(*Only in postchunk*)

This is the root element of the `postchunk` module and contains all the rest of the elements of the structural transfer rules file of this module.

3.5.4.4 Element for category definition section <section-def-cats>

This section contains the definition of the lexical categories that will be used to create the patterns used in rules. Each definition is made with a <**def-cat**>.

3.5.4.5 Element for category definition <def-cat>

Each category definition has a mandatory name `n` (e.g. `det`, `adv`, `prep`, etc.) and a list of categories (<**cat-item**>) that define it. The name of the category can not contain accents.

3.5.4.6 Element for category <cat-item>

This element has two well-differentiated uses depending on the module it is used in.

Use in chunker (shallow transfer and advanced transfer)

This element defines the lexical categories that will be used in patterns, that is, that the linguist wishes to detect in the source text. These categories are defined by a subsequence of the fine tags (see definition in page 57) that deliver both the morphological analyser and the tagger[10].

Each <**cat-item**> element has a mandatory attribute `tags` whose value is a sequence of grammatical symbols separated by a dot; this sequence is a subsequence of the fine tag, that is, of the sequence of grammatical symbols that defines every possible lexical form delivered by the tagger. According to this, a category represents a certain set of lexical forms. We must define as many different categories as kinds of lexical forms we want to detect in patterns. Thus, if we want to detect all the nouns to perform certain actions on them, we will create a category defined with the grammatical symbol `n`. On the other hand, if we want to

[10]Please note that throughout the different linguistic modules, different lexical categorizations are used: in morphological dictionaries, lemmas are accompanied by a fine tag (for instance, *<n><m><pl>* for plural masculine nouns); the POS tagger groups these fine tags in more general tags (for instance, the category `NOUN` for all the nouns), although its output is again the whole fine tag of each LF; finally, in the transfer module, the fine tags of LFs are grouped again in more general categories (although it is also possible to define particularized categories) depending on the type of lexical forms that one wants to detect in patterns.

```
<def-cat n="nom"/>
  <cat-item tags="n.*"/>
</def-cat>

<def-cat n="que"/>
  <cat-item lemma="que" tags="cnjsub"/>
  <cat-item lemma="que" tags="rel.an.mf.sp"/>
</def-cat>
```

Figure 3.32: Use of the <**cat-item**> element to define two categories, one for nouns without lemma specification (*nom*), which includes all lexical forms whose first grammatical symbol is *n*, and another one with associated lemma (*que*), which has two subsequences of fine tags, to include the *que* conjunction and the *que* relative pronoun.

detect all the plural feminine nouns, we will have to define a category using the symbols `n f` and `pl`.

When, for the set of lemmas we want to include in a category, a grammatical symbol used to define the category is followed by other grammatical symbols, the character `"*"` is used. For example, `tags="n.*"` covers all the lexical forms that contain this symbol, such as the Spanish nouns `casa<n><f><pl>` or `coche<n><m><sg>`. On the other hand, when after the used symbol there can not be any other symbol, the asterisk is not included: for example, `tags="adv"` will cover all adverbs, since in our system they are characterized with only one grammatical symbol. The asterisk can also be used to signal the existence of preceding symbols: `tags="*.f.*"` includes all feminine lexical forms, whichever category they are. Furthermore, an optional attribute, `lemma`, can be used to define lexical forms on the basis of its lemma (see Figure 3.32).

Use in interchunk

It is used like in the `chunker` module, but here, instead of being defined with the grammatical symbols of lexical forms, it is defined with the symbols of the chunks delivered by the `chunker`. For example, in the case that we want to define a category to detect all the determined noun phrases, we will define it with the symbols `NP` and `DET` if this is how we tagged these chunks by means of the `<tag>` instructions contained in the `<chunk>` element (see 3.5.5.1). You can also use the optional attribute `lemma` to refer to the *pseudolemma* of the chunk. So, its formal characteristics are the same in the modules `chunker` and `interchunk`, with the difference that in the

```
<def-cat n="det-nom"/>
  <cat-item name="det-nom"/>
</def-cat>
```

Figure 3.33: Use of the <**cat-item**> element in the postchunk to detect chunks of determiner-noun.

former they are used to detect lexical forms, and in the latter, to detect chunks.

Use in postchunk

In this module, this element only has the mandatory attribute `name`, which refers to the name of the chunk,

without tags, since in the `postchunk` module only the pseudolemma (name of the chunk) is used for detection. Case is ignored in detection, because the pseudolemma is used to convey information about the case of the chunk. (See Figure 3.33).

3.5.4.7 Element for category attribute definition section <section-def-attrs>

This section is to describe the attributes that will be extracted from the categories detected by the pattern and that will be used in the action part of the rules. Each attribute is defined by a <**def-attr**> tag.

3.5.4.8 Element for category attribute definition <def-attr>

Each <**def-attr**> defines an attribute regarding morphological information (both inflection information –gender, number, person, etc.–, and categorial –verb, adjective, etc–) by specifying a list of category attribute (<**attr-item**>) elements, and has a mandatory unique name `n`. Therefore, an attribute is defined on the basis of the grammatical symbols that can be found in a given lexical form. Each attribute extracts, from the lexical forms of the pattern, the symbols that these contain among the set of possible values defined.

```
<def-attr n="nbr"/>
  <attr-item tags="sg"/>
  <attr-item tags="pl"/>
  <attr-item tags="sp"/>
  <attr-item tags="ND"/>
</def-attr>

<def-attr n="a_nom"/>
  <attr-item tags="n"/>
  <attr-item tags="n.acr"/>
</def-attr>
```

Figure 3.34: Definition of the category attribute `nbr` for *number*, which can take the values *singular*, *plural*, *singular-plural* or *number to be determined*, and the category attribute `a_nom` for *noun*, which can take the values of the symbols *n* or *n acr*.

3.5.4.9 Element for category attribute <attr-item>

Each category attribute element represents one of the possible values the attribute can take. For example, the attribute for number `nbr` can take the values singular `sg`, plural `pl`, singular–plural `sp` and number to be determined `ND`. These values are a subsequence of the morphological tags that characterize each lexical form, and are specified in the `tags` attribute of the element, separated by a dot if there is more than one. In Figure 3.34 you can find an example for the attributes for *number* and *noun*.

Compare the definition of the attribute for number in this figure (with all possible values and without asterisks) with the definition of the category for noun in Figure 3.32.

3.5.4.10 Element for variable definition section <section-def-vars>

In this section, <**def-var**> tags are used to define global string variables, that will be used to transfer information inside the rule and from one rule to another one (for example, to transmit information on gender or number between two patterns)

3.5.4.11 Element for variable definition <def-var>

The definition of a global string variable has a mandatory unique name `n` that will be used to refer to it inside a rule. Variables contain strings that describe state information, such as the existence of agreement between two elements, the detection of a question mark in SL that should be deleted in TL, etc.

3.5.4.12 Element for string lists definition section <section-def-lists>

In this section, lists are defined (with <**def-list**> tags) that will be used to do string searches. These lists can be used to group word lemmas that have a common feature (i.e. verbs expressing movement, adjectives expressing emotions, etc.). This section is optional.

3.5.4.13 Element for string lists definition <def-list>

This element is used to name the string list, with the attribute `n`, and to encapsulate the list defined by one or more <**list-item**> elements. An example of its use can be found in Figure 3.35.

3.5.4.14 Element for string list item <list-item>

It defines, with the value of the attribute `v`, the specific string that is included in the definition of the list. An example of its use can be found in Figure 3.35.

3.5.4.15 Element for macro-instruction definition section <section-def-macros>

This section is for the definition of macro-instructions that contain pieces of code used frequently in the action part of the rules.

3.5.4.16 Element for macro-instruction definition <def-macro>

Each macro-instruction definition has a mandatory name (the value of the attribute `n`), the number of arguments it receives (attribute `npar`) and a body with instructions.

```
<def-list n="verbos_est">
  <list-item v="actuar"/>
  <list-item v="buscar"/>
  <list-item v="estudiar"/>
  <list-item v="existir"/>
  <list-item v="ingressar"/>
  <list-item v="introduir"/>
  <list-item v="penetrar"/>
  <list-item v="publicar"/>
  <list-item v="treballar"/>
  <list-item v="viure"/>
</def-list>
```

Figure 3.35: Definition of a list of Catalan lemmas. These lemmas are used in the rule in Figure 3.40.

3.5.4.17 Element for rules section <section-rules>

This section contains the structural transfer rules, each one in a <**rule**> element.

3.5.4.18 Element for rule <rule>

Each rule has a pattern (<**pattern**>) and the associated action (<**action**>) performed when the pattern is matched.

The rule can have an optional attribute `comment` with a comment on, usually, the function of the rule.

3.5.4.19 Element for pattern <pattern>

A pattern is specified using pattern items (<**pattern-item**>), each one of which corresponds to a lexical form in the matched pattern, in order of appearance.

3.5.4.20 Element for pattern constituent <pattern-item>

Each pattern item specifies, in the attribute with mandatory name `n`, which kind of lexical form is to be matched. To do that, one has to use the categories defined in <**section-def-cats**> (see in Figure 3.42 the definition of a pattern for determiner–noun).

3.5.4.21 Element for action <action>

This element contains the "instructions" that have to be executed to process as desired each matched pattern.

The processing part for matched patterns is a block of zero or more instructions of the kind: <**choose**> (conditional processing), <**let**> (value assignment), <**out**> (print TL lexical forms), <**modify-case**> (modify case state of a lexical form), <**call-macro**> (call a macro-instruction) and <**append**> (concatenate strings).

Through the processing step, depending on whether a series of conditional options are met or not, different operations are carried out, such as creating agreement between pattern components, necessary when these undergo gender or number changes in the lexical transfer process. To do this, in spite of working with TLLF, also the SL information is taken into account, since, for example, if pattern components do not agree in SL, maybe they do not have to agree in TL either. As a consequence of the application of the different operations in a pattern, values are assigned to pattern attributes and, if applicable, to global or state variables, and the information on the resulting TL pattern is sent to the next module (the morphological generator in a shallow-transfer system, or the next transfer module in an advanced transfer system).

3.5.4.22 Element for macro-instruction call <call-macro>

In a rule it is possible to call any of the macro-instructions defined in <**section-def-macros**>. To do this, one has to specify the name of the macro-instruction in the `n` attribute, and one or more arguments in the parameter element <**with-param**> (see next).

3.5.4.23 Element for parameters <with-param>

This element is used inside a macro-instruction call <**call-macro**>. The `pos` attribute of an argument is used to refer to a lexical form of the rule from where the macro-instruction is called. For example, if a macro-instruction with 2 parameters has been defined, to make agreement operations between noun–adjective, it can be used with arguments 1 and 2 in a rule for noun–adjective, with arguments 2 and 3 in a rule for determiner–noun–adjective, with arguments 1 and 3 in a rule for noun–adverb–adjective and with arguments 2 and 1 in a rule for adjective–noun. You can see an example of macro-instruction call in Figure 3.36.

```
<call-macro n="f_concord2">
  <with-param pos="3"/>
  <with-param pos="1"/>
</call-macro>
```

Figure 3.36: Call of the macro-instruction `f-concord2` designed to create agreement between two elements in a pattern such as determiner–adverb–noun. Propagation of gender and number is done from one of the components, in this case, from the noun which is the third element of the pattern (3). Therefore, the position of the noun is the first parameter given, and the other parameters come next. Since the adverb (in position 2) does not need agreement information, only the position of the determiner is specified (1).

3.5.4.24 Element for selection <choose>

The selection instruction consists of one or more conditional options (<**when**>) and an alternative option <**otherwise**>, which is optional.

3.5.4.25 Element for condition <when>

This element describes a conditional option (see Section 3.5.4.24). It contains the condition to be tested <**test**> and one block of zero or more instructions of the kind <**choose**>, <**let**>, <**out**>, <**modify-case**>, <**call-macro**> or <**append**>, which will be executed if the above condition is met.

3.5.4.26 Element for alternative option <otherwise>

The element <**otherwise**> contains one block of one or more instructions (of the kind <**choose**>, <**let**>, <**out**>, <**modify-case**>, <**call-macro**> and <**append**>) that must be executed if none of the conditions described in the <**when**> elements of a <**choose**> is met.

3.5.4.27 Element for evaluation <test>

The test element <**test**> in a condition element <**when**> can contain a conjunction (<**and**>), a disjunction (<**or**>) or a negation (<**not**>) of conditions to be tested, as well as a simple condition of string equality (<**equal**>), string beginning (<**begins-with**>), string end (<**ends-with**>), substring (<**contains-substring**>) or inclusion in a set (<**in**>).

3.5.4.28 Elements for conditional or boolean operators: <equal>, <and>, <or>, <not>, <in>

- The conjunction element <**and**> represents a condition, consisting of two or more conditions, that is met when all included conditions are true. An example of its use can be found in Figure 3.42.

- The disjunction element <**or**> represents a condition, consisting of two or more conditions, that is met when at least one of the included conditions is true. Figure 3.39 displays an example of this condition type used when testing gender agreement in a SL pattern.

- The negation element <**not**> represents a condition that is met when the included condition is not met, and vice versa. An example of negation of an equality can be found in Figure 3.39.

- The conditional equality operator <**equal**> is an instruction that evaluates if two arguments (two strings) are identical or not. See examples of its use in Figures 3.37 and 3.38. In addition, this operator can have the attribute `caseless`, which, when its value is `yes`, causes the comparison of strings to be made ignoring case.

- The "search in lists" operator <**in**> is used to search for any value (specified as the first parameter of the condition) in a list referred to by the `n` attribute of the <**list**> element; this list must be defined in the appropriate section (<**section-def-lists**). The search result is true if the value is found in the list. This comparison can also use the attribute `caseless`: if its value is `yes`, the search is done ignoring case. Figure 3.40 shows an example of its use.

3.5.4.29 Element <clip>

The <**clip**> element represents a substring of a SL or TL lexical form, defined by the value of its different attributes (see an example in Figure 3.37):

- `pos` is an index (1, 2, 3, etc.) used to select a lexical form inside a rule: it refers to the place the lexical form occupies in the pattern. In the *postchunk* module there is also the index "0", which refers to the pseudolemma of the chunk , which is treated as a word by itself in order to be able to consult its information and make decisions from this.

```
<test>
  <not>
    <equal>
      <clip pos="2" side="tl" part="gen"/>
      <clip pos="2" side="sl" part="gen"/>
    </equal>
  </not>
</test>
```

Figure 3.37: Extract from a rule where it is tested whether the TL (`tl`) gender (`gen`) of the second lexical unit identified in a pattern is different from the gender of the same lexical unit in the SL (`sl`)

.

- `side` *(only in the `chunker` module)* specifies if the selected *clip* is from the source language (`sl`) or from the target language (`tl`).

- `part` indicates which part of the lexical form is processed; generally its value is one of the attributes defined in <**`section-def-attrs`**> (`gen`, `nbr`, etc.), although it can also take four predefined values: `lem` (refers to the lemma of the lexical form), `lemh` (the first part of a split lemma), `lemq` (the queue of a split lemma), and `whole` (the whole lexical form, including lemma and all grammatical symbols, which may have been modified in the preceding part of the rule).

- `link-to` *(only in the `chunker` module in advanced mode)* replaces the value that would result from consulting the rest of the attributes of the clip, by the value specified in this attribute, which must be a natural number (> 0). This number indicates to which <**`tag`**> of the <**`chunk`**> is linked the clip content, the number being the order this tag occupies inside the element <**`tags`**>. The other attributes of the clip remain only for informational purposes, since they are overwritten by the value of the linked tag. An example of its use can be found in Figure 3.46.

3.5.4.30 Element for literal string <lit>

This element is used to specify the value of a literal string by means of the attribute `v`. For example, <**`lit`** `v="andar"/>` represents the string *andar*.

```
<equal>
  <clip pos="2" side="tl" part="nbr"/>
  <lit-tag v="ND"/>
</equal>
```

Figure 3.38: Use of the element <**lit-tag**>: it is tested whether the number (`nbr`) symbol of the second lexical unit in the TL (`tl`) is `ND` (number to be determined)

3.5.4.31 Element for tag value <lit-tag>

It is similar to the <**lit**> element, with the difference that it does not specify the value of a literal string but the value of a grammatical symbol or tag, by means of the attribute `v`. An example of its use can be found in Figure 3.38.

3.5.4.32 Element for variable <var>

Each <**var**> is a variable identifier: the mandatory attribute `n` specifies its name as has been defined in <**section-def-vars**>. When it appears in an <**out**>, a <**test**>, or the right part of a <**let**>, it represents the value of the variable; when it appears on the left side of a <**let**>, in an <**append**> or in a <**modify-case**>, it represents the reference of the variable and its value can be changed.

3.5.4.33 Element for reference to string list <list>

This element is only used as the second parameter of a <**in**> search. The `n` attribute refers to the specific list defined in the string lists definition section <**section-def-lists**>. An example of its use can be found in Figure 3.40.

3.5.4.34 Element for case application <get-case-from>

The <**get-case-from**> element represents the string obtained after applying the letter case state of the lemma of a SL lexical unit to a string (*clip*, *lit* or *var*). To refer to the lexical unit from where the information is taken, the attribute `pos` is used, which indicates the position of that unit in the SL. This element is useful when the lexical units in a pattern are reordered, or when a lexical unit is added or deleted. You can see an example of its use in Figure 3.41, which displays a rule to transform the simple perfect

```
<test>
 <or>
   <not>
     <equal>
       <clip pos="1" side="sl" part="gen"/>
       <clip pos="3" side="sl" part="gen"/>
     </equal>
   </not>
   <not>
     <equal>
       <clip pos="2" side="sl" part="gen"/>
       <clip pos="3" side="sl" part="gen"/>
     </equal>
   </not>
 </or>
</test>
```

Figure 3.39: Extract from a rule where it is tested whether the SL gender of the first or the second lexical unit matched in a pattern (it could be, for example, determiner–adjective–noun) is different from the gender of the third lexical unit also in the SL.

preterite tense in Spanish (*dije*, "I said") into the compound form in Catalan (*vaig dir*). In this rule, a LF with lemma *anar* and grammatical symbol *vaux* ("auxiliary verb") is added; it has to take the case information from the Spanish verb (which has position "1" in the pattern), so that the system translates *Dije* as *Vaig dir*, *dije* as *vaig dir* and *DIJE* as *VAIG DIR*.

3.5.4.35 Element for case pattern query <case-of>

It is used to get the case pattern of a string, that is, one of the values "aa", "Aa" or "AA". It works like the <**clip**> element, since it has the same attributes: pos, the position of the word in the matched pattern; part, the specific attribute that we refer to (normally the lemma), which has the predefined attributes described in Section 3.5.4.29, and finally, only in the chunker module, the attribute side, referring to the translation side, sl or tl. In Figure 3.41 you can see this element in use, and you can find a more detailed description of this example in the following Section (description of <**modify-case**>).

```
<rule>
  <pattern>
    <pattern-item n="verb"/>
    <pattern-item n="a"/>
  </pattern>
  <action>
  <choose>
    <when>
      <test>
        <in caseless="yes"/>
          <clip pos="1" side="sl" part="lem"/>
          <list n="verbos_est"/>
        </in>
      </test>
      <let>
        <clip pos="2" side="tl" part="lem"/>
        <lit v="en"/>
     </let>
  </when>
  <!-- ... -->
```

Figure 3.40: Extract of a rule that detects a pattern made of a verb and the preposition *a*, and then testes whether the verb (the lemma indicated in `lem`) of the source language (`sl`) is one of the lemmas included in the list of state verbs (defined in Figure 3.35). If that be the case, the lemma of the second word in target language (`tl`) is changed to *en*.

3.5.4.36 Element for case modification <modify-case>

This instructions is used to modify the case of the first parameter (usually a lemma) by means of the second parameter (a literal or a variable). The first parameter can be a <**var**>, a <**clip**> or a <**case-of**>, whereas the second one can be anything that delivers a value, but in principle it will be a <**var**> or a <**lit**>. The values that this value can take are usually "Aa", to express that the "left part" of this case modification must have the first letter in upper case and the rest in lower case, "aa" to put all in lower case, and "AA" to put all in upper case.

Figure 3.41 shows a rule where this element is used. It modifies in this rule the case of the TL lemma in position "1", which corresponds to *dir*, because, although in the rule output this verb is the second lexical form (*vaig dir*), it is actually the translation of the LF which has position 1 in the SL, and, therefore, it retains the same assigned position in the TL. This lemma is assigned the value "aa" in the case that the SL lemma has the state "Aa". There is nothing to specify for the rest of the cases, since the case state of the LF in position 1 will be the same in the SL and in the TL and, therefore, will be automatically transferred (see Section 19 to obtain more information on letter case handling in dictionaries).

3.5.4.37 Element for assignment <let>

The assignment instruction <**let**> assigns the value of the right part of the assignment (a literal string, a clip, a variable, etc.) to the left part (a clip, a variable, etc.). An example of its use can be found in Figure 3.42.

3.5.4.38 Element for string concatenation <concat>

This element is used to concatenate strings in order to assign them to a variable. It is used in combination with <**let**>, and the previous value of the variable is lost with the assignment of <**concat**>.

It does not have any attribute. It can contain any instruction that delivers a string, such as <**lit**>, <**lit-tag**> or <**clip**>.

Figure 3.43 shows an example of its use.

3.5.4.39 Element for string concatenation <append>

The <**append**> instruction can be used to save the output of an action before printing it in the corresponding <**out**>, if required by the designer of the transfer rules.

```
<rule>
  <pattern>
    <pattern-item n="pretind"/>
  </pattern>
  <action>
    <out>
      <lu>
         <get-case-from pos ="1">
           <lit v="anar"/>
         </get-case-from>
         <lit-tag v="vaux"/>
         <clip pos="1" side="sl" part="persona"/>
         <clip pos="1" side="sl" part="nbr"/>
       </lu>
       <b/>
     </out>
     <choose>
       <when>
         <test>
           <equal>
              <case-of pos="1" side="sl" part="lemh"/>
              <lit v="Aa"/>
           </equal>
         </test>
         <modify-case>
             <case-of pos="1" side="tl" part="lemh"/>
             <lit v="aa"/>
         </modify-case>
       </when>
     </choose>
     <out>
       <lu>
          <clip pos="1" side="tl" part="lemh"/>
          <clip pos="1" side="tl" part="a_verb"/>
          <lit-tag v="inf"/>
          <clip pos="1" side="tl" part="lemq"/>
       </lu>
    </out>
  </action>
</rule>
```

Figure 3.41: Rule for the translation from Spanish into Catalan, which turns the verbs in simple perfect preterite tense (*dije*) into the compound perfect preterite tense usual in Catalan (*vaig dir*), and at the same time assigns the appropriate case state to the two resulting words.

```
<rule>
  <pattern>
    <pattern-item n="det"/>
    <pattern-item n="nom"/>
  </pattern>
  <action>
      <choose>
        <when>
          <test>
            <and>
              <not>
                <equal>
                  <clip pos="2" side="tl" part="gen"/>
                  <clip pos="2" side="sl" part="gen"/>
                </equal>
              </not>
              <not>
                <equal>
                  <clip pos="2" side="tl" part="gen"/>
                  <lit-tag v="mf"/>
                </equal>
              </not>
              <not>
                <equal>
                  <clip pos="2" side="tl" part="gen"/>
                  <lit-tag v="GD"/>
                </equal>
              </not>
            </and>
          </test>
          <let>
            <clip pos="1" side="tl" part="gen"/>
            <clip pos="2" side="tl" part="gen"/>
          </let>
        </when>
      </choose>
      <!-- Other gender and number agreement actions -->
```

Figure 3.42: Extract from a rule for the pattern `determiner--noun` (continues in Fig. 3.45): in this part of the rule, the gender of the noun is assigned to the determiner in the case that the gender of the noun changes from the SL (`sl`) to the TL (`tl`) during the lexical transfer process between both languages.

```
<let>
  <var n="palabra"/>
    <concat>
       <clip pos="3" side="tl" part="lem"/>
       <lit-tag v="adj"/>
    </concat>
</let>
```

Figure 3.43: In this example, the variable `palabra` is assigned the value of the concatenation of a `clip` (the lemma in position 3) and the *adj* tag.

```
<append n="temporal">
  <clip pos="3" part="gen" side="tl"/>
</append>
```

Figure 3.44: In this example, the variable `temporal` is assigned the value of the gender, in the TL, of the third word matched by the rule.

The mandatory attribute `n` specifies the name of the variable used. After applying the instruction, the previous content of the referred variable will be the prefix of the new content, that is, the new content inserted in the <**append**> will be concatenated to the pre-existing content of the variable specified in `n`.

The content of this instruction can be one or more of the following tags: <**b**>, <**clip**>, <**lit**>, <**lit-tag**>, <**var**>, <**get-case-from**>, <**case-of**> or <**concat**>. There is an example of its use in Figure 3.44.

3.5.4.40 Element for output <out>

The output instruction is used to specify the lexical forms that are sent at the output of the module after having been applied the required structural transfer operations. Its use is different according to the module. On the one hand, its use in the `chunker` module when it runs as only module (shallow-transfer) and its use in the `postchunk` module are similar, since in both cases, the output must be the input for the generator. The `chunker` in Apertium 2 and the `interchunk` have different use modes: the former to create the chunks, and the latter to modify the chunks without modifying its internal part.

1. **Use in `chunker` in shallow-transfer mode, and in `postchunk`**

```
    <!-- ... -->
    <out>
      <lu>
        <clip pos="1" side="tl" part="whole"/>
      </lu>
      <lu>
        <clip pos="2" side="tl" part="whole"/>
      </lu>
    </out>
  </process>
 </action>
</rule>
```

Figure 3.45: Extract from a rule (comes from Fig. 3.42). At the end of the rule, and after different actions, the resulting data are sent by means of the attribute `whole`, which contains the lemma and the grammatical symbols of each LF (positions 1 and 2 in the pattern).

The instruction sends each lexical form inside a <**lu**> set, which in turn can be contained inside a <**mlu**> element when the output is a multiword made of two or more LF. Besides, also the blanks or superblanks (<**b**>) between LF and LF are sent. You can find an example of its use in Figures 3.41 and 3.45.

2. **Use in `chunker` in advanced mode**

   The output of this module is expected to be a sequence of one or more chunks (sent inside a <**chunk**> element) separated by blanks <**b**>. Lexical forms and multiforms, as well as the blanks between them, are sent inside chunks. You can see in Figure 3.46 an example of use.

3. **Use in `interchunk`**

   In this module, lexical forms (words) are inaccessible, since it is only possible to operate with chunks and, therefore, inside an <**out**> element you can only put <**chunk**> elements or blanks <**b**>. The information on lemma and tags specified here in a <**chunk**> element refers exclusively to the lemma (pseudolemma) and the tags of the chunk.

   An example of its use can be found in Figure 3.47.

```
<out>
  <chunk name="pr" case="caseFirstWord">
    <tags>
      <tag><lit-tag v="PREP"/></tag>
    </tags>
    <lu>
      <clip pos="1" side="tl" part="whole"/>
    </lu>
  </chunk>
  <b pos="1"/>
  <chunk name="probj" case="caseOtherWord">
    <tags>
      <tag><lit-tag v="NP"/></tag>
      <tag><lit-tag v="tn"/></tag>
      <tag><clip pos="2" side="tl" part="pers"/></tag>
      <tag><clip pos="2" side="tl" part="gen"/></tag>
      <tag><clip pos="2" side="tl" part="nbr"/></tag>
    </tags>
    <lu>
      <clip pos="2" side="tl" part="lem"/>
      <lit-tag v="prn"/>
      <lit-tag v="2"/>
      <clip pos="2" side="tl" part="pers"/>
      <clip pos="2" side="tl" part="gen" link-to="4"/>
      <clip pos="2" side="tl" part="nbr" link-to="5"/>
    </lu>
  </chunk>
</out>
```

Figure 3.46: Output instruction that sends two chunks separated by a blank. The printed sequence is a preposition followed by a noun phrase (”NP”). The tags that are linked from the second chunk to the outside are pronoun type (”tn”), gender and number of the noun phrase (pronoun). The <**tag**> elements are used to specify the tags of the chunk, and the value of the attributes `name` and `case` is used to specify the pseudolemma of the chunk.

```
<out>
  <b pos="1"/>
  <chunk>
    <clip pos="2" part="lem"/>
    <clip pos="2" part="tags"/>
    <clip pos="2" part="chcontent"/>
  </chunk>
</out>
```

Figure 3.47: The aim of this rule output is to discard the first chunk of the matched pattern (pronoun drop). The three <**clip**> elements have been included here for illustrative purposes, since they could have been replaced by the `part="whole"` which would group them in a single <**clip**> .

3.5.4.41 Element for lexical unit <lu>

This is the element by means of which each TLLF is sent out at the end of a rule, inside an <**out**> element. With this element, one can send the whole lexical form, using the attribute `whole` of a <**clip**>, or, if required, specify its parts separately (lemma plus tags, indicated by means of <**clip**> strings, literal strings <**lit**>, tags <**lit-tag**>, variables <**var**>, besides case information [<**get-case-from**>, <**case-of**>]).

Please note that, as has been explained before, in the case of multiwords with *split lemma* it is necessary to replace the lemma queue *after* the grammatical symbols of the inflected word (or lemma head), because the `pretransfer` module has moved the queue to put it before the grammatical symbols of the head. This replacement is done here, inside the <**lu**> element, using the values `lemh` and `lemq` of the attribute `part` in a <**clip**>. The `lemh` attribute refers to the lemma head, and `lemq` to the lemma queue. As can be seen in the example 3.41, the `lemq` part of a <**clip**> is placed after the lemma head and all the grammatical symbols that follow it. This rule would be suitable, for example, for the Spanish form *eché de menos* ("I missed"), which has to be translated into Catalan as *vaig trobar a faltar*. The attribute `a_verb` which comes after `lemh` contains the grammatical symbol that describes the verb category (*vblex*, *vbser*, *vbhaver* or *vbmod* as applicable). Therefore, the last lexical form sent by this rule, in the case of *vaig trobar a faltar*, would be, in the data stream:

```
^trobar<vblex><inf># a faltar$
```

The number sign `#` in the data stream corresponds to the <**g**> element in dictionaries, used to signal the position of the invariable part in a split

lemma multiword.

It is important to note that the attributes included in <**lu**> may be empty. So, a verb matched by the rule in Fig. 3.41 which is not a split lemma multiword, will be sent with an empty `lemq` attribute, since the verb does not have lemma queue. This way it is not necessary to define different rules for lexical forms with and without queue. You can find another example of this in page 145, where the rule for verb sends in a <**lu**> the attributes `gen` (*gender*) and `nbr` (*number*). This way, it includes participles (with gender and number) and the rest of verb forms (which will have these attributes empty).

In the same page you can see a rule for a verb followed by an enclitic pronoun. Here, the lemma queue is placed after the enclitic pronoun; so, for a split lemma multiword joined to an enclitic pronoun, such as *echándote de menos*, the output in the data stream would be, when translating into Catalan:

```
^trobar<vblex><ger>+et<prn><enc><p2><mf><sg># a faltar$
```

Of course, this rule works also for verbs which do not belong to this multiword type; so, the form *explicándote* ("explaining to you") would be output, when translating from Spanish to Catalan:

```
^explicar<vblex><ger>+et<prn><enc><p2><mf><sg>$
```

As for the attribute `whole` of a <**clip**>, it must be taken into account that it can be used to send the whole lexical form only in the case that the sent word can not be a multiword, that is, can not contain a split lemma. Compare figures 3.41 and 3.45. The `whole` attribute can be used in the second example because it contains the lemma `lem` plus all the morphological tags of the lexical forms in position 1 and 2 (determiner and noun). Contrarily, in the first example, the lexical form in <**lu**> is sent in parts, with a `lemh` (lemma head) and a `lemq` (lemma queue), since it may occur that the verb matched in the pattern is a multiword with split lemma. In practice, in our system this means that the `whole` attribute can be used to send any kind of lexical form except verbs and nouns, because we defined multiwords with inner inflection only for verbs and nouns.

3.5.4.42 Element for lexical unit <mlu>

Its name derives from *multilexical unit*; it is used inside the <**out**> element to output multiwords consisting of more than one lexical form. Each lexical form in a <**mlu**> is sent inside a <**lu**> element. On the output of

the module, lexical forms contained in this element will be joined to each other by the symbol '+' in the data stream. This means that they will become a multiword made of different lexical forms, which will be treated as a single unit by the subsequent modules; therefore, the generation dictionary will have to contain an entry for this multiword in order for it to be generated.

In our system, this element is used to join enclitic pronouns to conjugated verbs.

3.5.4.43 Element for chunk encapsulation <chunk>

This is the element in which chunks are sent, in an <**out**> element, on the output of the module. It is only used in the `chunker` module in advanced mode, and in the `interchunk` module. It is not used in the `postchunk` module because its output does not contain any chunk. Neither it is used in the `chunker` module in shallow-transfer mode, because its output does not contain chunks but individual lexical units and blanks.

1. **Use in `chunker` in advanced mode**

   In this mode, the <**chunk**> element must have an attribute `name`, which is the lemma of the chunk, or an attribute `namefrom` which refers to a variable previously defined, whose value will be used as the lemma of the chunk. Besides, it can include the attribute `case` to specify which variable is the case policy taken from (for example, a value obtained with the instruction <**case-from**>).

   An example of its use can be found in Figure 3.46.

2. **Use in `interchunk`**

   In this module, the <**chunk**> element does not specify any attribute; it is used just as the <**lu**> element is used in the shallow-transfer or in the `postchunk` to delimit the lexical forms. The elements it sends are (generally in a <**clip**> instruction): the lemma of the chunk (`lem`), its tags (`tags`) and the chunk content (`chcontent`, contains LF plus blanks), which is an invariable part since it can not be accessed from the `interchunk` module. The invariable part of the chunk is sent at the end. You can also use the `whole` attribute to send the whole chunk (lemma, tags and invariable content).

   An example of its use can be found in Figure 3.47.

3.5.4.44 Element for tag links section <tags>

Only in chunker in advanced mode.

This element is used to specify a list of tags, or <**tag**> elements, which will become the pseudotags of the chunk. It does not have attributes, and must be included as first item inside the <**chunk**> element. See Figure 3.46.

3.5.4.45 Element for tag link <tag>

Only in chunker in advanced mode.

The <**tag**> element must contain a morphological tag, which can be specified by means of a <**clip**> instruction or a literal tag <**lit-tag**>. It does not have attributes.

The tag or tags specified this way in a chunk will become the grammatical symbols of the chunk; the next module, `interchunk`, will be able to use them to test and modify the characteristics of the chunks.

3.5.4.46 Element for blank <b>

The <**b**> element refers to [super]blanks and is indexed by the attribute `pos`. For example, a <**b**> with `pos="2"` refers to the [super]blanks (including format data encapsulated by the de-formatter) between the 2nd SLLF and the 3rd SLLF. The explicit management of [super]blanks enables the correct placement of format when the result of the structural transfer has more or less elements than the original, or when it has been reordered in some way.

3.5.5 Specification of the three modules that build an advanced transfer system

In the following lines we describe the differences between the rule format in the three modules of an advanced transfer system. When Apertium works as a shallow-transfer system, the only module to be run is the first one, called `chunker`, which communicates directly with the generation module.

3.5.5.1 `Chunker` module

This module can be used alone as a shallow-transfer system, or in combination with the other two transfer modules to build an advanced transfer

system. An attribute of the <transfer> element controls its run mode.

Input/output

- Input: data in the pretransfer output format, that is, with invariable queues of multiwords moved to the position right before the first grammatical symbol.
- Output:
  - in advanced mode (in an advanced transfer system): chunks, that will be detected and processed by the next module
  - in shallow-transfer mode (in a shallow-transfer system): lexical forms, that will be the input of the generation module.

Data files

This program uses a single configuration file and a precompiled file for pattern detection calculated from the former. The name of the pattern file (the configuration file) will have the extension .t1x. Since the chunker is the program that looks up the bilingual dictionary, this dictionary (compiled) also has to be provided to the program.

The DTD of this data file is specified in Appendix A.3, and the elements used to create the rules in the file are described in Section 3.5.4.

Pattern matching

The rule matching system in this module will be the one described in 3.5.2, since it is the same in advanced transfer mode and in shallow-transfer mode. The apertium-pretransfer program is needed to adapt the tagger output format to the input format required by the transfer module. There is the possibility that, in later versions of Apertium, the *part-of-speech tagger* is modified so that it does the work of apertium-pretransfer.

How it works

The module works similarly in shallow-transfer mode and in advanced mode, with these differences:

- If we want that the module works as the first module in an advanced transfer system, we must specify the value chunk in the optional attribute default of the root element <transfer>. The de-

fault value is `lu`, which implies that the `chunker` works in shallow-transfer mode (as a single module).

- Chunk generation in the output: the `<chunk>` tag is an element one level higher than `<lu>` (*lexical unit*), which generates chunks with the characteristics described in 3.5.3.1; it has the following attributes:
  - `name` (optional): pseudolemma of the chunk. It contains a string that is identified as the pseudolemma of the chunk.
  - `namefrom` (optional): pseudolemma of the chunk, obtained from a variable. It is compulsory to specify whether `name` or `namefrom`.
  - `case` (optional): variable that is used to obtain the information on case from it and apply it to the lemma specified in `name` or in `namefrom`.
- Each chunk begins with a `<tags>` instruction, which does not allow any attribute, and which contains one or more individual instructions `<tag>`.
- Instructions `<tag>` do not have attributes. They can include any instruction that returns a string as a value: `<lit>`, `<var>` .
- Instructions `<clip>` have an optional attribute: `link-to`, which is used to specify a tag *<value of link-to>* that replaces the information specified by the `<clip>` in the rest of its attributes. This information is dispensable but can be useful as information on the origin of the linguistic decision.

The following is a use example of the `<chunk>` element :

```
<out>
  <chunk name="adj-noun" case="variableCase">
    <tags>
      <tag><lit-tag v="NP"/></tag>
      <tag><clip pos="2" side="tl" part="gen"/></tag>
      <tag><clip pos="2" side="tl" part="nbr"/></tag>
    </tags>
    <lu>
      <clip pos="2" side="tl" part="lemh"/>
      <clip pos="2" side="tl" part="a_noun"/>
      <clip pos="2" side="tl" part="gen" link-to="2"/>
      <clip pos="2" side="tl" part="nbr" link-to="3"/>
```

```
    </lu>
    <b pos="1"/>
    <lu>
      <var n="adjectiu"/>
      <clip pos="1" side="tl" part="lem"/>
      <clip pos="1" side="tl" part="a_adj"/>
      <clip pos="2" side="tl" part="gen" link-to="2"/>
      <clip pos="2" side="tl" part="nbr" link-to="3"/>
    </lu>
  </chunk>
</out>
```

Default action

Isolated *superblanks* which are not detected by any pattern in this module, are written in the same order in which they arrive.

The default action for words not matched by any pattern is different depending on the transfer mode (that is, on the value of the optional attribute `default` of the root element `<transfer>`):

- if the value is `chunk` (i.e. the module works in advanced mode): it will generate trivial chunks with the words not matched by any rule, so that in the output there are no words not included in a chunk. The new chunk will be created with the translation of the word by the bilingual dictionary. The fixed lemma of these implicitly created chunks is `default`.
- if the value is `lu` (default value; i.e. the module works as single module in a shallow-transfer system): it will not create chunks for words not matched by rules, they will just be translated using the bilingual dictionary.

The following is an automatically generated chunk for a lexical form not matched by any rule in the `chunker` module when the `default` attribute has the value `chunk`:

```
^default{^that<cnjsub>$}$
```

3.5.5.2 `Interchunk` module

The `interchunk` module processes chunks; it may reorder them and change its morphosyntactic information. This is done by detecting patterns of chunks (sequences of chunks). The instructions that control how it

works are, with little differences, the same used by the `chunker` module; they are written, however, in a different file. Chunks are processed here in a similar way as words are processed in the `chunker` of Apertium.

Input/output

- Input: chunks from the `chunker`.
- Output: chunks possibly reordered and with the data on its pseudolemmas (lexical pseudoforms) possibly changed.

Data files

This module uses two data files. A specification file of the `interchunk` program, with extension `.t2x` by analogy with the previous module, and a file of precalculated patterns to accelerate the analysis of the input. The binary file of the bilingual dictionary is not included because it is not used.

The syntax of the specification file is very similar to that of the `chunker`. Its DTD is specified in Appendix A.4, and the elements used to create the rules in the file are described in Section 3.5.4.

Pattern matching

Rules detect patterns defined by sequences of lexical pseudoforms. These lexical pseudoforms have a format based on the format of lexical forms for words. In practice, a lexical pseudoform is seen equivalently as lexical forms are seen in the `chunker` regarding pattern matching. This way, pattern matching will be based on attributes defined for lexical pseudoforms, not for lexical forms (words) of the original pattern.

How it works

With regard to the set of instructions used in `chunker`, the changes on the set of instructions for this module are the following:

- The root element is called `<interchunk>` and does not have any attribute.
- The attribute `side` disappears: This module does not use bilingual dictionaries; therefore, the attribute used to indicate whether the consulted side is SL or TL looses sense. This attribute was basically used in the `<out>` instructions.

- The <chunk> tag is used here without attributes, simply inside an <out> to delimit the output of chunks.
- The predefined attribute lem refers to the pseudolemma of the chunk. In the same way, the predefined attribute tags refers to the grammatical symbols or tags of the chunk. The chunk content becomes something like a queue which can be printed with the implicit attribute chcontent.
- All the values of part, except chname, access the pseudolemma and the tags of the chunk (not of individual words).
- Unlike what happens in the chunker module, in the rules of this module it is not allowed to print anything else than <chunk>s in the <out> instructions, in no case isolated words.

Default action

Like in the previous module, a default action has been defined, which writes without modifications the chunks not matched by any pattern of the specification file. This default action writes exactly what it reads, be it chunks or blanks.

3.5.5.3 `Postchunk` module

The postchunk module detects single chunks and, for each of them, performs the specified actions. Detection is based on the lemma of the chunk, and not in patterns (not in tags); this causes detection in this module to be done specific for each "name" of chunk.

On the other hand, detection and processing in rules is based on the fact that references to parameters are solved right after detection, that is, the tags <1>, <2>, etc. are automatically replaced by the value of the parameters before the processing begins. Positions (attribute pos) specified in instructions such as <clip>, refer to the position of the words inside the chunk.

Also the case policy is automatically applied (see Section 3.5.3.2) from the pseudolemma of the chunk to the words inside the chunk.

Input/output

- Input: chunks from the interchunk.
- Output: valid input for the morphological generator of Apertium.

Data files

This program has its own specification file, which will have the extension `.t3x`. Its syntax is based as well on the `chunker` and the `interchunk`.

Pattern matching

Chunk matching is based on the name of the chunk. Unmatched chunks receive the default processing.

How it works

The differences with regard to the `interchunk` module are the following:

- It is not allowed to write chunks (`<chunk>`) in the output: only lexical units (`<lu>` or `<mlu>`) and blanks can be written.
- New detection attribute `name` in `<cat-item>`, which is used in the `<pattern>` part of rules isolatedly, to force pattern detection basing on its name.
- Also the attribute `side` is not used here, as in the `interchunk`, for the same reason: the bilingual dictionary is not looked up.

Default action

In this module, the default action is to write the words contained in the chunks, replacing the references with the parameters of the chunk. It will be applied to most chunks, since it is foreseen that this module performs non-default actions only for specific cases requiring some special processing.

Also the case policy is applied by default (see Section 3.5.3.2).

In any case, blanks outside chunks are copied in the same order as are read, since chunk matching is done individually (this module does not group chunks).

3.5.6 Preprocessing of the structural transfer module

Specification files for the structural transfer modules, also called *transfer rules files*, are pre-processed by the program *apertium-preprocess-transfer*, which calculates the patterns to match rules preconditions, and indexes

the rules to speed up its processsing during execution time. This information is saved in a binary file which is read together with the bilingual dictionary and the rules file itself, because the structural transfer and lexical transfer modules are executed together.

3.6 De-formatter and re-formatter

3.6.1 Format processing

This section describes how the de-formatter and re-formatter process the format of the documents. These two modules are created from a set of format specification rules in XML, which are described in Section 3.6.2.

Apertium can process documents in XML, HTML, RTF and plain text. For all these document types, format is *encapsulated* as explained in the following lines.

Text strings that are identified as part of the format —from now on referred to as *blocks of format* or *superblanks*— are encapsulated between delimiters that depend on the specification of the data flow between modules (which is described in detail in Section 2); so, in the flow format (sections 2.2.1 and 2.3), *superblanks* are put between brackets '[' and ']'. Each of these encapsulated strings will be treated as it were a blank <**b**/> (page 34) —that is why they are called *superblanks*— and will be restored in the correct order in the translator's output.

As has been explained in Section 2, when the blocks of format are large (as is sometimes the case in HTML with Javascript code fragments, or in RTF with bitmap images), these blocks will be saved as temporary files so that they can be removed from the data flow of the translation.

Sometimes, the format in a document can implicitly indicate the division of the text into sentences (see page 13 in Section 2). For example, section or document titles can be a sentence without full stop. If we know that a format mark is indicating this division, we have to take advantage of this information in order to do a better translation. Some examples of format that give us data about the end of a sentence are: two consecutive line breaks in plain text format, a </h1> tag in HTML, etc. The de-formatter generates in such cases a mark of sentence end that is equivalent to a full stop.

3.6.1.1 Format encapsulation method

The types of blocks of format or *superblanks* that can be generated as a result of the format processing are the following:

- *Non-empty blocks of format or superblanks*. They contain exclusively format marks of the source document. In the data flow described in Section 2 , they begin with a left square bracket '[' and end with a right square bracket ']'.

- *Blocks of format with reference to an external file* or *extensive superblanks*. They encapsulate long format fragments in a way that improves the translator's performance. In the data flow described in Section 2, they begin with the characters '[@', then there is the name of the file where the format fragment extracted from the source text is saved, and finally they end with a right square bracket ']'.

- *Empty blocks of format*. They contain artificial information on text division obtained from the format data. Before the empty block of format, the system places the appropriate artificial punctuation mark. When the original format is restored in the document at the end of the process, the presence of a block of format like this will cause the deletion of the character right before the block in the data flow.

The general criteria applied to the creation of blocks of format are the following:

- Everything that is considered not to be part of the text to be translated, has to be encapsulated in blocks of format.

- There can not be two or more strictly consecutive non-empty blocks of format. Two consecutive blocks of format must be merged into a single block.

- Empty blocks of format must precede a non-empty block of format or the end of the file.

Figure 3.48 shows an example document the format of which must be processed before translation; the encapsulation corresponds to the flow format not based on XML. Figure 3.49 displays the result of processing the mentioned document.

We would like to emphasize the following from this example:

```
<html>
<head>
<title>This is the title</title>
<script>
<!-- ... (an extensive code block) -->
</script>
</head>
<body>
<p>This
is a paragraph in two lines</p>
</body>
</html>
```

Figure 3.48: Example of HTML document

```
[<html>
<head>
<title>]This is the title.[][@/tmp/temp35345]This[
]is a paragraph in two lines.[][</p>
</body>
</html>]
```

Figure 3.49: Example of HTML document where the blocks of format have been encapsulated by the de-formatter

- The system does not generate consecutive blocks of format with content (non-empty).
- Tags like </**title**> or </**p**> cause the insertion of an artificial punctuation mark; this insertion is done systematically, even when it is not necessary, because it does not interfere and is efficient.
- Extensive superblanks are literally removed from the translation process. In this case, the temporary file `temp35345` contains the tags from </**title**> to <**p**>
- Simple blanks between words are not encapsulated. But the system does encapsulate multiple blanks (two or more consecutive blanks), tabs, etc. Also line breaks are encapsulated.

3.6.2 Data: format specification rules

This section describes how the de-formatter and re-formatter are generated from a format specification in XML.

Rules for format, like linguistic data, are specified in XML, and they contain regular expressions with `flex` syntax. The specification is divided in three parts (see its DTD in the Appendix A.6):

- **Configuration options**. Here one specifies the value for the maximum length of a non-extensive superblank, the input and output encodings, whether case must be considered, and the regular expressions for escape characters and space characters.
- **Format rules**. Describes the set of tags belonging to a specific format which have to be included in a block of format by the de-formatter. These tags may, optionally, indicate a sentence end, in which case the de-formatter will insert an artificial punctuation mark (followed by an empty block of format, as explained in the previous section). One has to specify the priority of application of the rules, although, when this is not relevant, it is possible to give the same priority to all the rules by assigning them the same value (any number).

  Everything that is not specified as format will be left without encapsulation and, therefore, will be considered as translatable text.
- **Replacement rules**. Allow to replace special characters in the text. A regular expression will match a set of special characters, and will replace it with the specified characters. For example, in HTML, the

characters specified in hexadecimal have to be replaced with the corresponding entity or ASCII character. For example, `cami&oacute;n` corresponds to `camión`.

Rules are described in more detail next.

- Root of the specification file. The attribute `name` contains the name of the format.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<format name="html">
  <options>
  ...
  </options>

  <rules>
  ...
  </rules>
</format>
```

It has to include the options and rules, an example of which is presented next:

- Options.

```
<options>
  <largeblocks size="8192"/>
  <input encoding="ISO-8859-1"/>
  <output encoding="ISO-8859-1"/>
  <escape-chars regexp='[\[\]^$\\]'/>
  <space-chars regexp='[ \n\t\r]'/>
  <case-sensitive value="no"/>
</options>
```

The element `<largeblocks>` specifies the maximum length of a non-extensive superblank, through the value of the attribute `size`. The elements `<input>` and `<output>` specify the input and output encoding of the text, through the attribute `encoding`.

The element `escape-chars` specifies, by means of a regular expression declared in the value of the attribute `regexp`, which characters must be escaped with a backslash. The element `<space-chars>` specifies the set of characters that must be considered as blanks.

Finally, the element `case-sensitive` specifies if case is relevant in the specifications of format attributes in which regular expressions are contained.

- Rules. There are format rules and replacement rules.

```
<rules>
  <format-rule ... >
    ...
  </format-rule>
  ...

  <replacement-rule>
    ...
  </replacement-rule>
  ...
</rules>
```

The two types are described in the following points.

- Format rules. The de-formatter will encapsulate in blocks of format the tags indicated by these rules in the field `regexp`. If they are begin and end tags, and everything delimited by them is format, one has to specify a `regexp` both for `begin` and for `end`:

```
<format-rule eos="no" priority="1">
  <begin regexp='"\&lt;!--"'/>
  <end regexp='"--\&gt;"'/>
</format-rule>
```

Otherwise only one `begin-end` element is used:

```
<format-rule eos="yes" priority="3">
  <begin-end regexp='"&lt;"[/]?"li"[^&gt;]*"&gt;"'/>
</format-rule>
```

Besides, in `priority` you have to specify a priority to tell the system in which order the format rules must be applied (the absolute value is not relevant, only the order resulting from the values). In "`eos`" you indicate, with `yes` or `no`, whether the block of format that contains the detected pattern must be preceded by an artificial punctuation mark or not.[11]

[11] In all these cases, the typical entities `<` and `>` are used to represent the characters < and > respectively.

- Replacement rules. Are used to replace special characters in the text. The regular expression in the attribute `regexp` will recognize a set of special characters and will replace them with the specified characters in the text to be translated. The correspondence between original and replacement characters is stated in the attributes `source` and `target` of the `replace` elements, which can be multiple:

```
<replacement-rule regexp='"&amp;"[^;]+;'>
  <replace source="&amp;Agrave;" target="À"/>
  <replace source="&amp;#192;" target="À"/>
  <replace source="&amp;#xC0;" target="À"/>
  <replace source="&amp;#xc0;" target="À"/>
  <replace source="&amp;Aacute;" target="Á"/>
  <replace source="&amp;#193;" target="Á"/>
  <replace source="&amp;#xC1;" target="Á"/>
  <replace source="&amp;#xc1;" target="Á"/>
  ...
</replacement-rule>
```

- Regular expressions of `regexp` attributes. They have the syntax used in `flex` [9].

As example of a format specification, we will give that for HTML. The explanation given in the following paragraphs can be followed looking at Figure 3.50.

In the first place, we find the format rule that specifies in a general way all the HTML tags: it considers as HTML tag everything that begins with the sign **<** and ends with the sign **>**. This rule has the lowest priority (4) so that the more specific rules are applied preferentially. But before considering a tag in a general way by applying this rule, some of the higher priority rules will be applied. In the case of HTML, the highest priority is for comments `<!-- ... -->`. The marks for beginning and end `<script> </script>` and `<style> </style>`, where everything included by them is considered to be format, has priority 2. Priority 3 is for tags that indicate end of sentence (artificial punctuation), which are `</br>`, `</hr>`, `</p>`, etc.

Last of all are the replacement rules, which replace all the codes that begin with `&`, as specified in the regular expression. Then, each one of the replacements is defined: `&Agrave`, as well as `&#192`, `&#xC0` and `&#xc0` are replaced with `À`. The remaining special characters are declared in the same way.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<format name="html">
  <options>
    <largeblocks size="8192"/>
    <input encoding="ISO-8859-1"/>
    <output encoding="ISO-8859-1"/>
    <escape-chars regexp='[\[\]^$\\]'/>
    <space-chars regexp='[ \ n\ t\ r]'/>
    <case-sensitive value="no"/>
  </options>

  <rules>
   <format-rule eos="no" priority="1">
      <begin regexp='"&lt;!--"'/>
     <end regexp='"--&gt;"'/>
   </format-rule>

   <format-rule eos="no" priority="2">
     <begin regexp='"&lt;script"[^&gt;]*"&gt;"'/>
     <end regexp='"&lt;/script"[^&gt;]*"&gt;"'/>
   </format-rule>
   <format-rule eos="no" priority="2">
     <begin regexp='"&lt;style"[^&gt;]*"&gt;"'/>
     <end regexp='"&lt;/style"[^&gt;]*"&gt;"'/>
   </format-rule>

   <format-rule eos="yes" priority="3">
     <begin-end regexp='"&lt;"[/]?"br"[^&gt;]*"&gt;"'/>
   </format-rule>
   <!-- Here come more declarations of format-rule eos="yes"-->
   <!-- ...                                                  -->

   <format-rule eos="no" priority="4">
     <begin-end regexp='"&lt;"[a-zA-Z][^&gt;]*"&gt;"'/>
   </format-rule>

   <replacement-rule regexp='"&amp;"[^;]+;'>
     <replace source="&amp;Agrave;" target="À"/>
     <replace source="&amp;#192;" target="À"/>
     <replace source="&amp;#xC0;" target="À"/>
     <replace source="&amp;#xc0;" target="À"/>
     <!-- Here come more replace elements               -->
     <!-- ...                                             -->
   </replacement-rule>
  </rules>
</format>
```

Figure 3.50: Part of the rules definition for HTML format

3.6.3 Generation of de-formatters and re-formatters

To generate the de-formatter and re-formatter for a given format, the XML rules that declare the format are applied a style sheet that carries out the generation. This XSLT transformation produces a `lex` [9] file that, once compiled, is the executable of the de-formatter and the re-formatter for the specified format.

Thanks to the general specification of formats described in this chapter, it has been possible to define specifications for HTML, RTF and plain text. These specifications are in the `apertium` package, in the respective files `html-format.xml`, `rtf-format.xml`, `txt-format.xml`. In particular, it is quite simple to define de-formatters and re-formatters for any XML format.

Chapter 4

Installing and running the system

4.1 System requirements

The system where you want to install and run Apertium must have the following programs installed:

- `libxml2` version 2.6.17 or later (on Ubuntu you may need to install `libxml2-dev` too)
- `xmllint` tool (usually comes with `libxml2`, but may be an independent package on your system, i.e. Debian GNU-Linux)
- `xsltproc` tool (non-PowerPC users); also comes with `libxml2` but may also be an independent package in your system, as happens with the `xmllint` tool
- `sabcmd` tool (PowerPC users), provided by package `sablotron`
- flex 2.5.4 or earlier (in some distributions, flex-old package)
- GNU `make`, `gcc` (`g++`), `bash` shell

4.2 Installing program packages

To install the Apertium machine translation system programs and libraries first you need to download (from `http://sourceforge.net/projects/apertium`), compile and install the latest version of the following packages, in the specified order:

1. `lttoolbox`

2. `apertium`

The simplest way to compile each package is:

1. Go to the directory containing the package's source code and type `./configure` to configure the package for your system. If you're using csh on an old version of System V, you might need to type `sh ./configure` instead to prevent `csh` (the default shell in old System V) from trying to execute `configure` itself. Running `configure` takes a while. While running, it prints some messages telling which features it is checking for.

2. Type `make` to compile the package

3. Type `make install` (possibly with root privileges) to install the programs and any data files and documentation.

4. You can remove the program binaries and object files from the source code directory by typing `make clean`. To remove also the files that `configure` created (so you can compile the package for a different kind of computer), type `make distclean`. There is also a `maintainer-clean` option in the Makefile, but that is intended mainly for the package's developers. If you use it, you may have to get all sorts of other programs in order to regenerate files that came with the distribution.

If you don't have root privileges to install the programs in your system, you can use the `-prefix` flag with the configure script to install them at your user account. For example:

```
$ pwd
/home/me/lttoolbox-0.9.1
$ ./configure --prefix=/home/me/myinstall
```

Libraries will be installed in the `LIBDIR=$prefix/lib` directory. If no `-prefix` flag is specified with configure script, LIBDIR will be `/usr/local/lib`.

If you find some error to link against installed libraries in a given directory `LIBDIR`, you must either use libtool, and specify the full pathname of the library, or use the `LIBDIR` flag during linking and do at least one of the following:

- add `LIBDIR` to the `LD_LIBRARY_PATH` environment variable during execution

- add `LIBDIR` to the `LD_RUN_PATH` environment variable during linking
- use the `-Wl,--rpath -Wl,LIBDIR` linker flag
- have your system administrator add `LIBDIR` to `/etc/ld.so.conf` and run `ldconfig`

See any operating system documentation about shared libraries for more information, such as the `ld(1)` and `ld.so(8)` manual pages.

4.3 Installing data packages

To install the linguistic data packages, follow these steps:

1. Download a data package (`apertium-`$LANG_1$`-`$LANG_2$`-`$VERSION$`.tar.gz`) from Apertium's website in Sourceforge (`http://apertium.sourceforge.net/`). For example, to get version 0.9 of the linguistic data for the Spanish–Catalan translator, you need to download the package `apertium-es-ca-0.9.tar.gz`.
2. Unpack the tarball in any directory, go to this directory and type `make` in the terminal. Wait while linguistic data are compiled.

4.4 Using the translator

There are Apertium versions that work both in Linux systems (always more up-to-date) and in Windows systems. The information in this section is intended for Linux users.

To run the translator, you have to use the `apertium` tool referring to the directory where linguistic data are saved, and specifying the translation direction (`es-ca`, `ca-es`, `es-gl`, etc.), the file format (`txt`, `html`, `rtf`), the name of the file to be translated and the name of the output file. So, the command structure is as follows:

```
$ apertium -d <directory> <translation> <format>

                              < input_file > output_file
```

For example, if your directory is `/home/maria/apertium-es-ca`, you have to type the following to translate a file in `txt` format from Spanish to Catalan:

```
$ apertium -d /home/maria/apertium-es-ca es-ca
txt <file_sp >file_ca
```

It is recommended to go to the directory where linguistic data are saved, because this way you only need to type a dot to refer to the current directory:

```
$ apertium -d . es-ca txt <file_sp >file_ca
```

If no format is specified, the default format is `txt`. When working with the `txt`, `html` and `rtf` formats, unknown words are marked with an asterisk (*) and errors with a symbol (@, # or /); if you wish that neither unknown words nor errors are marked, you have to add a `u` to the format name. Therefore, the format options are the following:

- `txt` : Default option, text with marks for unknown words and errors
- `txtu` : text without marks for unknown words and errors
- `html` : HTML with marks for unknown words and errors
- `htmlu` : HTML without marks for unknown words and errors
- `rtf` : RTF with marks for unknown words and errors
- `rtfu` : RTF without marks for unknown words and errors

If you do not wish to translate a file but just a sentence or a paragraph in the screen, you can run the `apertium` tool without specifying any file name. The command, if you are in the directory where linguistic data are saved, would be the following:

```
$ apertium -d . es-ca
```

Then, you have to type or paste the text you wish to translate (it can contain line breaks). To get the translated version, press Ctrl + D. The translation will be displayed on the screen.

A third way of translating with Apertium is using the `echo` command to send text through the translator:

```
$ echo "text to be translated" | apertium . es-ca
```

Chapter 5

Maintaining linguistic data

5.1 Description of linguistic data currently available

At present, Apertium has linguistic data for three language pairs : Spanish–Catalan and Spanish–Galician. The files containing the linguistic data are saved in a single directory: `apertium-es-ca` for the pair Spanish–Catalan and `apertium-es-gl` for the pair Spanish–Galician. The names of the files in this directory have the following structure:

- `apertium-PAIR.LANG.dix` : monolingual dictionary for LANG.
- `apertium-PAIR.LANG1-LANG2.dix` : `LANG1-LANG2` bilingual dictionary.
- `apertium-PAIR.trules-LANG1-LANG2.xml` : structural transfer rules for the translation from `LANG1` to `LANG2` .
- `apertium-PAIR.LANG.tsx` : tagger definition file for `LANG`.
- `apertium-PAIR.post-LANG.dix` : Post-generation dictionary for `LANG` (applies when translating into `LANG`).
- directory `LANG-tagger-data` : contains data needed for the `LANG` tagger (corpora, etc.)

`apertium-PAIR` refers to the linguistic combination of the translator. Its two possible values at the moment are `apertium-es-ca` and `apertium-es-gl`. According to this structure, the Catalan monolingual dictionary is called `apertium-es-ca.ca.dix`, the Spanish–Galician bilingual dictionary is called `apertium-es-gl.es-gl.dix` and the structural transfer rules file for the translation from Catalan into Spanish is called `apertium-es-ca.trules-ca-es.xml`.

The linguistic data available (by January 2006) for the different language pairs are summarized in the following table.

Translator Apertium-es-ca	
Spanish monolingual dictionary	11.800 entries
Catalan monolingual dictionary	11.800 entries
Spanish–Catalan bilingual dictionary	12.800 entries (correspondences `es-ca`)
Structural transfer rules from Spanish into Catalan	44 rules
Structural transfer rules from Catalan into Spanish	58 rules
Spanish post-generation dictionary	25 entries and 5 paradigms
Catalan post-generation dictionary	16 entries and 57 paradigms
Translator Apertium-es-gl	
Spanish monolingual dictionary	9.000 entries
Galician monolingual dictionary	8.600 entries
Spanish–Galician bilingual dictionary	8.500 entries (correspondences `es-gl`)
Structural transfer rules from Spanish into Galician	46 rules
Structural transfer rules from Galician into Spanish	38 rules
Spanish post-generation dictionary	36 entries and 12 paradigms
Galician post-generation dictionary	74 entries and 48 paradigms

5.2 Adding words to monolingual and bilingual dictionaries

When extending or adapting Apertium, the most likely operation that will be performed will be to extend its dictionaries. In fact, it will be far more common than adding transfer or post-generation rules.

We describe next the most important things one has to take into account when adding new words to the translator. This information is more general than the data provided in the section describing dictionaries (chapter 3.1.2), although we give here some practical information that might be very useful to the users who decide to make changes in the translator.

IMPORTANT: Every time a set of modifications is made to any of the dictionaries, the modules have to be recompiled. Type *make* in the directory where the linguistic data are saved (apertium-es-ca, apertium-es-gl or what may be applicable) so that the system generates the new binary files.

If you want to add a new word to Apertium, you need to add three entries in the dictionaries. Suppose you are working with the Spanish-Catalan pair. In this case, you have to add:

1. an entry in the Spanish monolingual dictionary: so that the translator can analyze ("understand") the word when it finds it in a text, and generate it when translating this word into Spanish.
2. an entry in the bilingual dictionary: so that you can tell Apertium how to translate this word from one language to the other.
3. an entry in the Catalan monolingual dictionary: so that the translator can analyze ("understand") the word when it finds it in a text, and generate it when translating this word into Catalan.

You will need to go to the directory containing the XML dictionaries (for the Spanish-Catalan pair, this is `apertium-es-ca`) and open with a text editor or a specialized XML editor the three dictionary files mentioned: `apertium-es-ca.es.dix`, `apertium-es-ca.es-ca.dix` and `apertium-es-ca.ca.dix`. The entries you need to create in these three dictionaries share a common structure.

Monolingual dictionary (Spanish)

You may want, for example, to add the Spanish adjective *cósmico*, whose equivalent in Catalan is *còsmic*. The first step is to add this word to the Spanish monolingual dictionary.

You will see that a monolingual dictionary has basically two types of data: **paradigms** (in the "`<pardefs>`" section of the dictionary, each paradigm inside a `<pardef>` element) and **word entries** (in the main (`<section>` of the dictionary, each one inside an `<e>` element). Word entries consist of a lemma (that is, the word as you would find it in a typical paper dictionary) plus grammatical information; paradigms contain the inflection data of all lemmas in the dictionary. You can search a particular word by searching the string `lm="word"` (`lm` meaning *lemma*). Bear in mind, however, that the element `lm` is optional and some other dictionaries may not contain it.

Look at the word entries in the Spanish monolingual dictionary, for example at the entry for the adjective *bonito*. You can find it by searching `lm="bonito"`:

```
<e lm="bonito">
  <i>bonit</i>
```

```
  <par n="absolut/o__adj"/>
</e>
```

To add a word, you will have to create an entry with the same structure. The part between and contains the prefix of the word that is common to all inflected forms, and the element <par> refers to the inflection paradigm of this word. Therefore, this entry means that the adjective *bonito* inflects like the adjective *absoluto* and has the same morphological analysis: the forms *bonit**o***, *bonit**a***, *bonit**os***, *bonit**as*** are equivalent to the forms *absolut**o***, *absolut**a***, *absolut**os***, *absolut**as*** and have the morphological analysis: `adj m sg`, `adj f sg`, `adj m pl` and `adj f pl` respectively.

Now, you have to decide which is the lexical category of the word you want to add: the word *cósmico* is an adjective, like *bonito*. Next, you have to find the appropriate paradigm for this adjective. Is it the same as the one for *bonito* and *absoluto*? ¿Can you say *cósmic**o***, *cósmic**a***, *cósmic**os***, *cósmic**as***? The answer is yes, and, with all this information, you can now create the correct entry:

```
<e lm="cósmico">
    <i>cósmic</i>
  <par n="absolut/o__adj"/>
</e>
```

If the word you want to add has a different paradigm, you have to find it in the dictionary and assign it to the entry. You have two ways to find the appropriate paradigm: looking in the <pardefs> section of the dictionary, where all the paradigms are defined inside a <pardef> element, or finding another word that you think may already exist in the dictionary and that has the same inflection paradigm as the one to be added. For example, if you want to add the word *genoma*, you need to find an appropriate paradigm for a **noun** whose gender is masculine and forms the plural with the addition of an **-s**. This will be the paradigm "`abismo__n`" in our present dictionaries. Therefore, the entry for this new word would be:

```
<e lm="genoma">
    <i>genoma</i>
  <par n="abismo__n"/>
</e>
```

In exceptional cases you will need to create a new paradigm for a certain word. You can look at the structure of other paradigms and create

one accordingly. For a more detailed description of paradigms and word entries in the dictionaries, refer to section 3.1.2.

Monolingual dictionary (Catalan)

Once you have added the word to one monolingual dictionary, you have to do the same to the other monolingual dictionary of the translation pair (in our example, the Catalan monolingual dictionary) using the same structure. The result would be:

```
<e lm="còsmic">
    <i>còsmi</i>
  <par n="acadèmi/c__adj"/>
</e>
```

Monolingual dictionary (Galician)

In the case you are trying to improve the XML dictionaries for the Spanish-Galician pair, you will need to go to the directory `apertium-es-gl` and open with a text editor or a specialized XML editor the three dictionary files `apertium-es-gl.es.dix`, `apertium-es-gl.es-gl.dix` and `apertium-es-gl.gl.dix`. In that case, once you have added the new Spanish word *genoma* to the Spanish monolingual dictionary (`apertium-es-gl.es.dix`), you have to add the equivalent Galician word *xenoma* to the Galician monolingual dictionary (`apertium-es-gl.gl.dix`), that is:

```
<e lm="xenoma">
    <i>xenoma</i>
  <par n="Xulio__n"/>
</e>
```

Bilingual dictionary

The last step is to add the translation to the bilingual dictionary.

A bilingual dictionary does not usually have paradigms, only lemmas. An entry contains only the lemma in both languages and the first grammatical symbol (the lexical category) of each one. Entries have a left side (`<l>`) and a right side (`<r>`), and each language has always to be in the same position: in our system, it has been agreed that Spanish occupies the left side, and Catalan, Galician and Portuguese the right side.

With the addition of the lemma of both words, the system will translate all their inflected forms (the grammatical symbols are copied from the source language word to the target language word). This will only work if the source language word and the target language word are grammatically equivalent, that is, if they share exactly the same grammatical symbols for

all of their inflected forms. This is the case with our example; therefore, the entry you have to add to the bilingual dictionary is:

```
<e>
  <p>
    <l>cósmico<s n="adj"/></l>
    <r>còsmic<s n="adj"/></r>
  </p>
</e>
```

This entry will translate all the inflected forms, that is, `adj m sg`, `adj f sg`, `adj m pl` and `adj f pl`. It works for the translation in both directions: from Spanish to Catalan and from Catalan to Spanish.

In the case of the Spanish-Galician pair, the following bilingual entry in the Spanish-Galician bilingual dictionary (`apertium-es-gl.es-gl.dix`) will translate all the inflected forms for the equivalent words *genoma*/*xenoma* in both directions:

```
<e>
  <p>
    <l>genoma<s n="n"/></l>
    <r>xenoma<s n="n"/></r>
  </p>
</e>
```

What to do if the word pair is not equivalent grammatically (their grammatical symbols are not exactly the same)? In that case, you need to specify all the grammatical symbols (in the same order as they are specified in the monolingual dictionaries) until you reach the symbol that differs between the source language word and the target language word. For example, the Spanish noun *limón* has masculine gender and its equivalent in Catalan, *llimona*, has feminine gender. The entry in the bilingual dictionary must be as follows:

```
<e>
  <p>
    <l>limón<s n="n"/><s n="m"/></l>
    <r>llimona<s n="n"/><s n="f"/></r>
  </p>
</e>
```

A more difficult problem arises when two words have different grammatical symbols and the grammatical information of the source language

word is not enough to determine the gender (masculine or feminine) or the number (singular or plural) of the target language word. Take for example the Spanish adjective *canadiense*. Its gender is masculine–feminine since it is invariable in gender, that is, it can go both with masculine and feminine nouns (*hombre canadiense*, *mujer canadiense*). In Catalan, on the other hand, the adjective has a different inflection for the masculine and the feminine (*home canadenc*, *dona canadenca*). Therefore, when translating from Spanish to Catalan it is not possible to know, without looking at the accompanying noun, whether the Spanish adjective (*mf*) has to be translated as a feminine or a masculine adjective in Catalan. In that case, the symbol `GD` (for "gender to be determined") is used instead of the gender symbol. The word's gender will be determined by the structural transfer module, by means of a transfer rule (a rule that detects the gender of the preceding noun in this particular case). Therefore, `GD` must be used only when translating from Spanish to Catalan, but not when translating from Catalan to Spanish, as in Spanish the gender will always be `mf` regardless of the gender of the original word. In the bilingual dictionary you will need to add, in this case, more than one entry with direction indications, as you must specify in which translation direction the gender remains undetermined. The entries for this adjective should be as follows:

```
<e r="LR">
  <p>
    <l>canadiense<s n="adj"/><s n="mf"/></l>
    <r>canadenc<s n="adj"/><s n="GD"/></r>
  </p>
</e>
<e r="RL">
  <p>
    <l>canadiense<s n="adj"/><s n="mf"/></l>
    <r>canadenc<s n="adj"/><s n="f"/></r>
  </p>
</e>
<e r="RL">
  <p>
    <l>canadiense<s n="adj"/><s n="mf"/></l>
    <r>canadenc<s n="adj"/><s n="m"/></r>
  </p>
</e>
```

"`LR`" means *left to right* and "`RL`", *right to left*. Since Spanish is on the left and Catalan on the right, the adjective will be `GD` only when translating

from Spanish to Catalan (LR). For the translation RL you need to create two entries, one for the adjective in feminine and another one for the adjective in masculine.[1]

The same principle applies when it is not possible to determine the number of the target word for the same reasons mentioned above. For example, the Spanish noun *rascacielos* ("skyscraper") is invariable in number, that is, it can be singular as well as plural (*un rascacielos*, *dos rascacielos*). In Catalan, on the other hand, the noun has a different inflection for the singular and for the plural (*un gratacel*, *dos gratacels*). In this case the symbol used is "ND" ("number to be determined") and the entries should be like this:

```
<e r="LR">
  <p>
    <l>rascacielos<s n="n"/><s n="m"/><s n="sp"/></l>
    <r>gratacel<s n="n"/><s n="m"/><s n="ND"/></r>
  </p>
</e>
<e r="RL">
  <p>
    <l>rascacielos<s n="n"/><s n="m"/><s n="sp"/></l>
    <r>gratacel<s n="n"/><s n="m"/><s n="pl"/></r>
  </p>
</e>
<e r="RL">
  <p>
    <l>rascacielos<s n="n"/><s n="m"/><s n="sp"/></l>
    <r>gratacel<s n="n"/><s n="m"/><s n="sg"/></r>
  </p>
</e>
```

For a more detailed description of this kind of entries, refer to section 39.

5.2.1 Adding direction restrictions

In the previous example we have already seen the use of direction restrictions for entries with undetermined gender or number (GD or ND). These restrictions can also be used in other cases.

It is important to note that the current version of Apertium can give only a single equivalent for each source-language lexical form (a lexical

[1] You could also group them using a small paradigm

form is the lemma plus its grammatical information), that is, no word-sense disambiguation is performed.[2] When a lexical form can be translated in two or more different ways, one has to be chosen (the most general, the most frequent, etc.). You can tell Apertium that a certain word has to be analyzed ("understood") but not generated, as it is not the translation of any word in the other language.

Let's see this with an example. The Spanish noun *muñeca* can be translated in two different ways in Catalan depending on its meaning: *canell* ("wrist") or *nina* ("doll"). The context decides which translation is the correct one, but in its present state Apertium can not make such a decision .[3] Therefore, you have to decide which word you want as an equivalent when translating from Spanish to Catalan. From Catalan to Spanish, both words can be translated as *muñeca* without any problem. You have to specify all these circumstances in the dictionary entries using direction restrictions (`LR` meaning "left to right", that is, `es–ca`, and `RL` meaning "right to left", that is, `ca–es`). If you decide to translate *muñeca* as *canell* in all cases, the entries in the bilingual dictionary shall be:

```
<e>
  <p>
    <l>muñeca<s n="n"/><s n="f"/></l>
    <r>canell<s n="n"/><s n="m"/></r>
  </p>
</e>

<e r="RL">
  <p>
    <l>muñeca<s n="n"/></l>
    <r>nina<s n="n"/></r>
  </p>
</e>
```

This means that translation directions will be:

```
muñeca --> canell
muñeca <-- canell
muñeca <-- nina
```

[2]The system performs only part-of-speech disambiguation for homograph words, that is, for ambiguous words that can be analyzed as more than one lexical form, like *vino* in Spanish, that can mean both "wine" and "he/she came". This type of disambiguation is performed by the tagger.

[3]See Section 5.2.2 on multiword units for ways to circumvent this problem.

(Note that that there is also a gender change in the case of *muñeca* (feminine) and *canell* (masculine)).

It should be emphasized that a lemma can not have two translations in the target language, because the system would give an error when translating that lemma (see Section 5.5 "Detecting errors" to see how to find and correct these and other types of errors). When a word can be translated in two different ways in the target language in all contexts, you need to choose one as the translation equivalent and leave the other one as a lemma that can be analyzed but not generated, using direction restrictions like in the previous example. For example, the Catalan lemmas *mot* and *paraula* can be both translated into Spanish as *palabra* ("word") and the entries in the bilingual dictionary should look like this:

```
<e>
  <p>
    <l>palabra<s n="n"/></l>
    <r>paraula<s n="n"/></r>
  </p>
</e>

<e r="RL">
  <p>
    <l>palabra<s n="n"/><s n="f"/></l>
    <r>mot<s n="n"/><s n="m"/></r>
  </p>
</e>
```

Therefore, for this lemmas the translation directions will be:

```
palabra --> paraula
palabra <-- paraula
palabra <-- mot
```

One may have to specify restrictions regarding translation direction also in monolingual dictionaries. For example, both Spanish forms *cantaran* and *cantasen* should be analyzed as lemma *cantar*, verb, subjunctive imperfect, 3rd person plural, but when generating Spanish text, one has to decide which one will be generated. Monolingual dictionaries are read in two directions depending on its purpose: for the analysis, the reading direction is left to right; for the generation, right to left. Therefore, a word that must be analyzed but not generated must have the restriction `LR`, and a word that must be generated but not analyzed must have the restriction `RL`.

The case of *cantaran* or *cantasen* must have already been taken care of in inflection paradigms and it is unlikely to be a problem for most people extending a dictionary. In some other cases it can be necessary to introduce a restriction in the word entries of monolingual dictionaries.

5.2.2 Adding multiwords

It is possible to create entries consisting of two or more words, if these words are considered to build a single "translation unit". These multiword units can also be useful when it comes to select the correct equivalent for a word inside a fixed expression. For example, the Spanish word *dirección* may be translated into two Catalan words: *direcció* ("direction, management, directorate, steering", etc.) and *adreça* ("address"); including, for example, frequent multiword units such as *dirección general* → *direcció general* ("general directorate") and *dirección postal* → *adreça postal* ("postal address") may help get improved translations in some situations.

Multiword units can be classified basically into two categories: multiwords with inner inflection and multiwords without inner inflection.

5.2.2.1 Multiwords without inner inflection

They are just like the normal one-word entries, with the only difference that you need to insert the element `<b>` (which represents a blank) between the individual words that make up the unit. Therefore, if you want to add, for example, the Spanish multiword *hoy en día* ("nowadays"), whose equivalent in Catalan is *avui dia*, the entries you need to add to the different dictionaries are:

- Spanish monolingual dictionary:

```
<e lm="hoy en día">
  <i>hoy<b/>en<b/>día</i>
  <par n="ahora__adv"/>
</e>
```

- Catalan monolingual dictionary:

```
<e lm="avui dia">
  <i>avui<b/>dia</i>
  <par n="ahir__adv"/>
</e>
```

- Spanish-Catalan bilingual dictionary:

```
<e>
  <p>
    <l>hoy<b/>en<b/>día<s n="adv"/></l>
    <r>avui<b/>dia<s n="adv"/></r>
  </p>
</e>
```

For Spanish-Galician pair, if you want to add, for example, the Spanish multiword *manga por hombro* ("disarranged"), whose equivalent in Galician is *sen xeito nin modo*, the entries you need to add are:

- Spanish monolingual dictionary:

```
<e lm="manga por hombro">
  <i>manga<b/>por<b/>hombro</i>
  <par n="ahora__adv"/>
</e>
```

- Galician monolingual dictionary:

```
<e lm="sen xeito nin modo">
  <i>sen<b/>xeito<b/>nin<b/>modo</i>
  <par n="Deo_gratias__adv"/>
</e>
```

- Spanish-Galician bilingual dictionary:

```
<e>
  <p>
    <l>manga<b/>por<b/>hombro<s n="adv"/></l>
    <r>sen<b/>xeito<b/>nin<b/>modo<s n="adv"/></r>
  </p>
</e>
```

5.2.2.2 Brief introduction to paradigms

The paradigms of the previous examples, as adverbs do not inflect, contain only the grammatical symbol of the lexical form, as you see in this example:

```
<pardef n="ahora__adv">
  <e>
    <p>
      <l/>
      <r><s n="adv"/></r>
    </p>
  </e>
</pardef>
```

Paradigms are build like a lexical entry. We have seen so far lexical entries where the common part of the lemma is put between :

```
<e lm="cósmico">
  <i>cósmic</i>
  <par n="absolut/o__adj"/>
</e>
```

But you can also express the same with a pair of strings: a left string <l> and a right string <r> inside a <p> element:

```
<e lm="cósmico">
  <p>
    <l>cósmic</l>
    <r>cósmic</r>
  </p>
  <par n="absolut/o__adj"/>
</e>
```

These two entries are equivalent. The use of the element helps get more simple and compact entries, and you can use it when the left side and the right side of the string pair are identical. As has been explained before, monolingual dictionaries are read LR for the analysis of a text and RL for the generation. Therefore, when there is some difference between the analysed string and the generated string (not very usual) the entry can not be written using the element. This is what happens in paradigms, where the left and right strings are never identical, since the right side must contain the grammatical symbols that will go through all the modules of the system.

5.2.2.3 Multiwords with inner inflection

They consist of a word that can inflect (typically a verb) followed by one or more invariable words. For these entries you need to specify the inflection

paradigm just after the word that inflects. The invariable part must be marked with the element <g> (for *group*) in the right side. The blanks between words are indicated, like in the previous case, with the element <b>. Look at the following example for the Spanish multiword *echar de menos* (to miss), translated into Catalan as *trobar a faltar*:

- Spanish monolingual dictionary:

```
<e lm="echar de menos">
    <i>ech</i>
    <par n="aspir/ar__vblex"/>
    <p>
      <l><b/>de<b/>menos</l>
      <r><g><b/>de<b/>menos</g></r>
    </p>
</e>
```

- Catalan monolingual dictionary:

```
<e lm="trobar a faltar">
    <i>trob</i>
    <par n="abander/ar__vblex"/>
    <p>
      <l><b/>a<b/>faltar</l>
      <r><g><b/>a<b/>faltar</g></r>
    </p>
</e>
```

- Spanish-Catalan bilingual dictionary:

```
<e>
  <p>
    <l>echar<g><b/>de<b/>menos</g><s n="vblex"/></l>
    <r>trobar<g><b/>a<b/>faltar</g><s n="vblex"/></r>
  </p>
</e>
```

Note that the grammatical symbol is appended at the end, after the group marked with the <g>.

It can be the case that a lemma is a multiword of this kind in one language and a single word in the other language. In that case, in the bilingual dictionary, the multiword will contain the <g> element and the single word will not. In the monolingual dictionaries, each entry will be created

according to its type. Look at the following example for the Spanish multiword *darse cuenta* (to realize), translated into Catalan as the verb *adonar-se*:[4]

- Spanish monolingual dictionary:

```
<e lm="darse cuenta">
    <i>d</i>
    <par n="d/ar__vblex"/>
    <p>
      <l><b/>cuenta</l>
      <r><g><b/>cuenta</g></r>
    </p>
</e>
```

- Catalan monolingual dictionary:

```
<e lm="adonar-se">
    <i>adon</i>
    <par n="abander/ar__vblex"/>
</e>
```

- Spanish-Catalan bilingual dictionary:

```
<e>
  <p>
    <l>dar<g><b/>cuenta</g><s n="vblex"/></l>
    <r>adonar<s n="vblex"/></r>
  </p>
</e>
```

The same principles and actions described for basic entries (gender and number change, direction restrictions, etc.) apply to all kinds of multiwords. For a more detailed description of multiword units, refer to section 3.1.2.6.

[4]The verb *adonar-se* is considered a simple word, since the incorporation of enclitic pronouns (such as "-se") is treated inside the inflection paradigms of verbs (for all the Romance languages of *Apertium*); therefore, it is not necessary to specify them in lexical entries. The correct placement of clitic pronouns is one of the main reasons for using the <g>... </g> labels around the invariable part of multi-word verbs.

5.2.3 Consider contributing your improved lexical data

If you have successfully added general-purpose lexical data to any of the Apertium language pairs, please consider contributing it to the project so that we can offer a better toolbox to the community. You can e-mail your data (in three XML files, one for each monolingual dictionary and another one for the bilingual dictionary) to the following addresses:

Spanish-Catalan data	Mireia Ginestí: `mginesti@dlsi.ua.es`
Spanish-Portuguese data	Carme Armentano: `carmentano@dlsi.ua.es`[5]
Spanish-Galician data	Xavier Gómez-Guinovart: `xgg@uvigo.es`

If you believe you are going to contribute more heavily to the project, you can join the development team through www.sourceforge.net. If you do not have a Sourceforge account, please create one; then write to Mikel L. Forcada (`mlf@ua.es`) or Sergio Ortiz (`sortiz@dlsi.ua.es`), or to Xavier Gómez Guinovart if you are interested in the Spanish-Galician language pair, explaining briefly your motivations and background to join the project. The usual way to contribute is to use CVS; as a project member, you will be able to commit your changes to dictionaries directly.

The addition of simple lexical contributions will soon be made simpler by means of web forms in `http://xixona.dlsi.ua.es/prototype/webform/`, so that contributors do not have to deal directly with XML.

You should be aware that the data you contribute to the project, once added, will be freely distributed under the current license (GNU General Public License or Creative Commons 2.5 attribution-sharealike-noncommercial, as indicated). Make sure the data you contribute is not affected by any kind of license which may be incompatible with the licenses used in this project. No kind of agreement or contract is created between you and the developers. If you have any doubt, or you plan to make a massive contribution, contact Mikel L. Forcada.

5.3 Adding structural transfer (grammar) rules

The content in this chapter partially repeats information already presented in the chapter describing the structural transfer module (Section 3.5), although rules are described here in a more general and practical way, aimed at those who wish a first approach to them.

Structural transfer rules carry out transformations to the analysed and disambiguated text, which are needed because of grammatical, syntacti-

cal and lexical divergences between the two languages involved (gender and number changes to ensure agreement in the target language, word reorderings, changes in prepositions, etc.). The rules detect patterns (sequences) of source text lexical forms and apply to them the corresponding transformations. The module detects the patterns in a left-to-right, longest-match way; for example, the phrase *the big cat* will be detected and processed by the rule for *determiner–adjective–noun* and not by the rule for *determiner–adjective*, since the first pattern is longer. If two patterns have the same length, the rule that applies is the one defined in the first place.

The structural transfer module (generated from the structural transfer rules file) calls the lexical transfer module (generated from the bilingual dictionary) all through the process to determine the target language equivalents of the source language lexical forms.

The structural transfer rules are contained in a XML file, one for each translation direction (for example, for the translation from Spanish to Catalan, the file is `apertium-es-ca.trules-es-ca.xml`). You need to edit this file if you want to add or change transfer rules.

Rules have a **pattern** and an **action** part. The pattern specifies which sequences of lexical forms have to be detected and processed. The action describes the verifications and transformations that need to be done on its constituents. Usual transformation operations (such as gender and number agreement) are defined inside a macroinstruction which is called inside the rule. At the end of the action part of the rule, the resulting lexical forms in the target language are sent out so that they are processed by the next modules in the translation system.

A transfer rules file contains four sections with definitions of elements used in the rules, and a fifth section where the actual rules are defined. The sections are the following:

- `<section-def-cats>`: This section contains the definition of the categories which are to be used in the rule patterns (that is, the type of lexical forms that will be detected by a certain rule). For the rule presented below, the categories `det` and `nom` (determiner and noun) need to be defined here. Categories are defined specifying the grammatical symbols that the lexical forms have. An asterisk indicates that one or more grammatical symbols follow the ones specified. The following is the definition of the category `det`, which groups determiners and predeterminers[6] in the same category since they play the same role for transfer purposes:

[6] such as in Spanish *todo*, *toda*, *todos*, *todas*

```
<def-cat n="det">
    <cat-item tags="det.*"/>
    <cat-item tags="predet.*"/>
</def-cat>
```

It is also possible to define as a category a certain lemma, like the following for the preposition `en`:

```
<def-cat n="en">
    <cat-item lemma="en" tags="pr"/>
</def-cat>
```

- `<section-def-attrs>`: This section contains the definition of the attributes that will be used inside of the rules, in the action part. You need attributes for all the categories defined in the previous section, if they are to be used in the action part of the rule (to make verifications on them or to send them out at the end of the rule), as well as for other attributes needed in the rule (such as gender or number). Attributes have to be defined using their corresponding grammatical symbols and can not have asterisks; its name must be unique. The following are the definitions for the attributes `a_det` (for determiners) and `gen` (gender):

```
<def-attr n="a_det">
    <attr-item tags="det.def"/>
    <attr-item tags="det.ind"/>
    <attr-item tags="det.dem"/>
    <attr-item tags="det.pos"/>
    <attr-item tags="predet"/>
</def-attr>

<def-attr n="gen">
    <attr-item tags="m"/>
    <attr-item tags="f"/>
    <attr-item tags="mf"/>
    <attr-item tags="nt"/>
    <attr-item tags="GD"/>
</def-attr>
```

- `<section-def-vars>`: This section contains the definition of the variables used in the rules.

```
<def-var n="interrogativa"/>
```

- `<section-def-macros>`: Here the macroinstructions are defined, which contain sequences of code that are frequently used in the rules; this way, linguists do not need to write the same actions repeatedly. There are, for example, macroinstructions for gender and number agreement operations.
- `<section-def-rules>`: This is the section where the structural transfer rules are written.

The following is an example of a rule which detects the sequence *determiner–noun*:

```
<rule>
  <pattern>
    <pattern-item n="det"/>
    <pattern-item n="nom"/>
  </pattern>
  <action>
    <call-macro n="f_concord2">
      <with-param pos="2"/>
      <with-param pos="1"/>
    </call-macro>
    <out>
      <lu>
        <clip pos="1" side="tl" part="whole"/>
      </lu>
      <b pos="1"/>
      <lu>
        <clip pos="2" side="tl" part="whole"/>
      </lu>
    </out>
  </action>
</rule>
```

Part of the action performed on this pattern is specified inside the macroinstruction `f_concord2`, which is defined in the `<section-def-macros>`. It performs gender and number agreement operations: if there is a gender or number change between the source language and the target language (in the noun), the determiner changes its gender or number accordingly; furthermore, if gender or number are undetermined (`GD` or `ND`[7]), the noun

[7] See pages 40 or 131

receives the correct gender or number values from the preceding determiner. In the Apertium es–ca, es–gl and es–pt systems, there are agreement macroinstructions defined for one, two, three or four lexical units (`f_concord1`, `f_concord2`, `f_concord3`, `f_concord4`). When calling the macroinstructions in a rule, it must be specified which is the main lexical unit (the one which most heavily determines the gender or number of the other lexical units) and which other lexical units of the pattern have to be included in the agreement operations, in order of importance. This is done with the `<with-param pos=""/>` element. In the presented rule, the main lexical unit is the noun (position "2" in the pattern) and the second one is the determiner (positions "1" in the pattern).

After the pertinent actions, the resulting lexical forms are sent out, inside the `<out>` element. Each lexical unit is defined with a `<clip>`. Its attributes mean the following:

- `pos`: refers to the position of the lexical form in the pattern; `1` is the first lexical form (the determiner) and `2` the second one (the noun).
- `side`: indicates if the lexical form is in the source language (`sl`) or in the target language (`tl`). Of course, words are sent out always in the target language; source language lexical forms may be needed inside of a rule, when testing its attributes or characteristics.
- `part`: indicates which part of the lexical form is referred to in the `clip`. You can use some predefined values:
  - `whole`: the whole lexical form (lemma and grammatical symbols). Used only when sending out the lexical unit (inside an `<out>` element).
  - `lem`: the lemma of the lexical unit
  - `lemh`: the head of the lemma of a multiword with inner inflection (see Section 5.2.2 in this chapter, or Section 3.1.2.6 if you wish a more detailed description)
  - `lemq`: the queue of a lemma of a multiword with inner inflection

  Apart from these predefined values, you can use any of the attributes defined in `<section-def-attrs>` (for example `gen` or `a_det`).

  The values `lemh` and `lemq` are used when sending out multiwords with inner inflection in order to place the head and the queue of the lemma in the right position, since the previous module moved the

queue just after the lemma head for various reasons. In practice, in our system, this means that you must use these values instead of `whole` when sending out verbs. This is because, in our dictionaries, multiwords with inner inflection are always verbs and, if you use the value `whole` when sending them out, the multiword would not be well formed (the head and the queue of the lemma would not have the correct position and the multiword could not be generated by the generator).

Therefore, a rule that has a verb in its pattern must send the lexical forms like in the following two examples:

```
<rule>
  <pattern>
    <pattern-item n="verb"/>
  </pattern>
  <action>
    <out>
      <lu>
        <clip pos="1" side="tl" part="lemh"/>
        <clip pos="1" side="tl" part="a_verb"/>
        <clip pos="1" side="tl" part="temps"/>
        <clip pos="1" side="tl" part="persona"/>
        <clip pos="1" side="tl" part="gen"/>
        <clip pos="1" side="tl" part="nbr"/>
        <clip pos="1" side="tl" part="lemq"/>
      </lu>
    </out>
  </action>
</rule>
```

```
<rule>
  <pattern>
    <pattern-item n="verb"/>
    <pattern-item n="prnenc"/>
  </pattern>
  <action>
    <out>
      <mlu>
        <lu>
          <clip pos="1" side="tl" part="lemh"/>
          <clip pos="1" side="tl" part="a_verb"/>
```

```
          <clip pos="1" side="tl" part="temps"/>
          <clip pos="1" side="tl" part="persona"/>
          <clip pos="1" side="tl" part="nbr"/>
        </lu>
        <lu>
          <clip pos="2" side="tl" part="lem"/>
          <clip pos="2" side="tl" part="a_prnenc"/>
          <clip pos="2" side="tl" part="persona"/>
          <clip pos="2" side="tl" part="gen"/>
          <clip pos="2" side="tl" part="nbr"/>
          <clip pos="1" side="tl" part="lemq"/>
        </lu>
      </mlu>
    </out>
  </action>
</rule>
```

The first rule detects a verb and places the queue in the correct place, after all the grammatical symbols. The lexical unit is sent specifying the attributes separately: lemma head, lexical category (verb), tense, person, gender (for the participles), number and lemma queue.

The second rule detects a verb followed by an enclitic pronoun and sends the two lexical forms specifying also the attributes separately; the first lexical unit consists of: lemma head, lexical category (verb), tense, person and number; the second lexical unit consists of: lemma, lexical category (enclitic pronoun), person, gender, number and lemma queue (of the first lexical form). This way, the queue of the lemma is placed after the enclitic pronoun. The two lexical units (verb and enclitic pronoun) are sent inside a `<mlu>` element, since they have to reach the morphological generator as a multilexical unit (multiword).

Taking into account what we have explained here, if you want to **add a new transfer rule** you have to follow these steps:

1. Specify which pattern you want to detect. Bear in mind that words are processed only once by a rule, and that rules are applied left to right and choosing the longest match. For example, imagine you have in your transfer rules file only two rules, one for the pattern *determiner–noun* and one for the pattern *noun–adjective*. The Spanish phrase *el valle verde* ("the green valley") would be detected and processed by the first one, not by the second. You will need to add a rule for the pattern *determiner - noun - adjective* if you wish that the three lexical units are processed in the same pattern.

2. Describe the operations you want to perform on the pattern. In the Apertium `es-ca`, `es-gl` and `es-pt` systems, simple agreement operations (gender and number agreement) are easy to perform in a rule by means of a macroinstruction. To perform other operations, you will need to use more complicated elements; for a more detailed description of the language used to create rules, refer to the section 3.5.4.

3. Send the lexical units of the pattern in the target language inside an `<out>` element. Each lexical unit must be included in a `<lu>` element. If two or more lexical units must be generated as a multilexical unit (only for enclitic pronouns in the present language pairs) , they must be grouped inside a `<mlu>` element.

   All the words that are detected by a rule (that are part of a pattern) must be sent out at the end of the rule so that the next module (the generator) receives them. If a lexical unit is detected by a pattern and is not included in the `<out>` element, it will not be generated.

5.4 Adding data for the lexical categorial disambiguator (part-of-speech tagger)

The lexical categorial disambiguator takes the linguistic information needed to disambiguate a text basically from two sources: a tagset definition file and corpora. The tagset definition file is contained in the linguistic data directory and its name has the structure `apertium-PAIR.LANG.tsx`, whereas corpora information is contained in the `LANG-tagger-data` directory included in the previous directory.

The *tagset definition file* contains the definition of the coarse tags (or categories) used by the tagger when being trained and when disambiguating a text, as well as tag co-occurrence restrictions that help obtain better tag probabilities. In Section 3.2 you can find a detailed description of its characteristics.

The *corpora* that need to be in the `LANG-tagger-data` directory are different depending on whether the tagger is trained in a supervised way (with manually disambiguated text) or unsupervised (without manually disambiguated text):

- to train the tagger in a supervised way you need the files (examples from es-tagger-data): `es.tagged.txt`, `es.untagged`, `es.tagged`, `es.dic`.

- to train the tagger in an unsupervised way you need the files (examples from es-tagger-data): `es.crp.txt`, `es.crp`, `es.dic`

These files have the following characteristics:

- `es.tagged.txt`: A Spanish corpus in plain text format.
- `es.untagged`: The corpus `es.tagged.txt` morphologically analysed, which means, processed by the de-formatter and the morphological analyser (automatically generated corpus).
- `es.tagged`: The preceding corpus manually disambiguated.
- `es.crp.txt`: A large corpus (hundreds of thousands of words) used when training the tagger in an unsupervised way with Baum-Welch reestimation.
- `es.crp`: The preceding corpus processed consecutively by the de-formatter and the morphological analyser (automatically generated corpus).
- `es.dic`: File created from the Spanish monolingual dictionary `*.es.dix`, by means of the `lt-expand` and `apertium-filter-ambiguity` tools, which expand the dictionary and filter the ambiguity classes, so that the file contains all the forms identified as different ambiguity classes by the tagger defined with `*.es.tsx`; that is, which lexical categories can be homographs (automatically generated corpus).

When downloading Apertium from Sourceforge (`http://apertium.sourceforge.net/`), if the tagger has been trained in a supervised way, it is probable that you get the files needed for this kind of training, `es.tagged` and `es.tagged.txt` (for Spanish). The other required files are automatically generated when running the training. If the tagger has been trained in an unsupervised way, you will not get any corpus in the download since the files required for this kind of training are huge. If you wish to train the tagger with this method, you will need to collect a large corpus and name it `es.crp.txt`. The other required files are automatically generated when running the training.

Anyway, the Apertium translator comes with all the data required for a good performance of the tagger. You don't need to train the tagger in order to use Apertium. A retraining might be required in the case that you have made really extensive changes to the dictionaries or you have modified the tagset definition file.

Therefore, the tagger data can be modified in two ways:

1. Change the tagset definition file. You can add, change or delete the coarse tags used by the tagger, if you think that a new category could be useful for the disambiguation or that a certain category should be modified to obtain better results. You can also add restrictions (for example, you can forbid the sequence determiner–determiner if this is an impossible combination in a given language and can help in the disambiguation of certain homograph words).

2. Modify the corpora used to train the tagger. You can modify the manually disambiguated text (`es.tagged` for Spanish) if you think that certain tags have been wrongly selected. You can also add sentences to this text (and to `es.tagged.txt`, used to automatically generate the corpus `es.untagged`) in order to add information to the tagger, since it is possible that certain combinations are incorrectly disambiguated because the tagger has not found them in the training corpora.

There are two commands to run the training:

- to train in a supervised way, type, in the directory containing the linguistic data (example for *es–ca*): `make -f es-ca-supervised.make`

- to train in an unsupervised way, type, in the directory containing the linguistic data (example for *es–ca*): `make -f es-ca-unsupervised.make`

In both cases, planned files will be automatically generated.

5.5 Detecting errors

It is easy to make errors when adding new words or transfer rules to the Apertium system.

On the one hand, it is possible that, when compiling the new files, the system displays an error message. In this case, this is a formal error (a missing XML tag, a tag that is not allowed in a certain context, etc.). You just have to go to the line number indicated by the error message, correct the error and compile again. On the other hand, there are other types of errors not detected when compiling, but which can make the system mistranslate a word or give an incomprehensible text string. These are linguistic errors, which can be detected and corrected with the tips given in

this chapter. The following information is for Linux users, since Apertium works for the moment only in this operating system.[8]

5.5.1 Adjusting error symbols

When the system encounters a problem to translate any word of a source language text, in the default mode the system outputs the problematic word together with a symbol that indicates that an error has occurred. The meaning of the different symbols is the following:

- '@': The problem is in the lexical transfer module, which can not translate the lexical form (the bilingual dictionary does not contain it)
- '#': The problem has occurred in the generator, which can not generate the surface form from the input lexical form (the morphological dictionary does not contain it in the generation direction)
- '/': This symbol separates two or more surface forms delivered by the generator. The problem, therefore, is in the target language monolingual dictionary, which has, in the generation direction, two surface forms for a single lexical form, when it should have only one.

The generation module has three modes, which enable us to decide how errors will be displayed in the final output. The three possible parameters are:

- -n : error symbols and the unknown-word symbol will NOT be displayed, and neither will any grammatical symbols
- -g : error symbols and the unknown-word symbol will be displayed (default mode)
- -d : error symbols and the unknown-word symbol will be displayed, as well as the grammatical symbols of the lexical forms producing the error.

The preferable mode depends on the type of user and on the translation purpose. The first option is the most suitable when the user does not want that external signs interfere in the reading of the translation. The second

[8]There are in `http://apertium.org` experimental packages for Windows with fixed linguistic data (non-modifiable binary files).

option is useful when the user wants the system to show where there has been a problem in the translation (errors or unknown words) in order to be able to post-edit it easily. The third option is ideal for linguistic developers of Apertium, since it displays all the linguistic information of the forms that produced an error.

Taking advantage of the error symbols output by the system, it is possible to carry out a thorough test of the dictionaries of a certain language pair. This will enable you to detect and correct all its errors. To learn how to do it, see Section 5.5.4.

5.5.2 Output of the different Apertium modules

Sometimes it is difficult to find the origin of an error. In such cases, it is useful to see the output of each of the modules. As all the data processed by the system, from the original text to the translated text, circulate between the eight modules of the system in text format, it is possible to stop the text stream at any point to know what is the input or the output of a certain module.

Using a pipeline structure and the `echo` or `cat` commands, you can send a text through one or more modules to analyse their output and detect the origin of the error. We describe next how to do it. You have to move to the directory where the linguistic data are saved and type the described commands.

5.5.2.1 The morphological analyser output

To know how a word is analyzed by the translator, type the following in the terminal (example for the Catalan word *sabates*):

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
```

You can replace `ca-es` with the translation direction you want to test.
The output in Apertium should be:

```
^sabates/sabata<n><f><pl>$^./.<sent>$[][]
```

The string structure is `^word/lemma<`*morphological analysis*`>$`. The `<sent>` tag is the analysis of the full stop, as every sentence end is represented as a full stop by the system, whether or not explicitly indicated in the sentence.

The analysis of an unknown word is (ignoring the full stop info):

```
^genoma/*genoma$
```

and the analysis of an ambiguous word:

```
^casa/casa<n><f><sg>/casar<vblex><pri><p3><sg>/casar<vblex><imp><p2><sg>
```

Each lexical form (lemma plus morphological analysis) is presented as a possible analysis of the word *casa*.

5.5.2.2 The tagger output

To know the output of the tagger for a source language text, type the following in the terminal (example for the Catalan-Spanish direction):

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
|apertium-tagger -g ca-es.prob
```

The output will be:

```
^sabata<n><f><pl>$^./.<sent>$[][]
```

The output for an ambiguous word will be like the one above, since the tagger chooses one lexical form among all the possibilities. Therefore, the output for *casa* in Catalan will be, for example (depending on the context):

```
^casa<n><f><sg>$^.<sent>$[][]
```

5.5.2.3 The `pretransfer` output

This module applies some changes to multiwords (move the lemma queue of a multiword with inner inflection just after the lemma head). To know its output, type:

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
|apertium-tagger -g ca-es.prob | apertium-pretransfer
```

Since *sabates* is not a multiword, this module does not alter its input.

5.5.2.4 The structural and lexical transfer output

To know how a word, phrase or sentence is translated into the target language and processed by structural transfer rules, type the following in the terminal:

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
|apertium-tagger -g ca-es.prob | apertium-pretransfer
| ./ca-es.transfer ca-es.autobil.bin
```

The output for this word will be:

```
^zapato<n><m><pl>$^.<sent>$[][]
```

Analysing how a word or phrase is output by this module can help you detect errors in the bilingual dictionary or in the structural transfer rules. Typical bilingual dictionary errors are: two equivalents for the same source language lexical form, or wrong assignment of grammatical symbols. Errors due to structural transfer rules vary a lot depending on the actions performed by the rules.

5.5.2.5 The morphological generator output

To know how a word is generated by the system, type the following in the terminal:

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
|apertium-tagger -g ca-es.prob | apertium-pretransfer
| ./ca-es.transfer ca-es.autobil.bin | ltproc -g ca-es.autogen.bin
```

With this command you can detect generation errors due to an incorrect entry in the target language monolingual dictionary or to a divergence between the output of the bilingual dictionary (the output of the previous module) and the entry in the monolingual dictionary.

The correct output for the input *sabates* would be:

```
zapatos.[][]
```

There are in this step no grammatical symbols, and the word appears inflected.

5.5.2.6 The post-generator output

It is not very usual to have errors due to the post-generator, because of its generally small size and the fact that it is seldom changed after adding usual combinations, but you can also test how a source language text comes out of this module, by typing:

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
|apertium-tagger -g ca-es.prob | apertium-pretransfer
| ./ca-es.transfer ca-es.autobil.bin | ltproc -g ca-es.autogen.bin
| ltproc -p es-ca.autopgen.bin
```

5.5.2.7 The Apertium output

You can put all the modules of the system in the pipeline structure and see how a source language text goes through all the modules and gets translated into the target language. You just have to add the re-formatter to the previous command:

```
echo "sabates" | apertium-destxt | lt-proc ca-es.automorf.bin
|apertium-tagger -g ca-es.prob | apertium-pretransfer
| ./ca-es.transfer ca-es.autobil.bin | ltproc -g ca-es.autogen.bin
| ltproc -p es-ca.autopgen.bin | apertium-retxt
```

This is the same as using the `apertium` shell script provided by the Apertium package:

```
echo "sabates" | apertium . ca-es
```

(The dot indicates the directory where the linguistic data are saved, in this case the current directory).

Of course, instead of typing all the presented commands every time you need to test a translation, you can create shell scripts for every action and use them to test the output of each module.

5.5.3 Error examples

1) We can get the following kind of output in a translation:

```
$ echo "nord" | apertium . ca-es
$ #norte<n><m><sg>
```

This means that the word was correctly translated by the bilingual dictionary but that the system does not find it in the Spanish morphological dictionary to generate it. The problem can be in the morphological dictionary but can also be caused by an incorrect bilingual entry, in which the grammatical symbols that the translated word is assigned do not correspond with the grammatical symbols that this word has in the morphological dictionary.

2) The following `es-ca` bilingual entry does not take into account the gender change between *adhesiu* (masculine) and *pegatina* (feminine), causing the translator to give an error:

```
<e>
  <p>
    <l>pegatina<s n="n"/></l>
    <r>adhesiu<s n="n"/></r>
  </p>
</e>
```

```
$ echo "adhesiu" | apertium . ca-es
$ #pegatina<n><m><sg>
```

The correct entry should be:

```
<e>
  <p>
    <l>pegatina<s n="n"/><s n="f"/></l>
    <r>adhesiu<s n="n"/><s n="m"/></r>
  </p>
</e>
```

3) The following error is given when the source language lexical form can not be found in the bilingual dictionary, either because there is not an entry for this lemma or because the entry does not correspond with the grammatical symbols received from the analyser:

```
$ echo "illot" | apertium . ca-es
$ @illot<n><m><sg>
```

4) When a source language lexical form has two correspondences in the bilingual dictionary, the translator output is like the following one:

```
$ echo "llavor" | apertium . ca-es
$ #pepita<n>/semilla<n><m><sg>
```

The solution is to put a direction restriction in one of the bilingual entries.

Some errors can be due to structural transfer rules. The way to solve a problem whose origin we don't know, is to test the output of the different modules to detect where the problem arises.

5.5.4 Testing the integrity of the dictionaries

It is highly advisable to test the integrity of our dictionaries from time to time, especially if we changed them significantly –or if we changed the transfer rules, because some errors can be due to its application.

The test is carried out in one translation direction. For this reason, for a given language pair, you will have to perform two tests, one in each direction.

The steps you have to follow to perform the test are:

- expand the source language monolingual dictionary, using the `lt-expand` tool, to obtain all the lexical forms (which are the forms that appear on the right of the colon in the output file);
- send these lexical forms (except those that are only generation forms, which `lt-expand` will have marked with the symbol '<') through all the system modules from pretransfer to the generator;
- Search in the result, the lexical forms marked with the symbols '#' , '@' or '/', which will be the error forms (see Section 5.5.1).

5.6 Generating a new Apertium system from modified data

If you make changes to any of the linguistic data files of Apertium (dictionaries, transfer rules or tagger definition file), the changes will not be applied until you recompile the modules. To do this, type `make` in the directory where the linguistic data are saved so that the system generates the new binary files.

If changes were made to the tagger definition file or to the corpora used to train the tagger, you will need also to retrain the tagger: in the same linguistic data directory, you have to type (example for the Spanish tagger in the es-ca translator) `make -f es-ca-unsupervised.make` for unsupervised training or `make -f es-ca-supervised.make` for supervised training.

After compilation, `apertium` will already use the new data.

Chapter 6

Data insertion web forms

This chapter describes the dictionary maintaining system in Apertium 2. It is organized in two sections. Section 6.2 gives the necessary information to install and adjust the web application for word insertion. Section 6.3 describes how to use the tool to add linguistic data.

6.1 Introduction

Adding lemmas to the dictionaries of the different languages in Apertium is a slow task if you do it by manually editing the XML dictionaries; for this reason web forms have been created, which make the word insertion task considerably easier and, furthermore, allow the users to do it remotely from any computer with Internet access.

The tool consists of a set of forms written in `php` which can be used from any Internet navigator, either locally in the same computer where dictionaries are saved, or remotely.

6.2 Installing and managing

6.2.1 Installing the tool

The installation must be done in a Unix machine which has an Apache web server with `php` installed. So, you will first need to install the `php` server if it is not installed, and then proceed to install the form tool.

To install the tool, download the package *'apertium-lexical-webform-0.9'* from the Apertium web page in Sourceforge (`http://apertium.sourceforge.net/`) and unpack it in the directory where you want to leave the tool.

```
# cd /path/to the /forms tar -xvzf
# /path/apertium-lexical-webform-0.9.tar.gz
```

You must take into account that Apache only serves the pages that are in the root directory that we configured. Therefore, the directory where you place the forms must be a subdirectory inside the root directory of the Apache server.

Next, you have to edit the configuration file, which you can find in *private/config.php*, and give the appropriate values to the configuration variables:

- `$anmor`: entire path of the morphological analyser `lt-proc`.
- `$dicos_path`: path to the directory where the final dictionaries and the compiled binaries of each dictionary are saved. This directory must contain a subdirectory for each dictionary with which the form can work. The subdirectory name must have the following structure: `paradigmes-ll-rr` , where *ll* and *rr* are the initials of the language pair involved. Each directory must contain the final dictionaries used by the machine translation system and the corresponding compiled binaries. These directories can be replaced with symbolic links in the case that they are located in a different place.
- `$usuaris_professionals`: a list of the professional users in the system that have permission to insert words in the form dictionaries and to validate entries pending confirmation.
- `$mail`: E-mail address of the administrator of the forms. When someone wants to register as a user, an e-mail will be sent to this address.

Once the parameters of this file have been configured, the forms server is already in use.

6.2.2 Directory structure

All the files required by the application are structured as follows:

- `/index.php`: displays the initial insertion form. It has a section for each language pair, where the user inserts the SL lemma and the TL lemma and chooses the appropriate part of speech. After pressing the *'Go on'* button, the next page is displayed, where the user has to select the appropriate inflection paradigms for the SL lemma and the TL lemma.

- `/dics`: directory that contains the dictionaries with the entries inserted from the forms. It contains the files with the entries from non-professional users (pending validation) and the dictionaries with the `XML` entries from professional users.

- `/private`: most modules used in the forms are saved here. It contains also the directories with the definition of paradigms for all the languages of the forms; these directories have the name `paradigmes-ll-rr`, where *ll* and *rr* are the initials of a given language pair. The order chosen for the two languages, first *ll* and then *rr*, depends on the order defined for entries in the bilingual dictionary. This directory contains also the files that carry out the whole processing of the words being inserted. These files are:

  - `resultado.php`: This *php* is called when two words for any language pair are inserted from the module *index.php*. Basically, what it does is to establish the language pair involved (*$LR* and *$RL*) and the part of speech of the words being inserted (*$tipus*). It is included in the *selec.php* module, that is the next one called in the insertion process. In the case that the *tipus* (*type*) of the word being inserted is a multiword unit (*Multi Word Verb*), then *multip.php* is the module included and called instead of *selec.php*. The *Multi Word Verb* elements consist of a verb that can inflect followed by an invariable queue of one or more words (see Section 3.1.2.6 for a detailed description).

  - `selecc.php`: This module is in charge of the selection of paradigms for the pair of words, the SL word and the TL word. It displays a list of paradigms to be chosen from, which depends on the part of speech of the entry being inserted. When a new paradigm is selected for a lemma, it displays some examples of inflected forms of the lemma according to the chosen paradigm. If the user accepts the chosen paradigms, the module calls *insertarPro.php* or *insertar.php* depending on whether the user is professional or non-professional respectively.

  - `multip.php`: It has the same function as the *selecc.php* module but for multiword units. It uses the same variables and performs the same actions, but in the examples displayed, the verb is inflected and the words of the queue are added after it. It works in an analogous way as the *selecc.php* module, whose detailed description can be found in Section 6.2.3.2.

- `valida.php`: This module is called when a professional user wants to validate words that are in the queue of entries pending validation. It consults the file of words to be validated reading them one by one; it takes the data of the entry in turn (*LRlem, RLlem, paradigmaLR, paradigmaRL, LR, RL*, etc.) and calls *selecc.php* to continue with the insertion process of that specific entry.
- `insertarPro.php`: This module is called when the paradigms for the SL word and the TL word have already been selected (which was done in *selecc.php*), and displays what the resulting `XML` entries will look like for the three dictionaries (SL monolingual, bilingual and TL monolingual) . From this screen it is possible to directly modify the code, and finally to accept the new entry or to cancel the operation.
- `ins_multip.php`: It has the same function as *insertarPro.php* but it is designed for multiword entries, therefore, the entry is treated differently so that the inserted `XML` code is correct.
- `insertar.php`: This module is equivalent to *insertarPro.php* but for non-professional users. The actions it performs are much more simple, since the module just adds the lemmas and the paradigms selected by the non-professional user to the file of words to be validated; they remain in this file until a professional user validates them.
- `verSemi.php`: This module displays the file of entries inserted by non-professional users which are waiting for validation. It is useful for professional users who, before starting validating words, want to see which words are in the queue waiting for validation. It can be called from a link displayed in the form generated by *selec.php*.
- `paradigmas.xsl`: Style sheet used to generate the paradigm files that are used by the form modules. It is used with the specification of paradigms of a language written in `XML` format. This question will be explained in more detail in Section 6.2.5 *Paradigm files*.
- `creaparadigma.awk`: `awk` file used also to generate the mentioned paradigm files.
- `gen_paradig.sh`: Script that can be used if we want to generate automatically the paradigm files for all the language pairs installed in our system.

In the next sections you will find a detailed description of the tasks of each module.

6.2.3 Php files

6.2.3.1 resultado.php

Depending on the value of the variable `$nomtrad` updated by *index.php*, the module assigns the appropriate values to `$LR` and `$RL` (source language and target language respectively). Then, according to the part of speech of the word being inserted, the variable $tipo is assigned the appropriate value, and then *selec.php* or *multip.php* are called depending on whether the word is a simple unit or a multiword unit.

6.2.3.2 selecc.php

The function of this module is the selection of a paradigm for the words being inserted. The user will have to select a paradigm for the SL word and another one for the TL word.

There are a group of variables which, depending on the part of speech of the word, are assigned certain values that will be used at the end ; these variables are:

- `cadFich:` part of speech of the lemma.
- `show:` string displayed in the form that indicates the part of speech of the word being inserted.
- `tag:` string with the `XML` tag output by the morphological analyser for this part of speech.
- `tagout:` string with the `XML` code that shows the part of speech of the word. This string will be used when building the final `XML` entry that will be inserted in the dictionary.
- `nota:` string with possible comments to be inserted in the `XML` code of the entry.

Forms work with 4 kinds of dictionaries:

- *Semi-professional dictionaries*: They contain the words inserted from the form by non-professional users and which are pending validation. Their extension is *"semi.dic"*

- *Form dictionaries*: They contain the words inserted from the form by professional users, and also the ones that have been validated from the semi-professional dictionaries. Their extension is "*webform*".
- *Final dictionaries*: The files with all the entries written in `XML` code. These are the files finally used by the translator after being compiled. Their extension is "*dix*".
- *Final compiled dictionaries*: These are the compiled final dictionaries, which can already be used by the binaries of the translator. Their extension is "*bin*"

All these dictionaries are used by the forms; there are variables that contain the paths to them. Values are also assigned to variables that manage the paths to the auxiliary and the configuration files:

- `path`: path to the temporary dictionaries.
- `fich_LR`: source language dictionary with the words inserted from the form that are not yet in the final dictionary nor in the compiled dictionary.
- `fich_RL`: target language dictionary with the words inserted from the form that are not yet in the final dictionary nor in the compiled dictionary.
- `fich_LRRL`: bilingual dictionary with the words inserted from the form that are not yet in the final dictionary nor in the compiled dictionary.
- `fich-semi`: entries inserted from the form by non-professional users and which are pending validation.
- `path_paradigmasLR`: path to the files that contain the inflection paradigms of the source language.
- `path_paradigmasRL`: path to the files that contain the inflection paradigms of the target language.
- `anmor`: path to the morphological analyser.
- `aut_LRRL`: path to the bilingual binary from source language to target language.
- `aut_RLLR`: path to the bilingual binary from target language to source language.

Then the html code is inserted with the operations to be performed depending on the selected action. The actions performed by the module are the following, in sequential order:

- Tests that the source language lemma being inserted is not already in the dictionaries containing the words inserted from the form. If `selecc.php` has been called from the word validation screen (`valida.php`), then the module tests that the lemma is not already in the file of words inserted by non-professional users. It tests this also in the final dictionary.
- Performs the same test for the target language.
- Code is written to select translation direction restrictions.
- A series of functions are defined, which will be used when generating the examples for the lemmas after the selection of the appropriate paradigm. These are:
  - `esVocalFuerte`
  - `esVocalDebil`
  - `esVocal`
  - `PosicioVocalTall`

  These functions are described later in section 6.2.3.5.
- The paradigm file is opened to display a drop-down box with the paradigms that can be selected for the source language lemma. To do this, the program has to test sequentially the paradigms defined for the part of speech of the lemma, checking whether the paradigm can be applied to the lemma in question.
- Then the same is done with the paradigms for the target language lemma.
- After the lemmas and the corresponding paradigms have been selected, examples must be generated to show how these lemmas would be inflected according to the selected paradigms. To do this, we need the root of the lemma (`raiz_LR and raiz_RL`), as well as the example endings for the selected paradigm (`paradigma_LR and paradigma_RL`); these endings are obtained from the paradigm file. Finally, a string is build containing the generated examples (`ejemplos_LR and ejemplos_RL`), and these are displayed.

- If we arrived to this screen because we were validating words (`valida=1`), then a button is added to the form, which allows us to delete the current entry if we decide not to validate it.
- If the user that arrived to this screen is a professional user, then a button is added to the form, which allows the user to select the option for the validation of words entered by non-professional users.
- Finally, after one of the action buttons located at the bottom of the form is pressed, the applicable actions are performed. If the chosen action is *"Delete"*, which can only be the case if the user is validating entries, the current entry is deleted from the file of entries made by non-professional users. If the chosen action is a confirmation (*"Go on"* button), the module `insertarPro.php` or `insertar.php` is called, depending on whether the user is professional or non-professional respectively. These modules are in charge of inserting the words in the dictionaries.

After the entry has been inserted, the page `validar.php` or the page `selecc.php` are displayed again, depending on whether the user was doing a validation process (and then *valida=1*) or a normal insertion.

6.2.3.3 multip.php

The code and behaviour of this module is the same as *selecc.php*. The only difference is that this module is designed for managing multiword units, whereas *selec.php* manages the rest of units. Therefore, the main difference is the existence of the variables `$LRcua` and `$RLcua`, which contain the invariable queue that comes after the variable part of a multiword. When the examples are displayed, besides showing the variable part inflected according to the selected paradigm, also and editable text box is displayed with the invariable queue.

When the button to continue with the insertion of the entry in the dictionaries is pressed, the module *ins_multip.php* is called instead of *insertarPro.php*.

6.2.3.4 valida.php

This module is called when a professional user presses the button *"validate pairs"*. It reads the dictionary of entries pending validation ($fichSemi) for the applicable language pair. Then, the module enters a loop that goes through this file and reads the entries one by one. With the information of

a given entry, it assigns values to a set of variables that will be used in the modules that will perform the subsequent actions. These variables are, for example:

$LRlem	$RLlem
$paradigmaLR	$paradigmaRL
$direccions	$tipo
$comentarios	$user
$geneLR	$geneRL
$numLR	$numRL
$LR	$RL

Once the appropriate values for these variables have been established, the module *selec.php* comes into action and treats the entries as if they were made by a professional user. After inserting the entries in the dictionaries by means of *insertarPro.php*, the flow returns to *valida.php*, which proceeds to the next entry to be validated.

6.2.3.5 insertarPro.php

After the lemmas have been entered and their paradigms selected in *selec.php*, this is the module that generates the corresponding `XML` entries and inserts them in the monolingual dictionaries and the bilingual dictionary.

It performs many operations similar to those performed in *selec.php*, such as generating the examples for the inflected word. Thus, firstly, it gives values to `cadFich, show, tag, tagout, nota` depending on the part of speech (`$tipus`) of the word being inserted. It assigns paths to the file location variables and defines some required functions as occurred in *selec.php*.

- `esVocalFuerte`: Returns *true* if the vowel is strong, that is, *a, e, o*.
- `esVocalDebil`: Returns *true* if the vowel is weak, that is *i, u*.
- `esVocal`: Returns *true* if the character passed as an argument is a vowel.
- `diptongo`: Returns *true* if the two letters passed as an argument make a diphthong. This will be the case when at least one of the two vowels is not strong.

- `acentuar`: It receives a text string and accentuates it according to the Spanish accentuation rules, depending on the parameter *$sigu-ienteletra*.

- `esMayuscula`: Returns *true* if the character is in upper case.

- `TieneAcento`: Returns *true* if the string has an accent.

- `acentua`: Accentuates the last accentuable vowel of a word with an open or closed accent, depending on the direction specified in the parameter $sentit.

- `PonQuitaAcento`: Inserts or removes the accent of the first string passed as an argument depending on whether the second string passed as an argument has an accent or not.

- `PosicioVocalTall`: Returns the position in the lemma ($lema) for the vowel ($vocal) that separates the root from the ending. The vowel is searched from the end to the beginning and the first occurrence of $vocal is returned.

Now, the same operations as in *selec.php* are performed. Firstly, it makes sure that the entry is not yet in the dictionaries, and then generates the examples of the word inflected according to the paradigm previously selected. After this, it builds the string with the `XML` code that is going to be inserted in the source language monolingual dictionary. With the information on the lemmas entered in *selec.php*, a text string is generated (`$cad_LR`) that contains the `XML` code for the monolingual dictionary. This string is displayed in a text box that can be manually edited. The same process is done to generate the string for the target language monolingual dictionary (`$cad_RL`) and for the bilingual dictionary (`$cad_bil`). Then, the possible comments and the name of the user making the entry are concatenated to these variables, if applicable. Finally, the form screen is completed adding the buttons for accepting, deleting and going back. The code to process each one of the possible actions is at the end of the file:

- `Insert`: In this case, it makes some character replacements so that the entry has the right format in the dictionaries, and inserts the strings `$cad_LR`, `$cad_bil`, `$cad_RL` in the source monolingual, bilingual and target monolingual dictionaries respectively (`$fich_LR`, `$fich_LRRL`, `$fich_RL`). If some error occurs when inserting the entry, a warning message is displayed. If *insertarPro.php* was called

from a word validation process (*$valida=1*), then the button "*Continue*" is inserted to continue with the validation. If this is not the case, then a button to close the window is inserted, to allow the user to end the process.

- `Delete:` It deletes the entry from the file of entries pending validation.

6.2.3.6 ins_multip.php

It performs the same actions as *insertarPro.php* but it is intended for multiword units. The main difference is the existence of two additional variables, `$LRcua` and `$RLcua`, that contain the invariable part of a multiword. When the entry is added to the dictionaries, this queue has to be inserted in the right place and the blanks have to be turned into `<b/>` tags.

6.2.3.7 insertar.php

The function of this module is very simple. It builds a text string with the information provided by *selec.php* separated by tabs. This string contains all the required information to generate a dictionary entry:

```
$LRlem.$RLlem.$paradigmaLR.$direccion.$paradigmaRL.
$tipo.$comentarios.$user.$geneLR.$geneRL.
```

This entry is saved in a file ($fichSemi) that contains the queue with the entries waiting for validation inserted by non-professional users. When a professional user wishes to validate pending entries, the *valida.php* module will read from this file.

6.2.3.8 verSemi.php

It displays the file of entries waiting for validation, in this way: it reads the file containing the entries (*$fichSemi*) and enters a loop that reads all the entries of the file. For each entry, it displays a line with the following information:

```
$LRlem $paradigmaLR $direccion $RLlem
$paradigmaRL $tipo $comentarios
```

6.2.4 Dictionary files

The files containing the entries inserted from the form are saved in `/dics`. There are here two kinds of files:

- `apertium-ll-rr.xx.webform`: This is the file that contains the entries in `XML` code, ready to be copied to the final dictionaries. The name of the file has the presented structure, where `ll-rr` are the initials of the language pair of the translator and `xx` the initials of the language of the monolingual dictionary or the languages of the bilingual dictionary referred to, as applicable. For example, the initials of the Spanish-Catalan translator are `es-ca`. For this translator, we have the Spanish monolingual (`es`), the Catalan monolingual (`ca`) and the bilingual (`es-ca`) dictionaries. Therefore, this directory will contain the following files for the Spanish-Catalan translator:

  `apertium-es-ca.es.webform`
  `apertium-es-ca.ca.webform`
  `apertium-es-ca.es-ca.webform`

- `oo-mm.semi.dic`: This is the file containing the entries pending validation for a given language pair. `oo-mm` are the initials of the pair. For example, for the Spanish-Catalan translator this file would be: `es-ca.semi.dic`

6.2.5 Paradigm files

The paradigms used for each language pair are specified in two `XML` files named `paradig.ll-rr.xx.xml`, where `xx` are the initials of the language and `ll-rr` the initials of the language pair. These files consist of a set of entries describing the paradigms or inflection models for the words of a given language. The `XML` file has the following parts:

- Head/root of the specification file.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="paradigmas.xsl"?>
<!DOCTYPE form SYSTEM "form.dtd">
<form lang="oc" langpair="oc-ca">
```

  The *lang* attribute states the initials of the language for which paradigms are specified, and the *langpair* attribute states the initials of the language pair of the translator for which the specification is made. It is required that the same directory containing the paradigm files contains the `form.dtd` file, which is the DTD specifying these files. You can find this DTD in the Appendix A.7.

- A set of elements that define the paradigms. To explain its format, we reproduce the following example:

```
<entry PoS="adj" nbr="sg_pl" gen="mf">
        <endings>
                <stem>amable</stem>
                <ending/>
                <ending>s</ending>
        </endings>
        <paradigms howmany="1">
                <par n="amable__adj"/>
        </paradigms>
</entry>
```

  Each paradigm is specified in a `<entry>` element. This element can have three attributes:

  - *PoS*: the part of speech of the paradigm. It can take the values: acr, adj, adv, noun, pname, pr, verbo. It is mandatory for any part of speech.
  - *nbr*: the numbers admitted by the paradigm. It can take the values: sg, pl, sg_pl, sp.
  - *gen*: the genders admitted by the paradigm. It can take the values: m, f, m f, mf.

  It has two more elements:

  - `endings`: the root and the endings used to select the paradigm in the form and display the inflection examples.
  - `paradigms`: specification of the paradigm/s that define the inflection of an entry. It requires the attribute *howmany* , which specifies the number of paradigms used by an entry. Each used paradigm is indicated in a line, where the name of the paradigm in the dictionary is inserted according to this format:

    `<par n="long__adj"/>`

From the `XML` paradigm file, it is necessary to generate the files directly used by the modules of the forms. Running the script `/private/gen_paradig.sh`, the process is automatically done for all the available language pairs:

```
#  cd private
# ./gen_paradig.sh
```

To add a new paradigm to the forms, an appropriate entry has to be added to the XML paradigm file, and then run the previous script to update the working files.

The automatic process can also be done manually if we do not want to update the files for all the installed language pairs. The manual generation of the working files has to be done with a XSL style sheet using the following command:

```
# xsltproc paradigmas.xsl paradigm_file.xml
                                    | ./creaparadig.awk
```

This action generates a working file for each part of speech. The generated files are saved in the directories /private/paradigmas.ll-rr. These directories contain the files with the paradigms that can be used for each language pair ll-rr and for each part of speech. Each one of these directories contain the following files:

- paradigacr_xx: paradigms for acronyms in the language xx.
- paradigadj_xx: paradigms for adjectives in the language xx.
- paradigadv_xx: paradigms for adverbs in the language xx.
- paradigcnjadv_xx: paradigms for adverbial conjunctions in the language xx.
- paradigcnjcoo_xx: paradigms for copulative conjunctions in the language xx.
- paradigcnjsub_xx: paradigms for subordinating conjunctions in the language xx.
- paradignoun_xx: paradigms for nouns in the language xx.
- paradigpname_xx: paradigms for proper nouns in the language xx.
- paradigpr_xx: paradigms for prepositions in the language xx.
- paradigverb_xx: paradigms for verbs in the language xx.

The files consist of one entry per line. Each entry contains the following information:

examples number of paradigms model_paradigms (numbers) (genders)

The separator used for the different parts of an entry is the tab.

- *Examples*: the endings that will be used to generate the examples when the user chooses this paradigm as a model for the word being inserted.
- *Number of paradigms*: the number of paradigms that are used in the dictionary to inflect this inflection model.
- *Model paradigms*: the name that have in the dictionary the paradigm/s that will be used to inflect a new entry.
- *(Numbers)*: Only completed for names, adjectives and acronyms. Refers to the grammatical number in the paradigm.
- *(Genders)*: Only completed for names, adjectives and acronyms. Refers to the grammatical gender in the paradigm.

So, therefore, for the Spanish-Catalan translator we would have the directory `/private/paradigmas.es-ca` that would contain two `XML` files: `paradig.es-ca.es.xml` and `paradig.es-ca.ca.xml`, specifying the paradigms used in each language. From these files, you may generate all the paradigm files for the language pair using the command:

```
# cd private/paradigmas.es-ca
# xsltproc ../paradigmas.xsl paradig.es-ca.es.xml
                                  | ../creaparadig.awk
# xsltproc ../paradigmas.xsl paradig.es-ca.ca.xml
                                  | ../creaparadig.awk
```

Or you can automatically generate them for all the language pairs, using:

```
# ./private/gen_paradig.sh
```

Among the generated working files, one would be, for example, a file called `paradigverb_ca` that would contain the possible verb paradigms for Catalan, where a possible line might be:

```
abra/çar /ço /ci 1 abalan/çar__vblex
```

that is generated from the `XML` entry:

```
<entry PoS="verb">
        <endings>
                <stem>abra</stem>
                <ending>çar</ending>
                <ending>ço</ending>
                <ending>ci</ending>
        </endings>
        <paradigms howmany="1">
                <par n="abalan/çar__vblex"/>
        </paradigms>
</entry>
```

6.3 Using the forms

6.3.1 Introduction

When a user wants to insert new entries in a dictionary, he/she has to use a web navigator to connect to the address where the form server has been installed; for example:

`http://xixona.dlsi.ua.es/forms`

A web page will be displayed with the portal of access to `Opentrad-Apertium Insertion Form`. The left margin contains links to get more *information* , *download* the programs and *contact* the administrator of the forms to request registration as a system user. To register as a user you will have to send an e-mail to the administrator.

To insert new words, you will have to introduce the required data in the form and press the *'Go On'* button; at this point you will have to identify yourself as a registered user, or else you will not be able to continue. There are two user registration types: you can be registered as a *professional* or as a *non-professional* user. Each mode has different functionalities, that are explained in the following section.

6.3.2 Insertion of entries

6.3.2.1 Professional mode

If you want to add a new entry to the dictionaries, you have to go to the section of the language pair you want to improve. There, you have to enter

the source language lemma and the target language lemma, and select their part of speech. Press the *Go on* button to continue.

A new window is displayed, with the lemmas and some parameters used to define the entries. If the entry already exists in one of the dictionaries, a warning message is displayed and the system automatically selects one-way translation (from left to right or vice versa). If none of the dictionaries contain the entry, the entry will be entered for both directions.

In this window you can do three actions:

- Choose the paradigm for the SL and the TL lemmas (this is mandatory, the remaining actions are not).[1]
- Select the translation direction of the entry if it is different from the automatically suggested.
- Add comments to the entry, that will be included in the final dictionary.

Once the required actions have been done, you have to press *'Go on'* if you want to confirm the entry or *'Close'* if you want to cancel the insertion operation.

The following and last screen displays the three generated `XML` entries for the SL monolingual, TL monolingual and bilingual dictionaries. These entries are displayed in three text boxes that can be edited if you want to do any change. Once you checked the entries, press the *'Insert'* button to finally insert them in the corresponding dictionaries. You can also press the *'Go back'* button to return to the previous step.

6.3.2.2 Non-professional mode

When a user enters the insertion system as a non-professional user, the word insertion mechanism is the same as for the professional user, with

[1]Choosing the paradigm has to be done very carefully. You have to choose the paradigm that describes exactly the grammatical and inflection characteristics of the inserted word. In the case of adjectives, nouns and acronyms, you have to select a paradigm that fits the inflection of the word and the genders it may present. For example, in the case of acronyms you have to consider the gender and the number admitted by each possible paradigm; the paradigm BBC, for example, is for feminine singular acronyms, whereas SA is for feminine acronyms that may have plural form. In the case of proper nouns, you have to choose a different paradigm depending on whether the word is a proper noun of a thing (e.g. a newspaper), a person or a place.

the difference that the entries will not be saved in the dictionaries generated by the forms, but will be entered in a queue of entries pending validation. The words in this queue will not be inserted in the dictionaries until a professional user validates them.

6.3.3 Validating entries

Professional users have two additional links in the screen for paradigm selection:

- *See pairs to be validated*: Selecting this option will open a screen that displays the content of the file of entries pending validation; these are the entries inserted by non-professional users. This is a merely informative screen, which can be closed pressing the *'Close'* button.

- *Validate pairs*: This option allows a professional user to validate one by one the entries waiting for validation. Selecting this button will open the screen for the selection of paradigms already described in section 6.3.2. This screen will show the data selected by the user for the added entry. Now, the professional user can modify the lemmas, delete the entry or continue with the insertion process. If the user decides to proceed with the insertion, the process is the same as for a normal insertion; only at the end, when the entry is finally added to the dictionaries of the form, the control returns to the following entry of the queue pending validation and displays it.

  This process is repeated until all the words of the queue are validated or until the process is finished by selecting *'Close'*.

Appendix A

Document Type Definitions (DTD) in XML

A.1 DTD for the format of dictionaries

Document type definition for the format of morphological, bilingual and post-generation dictionaries in XML; this definition is provided with the `apertium` package (last version) which can be downloaded from `http://www.sourceforge.net`.

The description of its elements can be found in Section 3.1.2.3.

```
<!ELEMENT dictionary (alphabet?, sdefs?,
      pardefs?, section+)>

<!ELEMENT alphabet (#PCDATA)>

<!ELEMENT sdefs (sdef+)>

<!ELEMENT sdef EMPTY>
<!ATTLIST sdef n ID #REQUIRED>

<!ELEMENT pardefs (pardef+)>

<!ELEMENT pardef (e+)>
<!ATTLIST pardef n CDATA #REQUIRED>

<!ELEMENT section (e+)>

<!ATTLIST section id ID #REQUIRED
                  type (standard|inconditional|postblank) #REQUIRED>
```

```
<!ELEMENT e (i | p | par | re)+>
<!ATTLIST e r (LR|RL) #IMPLIED
            lm CDATA #IMPLIED
            a CDATA #IMPLIED
            c CDATA #IMPLIED

<!ELEMENT par EMPTY>
<!ATTLIST par n CDATA #REQUIRED>

<!ELEMENT i (#PCDATA | b | s | g | j | a)*>

<!ELEMENT re (#PCDATA)>

<!ELEMENT p (l, r)>

<!ELEMENT l (#PCDATA | a | b | g | j | s)*>

<!ELEMENT r (#PCDATA | a | b | g | j | s)*>

<!ELEMENT a EMPTY>

<!ELEMENT b EMPTY>

<!ELEMENT g (#PCDATA | a | b | j | s)*>
<!ATTLIST g i CDATA #IMPLIED>

<!ELEMENT j EMPTY>

<!ELEMENT s EMPTY>

<!ATTLIST s n IDREF #REQUIRED>
```

A.1.1 Modification of the DTD of dictionaries for lexical selection

The DTD for the format of dictionaries has been slightly modified so that dictionaries can be used in a system that has a lexical selection module. The change only affects the <e> element and is displayed next.

```
...
```

```
<!ATTLIST e
        r (LR|RL) #IMPLIED
        lm CDATA #IMPLIED
        a CDATA #IMPLIED
        c CDATA #IMPLIED>
        i CDATA #IMPLIED
        slr CDATA #IMPLIED
        srl CDATA #IMPLIED>

  <!-- r: restriction LR: left-to-right,
                      RL: right-to-left -->
  <!-- lm: lemma -->
  <!-- a: author -->
  <!-- c: comment -->
  <!-- i: ignore ('yes') means ignore, otherwise it is not ignored) -->
  <!-- slr: translation sense when translating from left to right -->
  <!-- srl: translation sense when translating from right to left -->
...
```

A.2 DTD for the format of the tagger file

DTD that defines the format of the tagger specification file. This definition is provided with the `apertium` package (last version) which can be downloaded from `http://www.sourceforge.net`.

The description of its elements can be found in Section 3.2.2.2.

```
<!ELEMENT tagger (tagset,forbid?,enforce-rules?,preferences?)>
<!ATTLIST tagger name CDATA #REQUIRED>

<!ELEMENT tagset (def-label+,def-mult*)>

<!ELEMENT def-label (tags-item+)>
<!ATTLIST def-label name CDATA #REQUIRED
                    closed CDATA #IMPLIED>

<!ELEMENT tags-item #EMPTY>
<!ATTLIST tags-item tags CDATA #REQUIRED
                    lemma CDATA #IMPLIED>

<!ELEMENT def-mult (sequence+)>
<!ATTLIST def-mult name CDATA #REQUIRED
```

```
                    closed CDATA #IMPLIED>

<!ELEMENT sequence ((tags-item|label-item)+)>

<!ELEMENT label-item #EMPTY>
<!ATTLIST label-item label CDATA #REQUIRED>

<!ELEMENT forbid (label-sequence+)>

<!ELEMENT label-sequence (label-item+)>

<!ELEMENT enforce-rules (enforce-after+)>

<!ELEMENT enforce-after (label-set)>
<!ATTLIST enforce-after label CDATA #REQUIRED>

<!ELEMENT label-set (label-item+)>

<!ELEMENT preferences (prefer+)>

<!ELEMENT prefer EMPTY>
<!ATTLIST prefer tags CDATA #REQUIRED>
```

A.3 DTD of the structural transfer module (chunker)

DTD for the format of the structural transfer rules in the `chunker` module. This definition is provided with the `apertium` package (version 2.0) which can be downloaded from `http://www.sourceforge.net`.

Its elements are described in Section 3.5.4.

```
<!ENTITY % condition "(and|or|not|equal|begins-with|
                       ends-with|contains-substring|in)">
<!ENTITY % container "(var|clip)">
<!ENTITY % sentence "(let|out|choose|modify-case|
                      call-macro|append)">
<!ENTITY % value "(b|clip|lit|lit-tag|var|get-case-from|
                    case-of|concat)">
<!ENTITY % stringvalue "(clip|lit|var|get-case-from|
                         case-of)">
```

```
<!ELEMENT transfer (section-def-cats,
                    section-def-attrs,
                    section-def-vars,
                    section-def-lists?,
                    section-def-macros?,
                    section-rules)>

<!ATTLIST transfer default (lu|chunk) #IMPLIED>

<!ELEMENT section-def-cats (def-cat+)>

<!ELEMENT def-cat (cat-item+)>
<!ATTLIST def-cat n ID #REQUIRED>

<!ELEMENT cat-item EMPTY>
<!ATTLIST cat-item lemma CDATA #IMPLIED
                   tags CDATA #REQUIRED >

<!ELEMENT section-def-attrs (def-attr+)>

<!ELEMENT def-attr (attr-item+)>
<!ATTLIST def-attr n ID #REQUIRED>

<!ELEMENT attr-item EMPTY>
<!ATTLIST attr-item tags CDATA #IMPLIED>

<!ELEMENT section-def-vars (def-var+)>

<!ELEMENT def-var EMPTY>
<!ATTLIST def-var n ID #REQUIRED>

<!ELEMENT section-def-lists (def-list)+>

<!ELEMENT def-list (list-item+)>
<!ATTLIST def-list n ID #REQUIRED>

<!ELEMENT list-item EMPTY>
<!ATTLIST list-item v CDATA #REQUIRED>

<!ELEMENT section-def-macros (def-macro)+>

<!ELEMENT def-macro (%sentence;)+>
<!ATTLIST def-macro n ID #REQUIRED>
```

```
<!ATTLIST def-macro npar CDATA #REQUIRED>

<!ELEMENT section-rules (rule+)>

<!ELEMENT rule (pattern, action)>
<!ATTLIST rule comment CDATA #IMPLIED>

<!ELEMENT pattern (pattern-item+)>

<!ELEMENT pattern-item EMPTY>
<!ATTLIST pattern-item n IDREF #REQUIRED>

<!ELEMENT action (%sentence;)*>

<!ELEMENT choose (when+,otherwise?)>

<!ELEMENT when (test,(%sentence;)*)>

<!ELEMENT otherwise (%sentence;)+>

<!ELEMENT test (%condition;)+>

<!ELEMENT and ((%condition;),(%condition;)+)>

<!ELEMENT or ((%condition;),(%condition;)+)>

<!ELEMENT not (%condition;)>

<!ELEMENT equal (%value;,%value;)>
<!ATTLIST equal caseless (no|yes) #IMPLIED>

<!ELEMENT begins-with (%value;,%value;)>
<!ATTLIST begins-with caseless (no|yes) #IMPLIED>

<!ELEMENT ends-with (%value;,%value;)>
<!ATTLIST ends-with caseless (no|yes) #IMPLIED>

<!ELEMENT contains-substring (%value;,%value;)>
<!ATTLIST contains-substring caseless (no|yes) #IMPLIED>

<!ELEMENT in (%value;, list)>
<!ATTLIST in caseless (no|yes) #IMPLIED>
```

```
<!ELEMENT list EMPTY>
<!ATTLIST list n IDREF #REQUIRED>

<!ELEMENT let (%container;, %value;)>

<!ELEMENT append (%value;)+>
<!ATTLIST append n IDREF #REQUIRED>

<!ELEMENT out (mlu|lu|b|chunk)+>

<!ELEMENT modify-case (%container;, %stringvalue;)>

<!ELEMENT call-macro (with-param)*>
<!ATTLIST call-macro n IDREF #REQUIRED>

<!ELEMENT with-param EMPTY>
<!ATTLIST with-param pos CDATA #REQUIRED>

<!ELEMENT clip EMPTY>
<!ATTLIST clip pos CDATA #REQUIRED
               side (sl|tl) #REQUIRED
               part CDATA #REQUIRED
               queue CDATA #IMPLIED
               link-to CDATA #IMPLIED>

<!ELEMENT lit EMPTY>
<!ATTLIST lit v CDATA #REQUIRED>

<!ELEMENT lit-tag EMPTY>
<!ATTLIST lit-tag v CDATA #REQUIRED>

<!ELEMENT var EMPTY>
<!ATTLIST var n IDREF #REQUIRED>

<!ELEMENT get-case-from (clip|lit|var)>
<!ATTLIST get-case-from pos CDATA #REQUIRED>

<!ELEMENT case-of EMPTY>
<!ATTLIST case-of pos CDATA #REQUIRED
                  side (sl|tl) #REQUIRED
                  part CDATA #REQUIRED>

<!ELEMENT concat (%value;)+>
```

```
<!ELEMENT mlu (lu+)>

<!ELEMENT lu (%value;)+>

<!ELEMENT chunk (tags,(mlu|lu|b)+)>
<!ATTLIST chunk name CDATA #IMPLIED
               namefrom CDATA #IMPLIED
               case CDATA #IMPLIED>

<!ELEMENT tags (tag+)>
<!ELEMENT tag (%value;)>

<!ELEMENT b EMPTY>
<!ATTLIST b pos CDATA #IMPLIED>
```

A.4 DTD of the interchunk module

DTD for the format of the structural transfer rules in the `interchunk` module. This definition is provided with the `apertium` package (version 2.0) which can be downloaded from `http://www.sourceforge.net`.

Its elements are described in Section 3.5.4.

```
<!ENTITY % condition "(and|or|not|equal|begins-with|
                       ends-with|contains-substring|in)">
<!ENTITY % container "(var|clip)">
<!ENTITY % sentence "(let|out|choose|modify-case|
                      call-macro|append)">
<!ENTITY % value "(b|clip|lit|lit-tag|var|get-case-from|
                   case-of|concat)">
<!ENTITY % stringvalue "(clip|lit|var|get-case-from|
                         case-of)">

<!ELEMENT interchunk (section-def-cats,
                      section-def-attrs,
                      section-def-vars,
                      section-def-lists?,
                      section-def-macros?,
                      section-rules)>

<!ELEMENT section-def-cats (def-cat+)>

<!ELEMENT def-cat (cat-item+)>
<!ATTLIST def-cat n ID #REQUIRED>

<!ELEMENT cat-item EMPTY>
<!ATTLIST cat-item lemma CDATA #IMPLIED
                   tags CDATA #REQUIRED >

<!ELEMENT section-def-attrs (def-attr+)>

<!ELEMENT def-attr (attr-item+)>
<!ATTLIST def-attr n ID #REQUIRED>

<!ELEMENT attr-item EMPTY>
<!ATTLIST attr-item tags CDATA #IMPLIED>

<!ELEMENT section-def-vars (def-var+)>
```

```
<!ELEMENT def-var EMPTY>
<!ATTLIST def-var n ID #REQUIRED>

<!ELEMENT section-def-lists (def-list)+>

<!ELEMENT def-list (list-item+)>
<!ATTLIST def-list n ID #REQUIRED>

<!ELEMENT list-item EMPTY>
<!ATTLIST list-item v CDATA #REQUIRED>

<!ELEMENT section-def-macros (def-macro)+>

<!ELEMENT def-macro (%sentence;)+>
<!ATTLIST def-macro n ID #REQUIRED>
<!ATTLIST def-macro npar CDATA #REQUIRED>

<!ELEMENT section-rules (rule+)>

<!ELEMENT rule (pattern, action)>
<!ATTLIST rule comment CDATA #IMPLIED>

<!ELEMENT pattern (pattern-item+)>

<!ELEMENT pattern-item EMPTY>
<!ATTLIST pattern-item n IDREF #REQUIRED>

<!ELEMENT action (%sentence;)*>

<!ELEMENT choose (when+,otherwise?)>

<!ELEMENT when (test,(%sentence;)*)>

<!ELEMENT otherwise (%sentence;)+>

<!ELEMENT test (%condition;)+>

<!ELEMENT and ((%condition;),(%condition;)+)>

<!ELEMENT or ((%condition;),(%condition;)+)>

<!ELEMENT not (%condition;)>
```

```
<!ELEMENT equal (%value;,%value;)>
<!ATTLIST equal caseless (no|yes) #IMPLIED>

<!ELEMENT begins-with (%value;,%value;)>
<!ATTLIST begins-with caseless (no|yes) #IMPLIED>

<!ELEMENT ends-with (%value;,%value;)>
<!ATTLIST ends-with caseless (no|yes) #IMPLIED>

<!ELEMENT contains-substring (%value;,%value;)>
<!ATTLIST contains-substring caseless (no|yes) #IMPLIED>

<!ELEMENT in (%value;, list)>
<!ATTLIST in caseless (no|yes) #IMPLIED>

<!ELEMENT list EMPTY>
<!ATTLIST list n IDREF #REQUIRED>

<!ELEMENT let (%container;, %value;)>

<!ELEMENT append (%value;)+>
<!ATTLIST append n IDREF #REQUIRED>

<!ELEMENT out (b|chunk)+>

<!ELEMENT modify-case (%container;, %stringvalue;)>

<!ELEMENT call-macro (with-param)*>
<!ATTLIST call-macro n IDREF #REQUIRED>

<!ELEMENT with-param EMPTY>
<!ATTLIST with-param pos CDATA #REQUIRED>

<!ELEMENT clip EMPTY>
<!ATTLIST clip pos CDATA #REQUIRED
               part CDATA #REQUIRED>

<!ELEMENT lit EMPTY>
<!ATTLIST lit v CDATA #REQUIRED>

<!ELEMENT lit-tag EMPTY>
<!ATTLIST lit-tag v CDATA #REQUIRED>
```

```
<!ELEMENT var EMPTY>
<!ATTLIST var n IDREF #REQUIRED>

<!ELEMENT get-case-from (clip|lit|var)>
<!ATTLIST get-case-from pos CDATA #REQUIRED>

<!ELEMENT case-of EMPTY>
<!ATTLIST case-of pos CDATA #REQUIRED
                  part CDATA #REQUIRED>

<!ELEMENT concat (%value;)+>

<!ELEMENT chunk (%value;)+>

<!ELEMENT pseudolemma (%value;)>

<!ELEMENT b EMPTY>
<!ATTLIST b pos CDATA #IMPLIED>
```

A.5 DTD of the postchunk module

DTD for the format of the structural transfer rules in the `postchunk` module. This definition is provided with the `apertium` package (version 2.0) which can be downloaded from `http://www.sourceforge.net`.

Its elements are described in Section 3.5.4.

```
<!ENTITY % condition "(and|or|not|equal|begins-with|
                       ends-with|contains-substring|in)">
<!ENTITY % container "(var|clip)">
<!ENTITY % sentence "(let|out|choose|modify-case|
                      call-macro|append)">
<!ENTITY % value "(b|clip|lit|lit-tag|var|get-case-from|
                   case-of|concat)">
<!ENTITY % stringvalue "(clip|lit|var|get-case-from|
                         case-of)">

<!ELEMENT postchunk (section-def-cats,
                     section-def-attrs,
                     section-def-vars,
                     section-def-lists?,
                     section-def-macros?,
                     section-rules)>

<!ELEMENT section-def-cats (def-cat+)>

<!ELEMENT def-cat (cat-item+)>
<!ATTLIST def-cat n ID #REQUIRED>

<!ELEMENT cat-item EMPTY>
<!ATTLIST cat-item name CDATA #REQUIRED>

<!ELEMENT section-def-attrs (def-attr+)>

<!ELEMENT def-attr (attr-item+)>
<!ATTLIST def-attr n ID #REQUIRED>

<!ELEMENT attr-item EMPTY>
<!ATTLIST attr-item tags CDATA #IMPLIED>

<!ELEMENT section-def-vars (def-var+)>

<!ELEMENT def-var EMPTY>
```

```
<!ATTLIST def-var n ID #REQUIRED>

<!ELEMENT section-def-lists (def-list)+>

<!ELEMENT def-list (list-item+)>
<!ATTLIST def-list n ID #REQUIRED>

<!ELEMENT list-item EMPTY>
<!ATTLIST list-item v CDATA #REQUIRED>

<!ELEMENT section-def-macros (def-macro)+>

<!ELEMENT def-macro (%sentence;)+>
<!ATTLIST def-macro n ID #REQUIRED>
<!ATTLIST def-macro npar CDATA #REQUIRED>

<!ELEMENT section-rules (rule+)>

<!ELEMENT rule (pattern, action)>
<!ATTLIST rule comment CDATA #IMPLIED>

<!ELEMENT pattern (pattern-item+)>

<!ELEMENT pattern-item EMPTY>
<!ATTLIST pattern-item n IDREF #REQUIRED>

<!ELEMENT action (%sentence;)*>

<!ELEMENT choose (when+,otherwise?)>

<!ELEMENT when (test,(%sentence;)*)>

<!ELEMENT otherwise (%sentence;)+>

<!ELEMENT test (%condition;)+>

<!ELEMENT and ((%condition;),(%condition;)+)>

<!ELEMENT or ((%condition;),(%condition;)+)>

<!ELEMENT not (%condition;)>

<!ELEMENT equal (%value;,%value;)>
```

```
<!ATTLIST equal caseless (no|yes) #IMPLIED>

<!ELEMENT begins-with (%value;,%value;)>
<!ATTLIST begins-with caseless (no|yes) #IMPLIED>

<!ELEMENT ends-with (%value;,%value;)>
<!ATTLIST ends-with caseless (no|yes) #IMPLIED>

<!ELEMENT contains-substring (%value;,%value;)>
<!ATTLIST contains-substring caseless (no|yes) #IMPLIED>

<!ELEMENT in (%value;, list)>
<!ATTLIST in caseless (no|yes) #IMPLIED>

<!ELEMENT list EMPTY>
<!ATTLIST list n IDREF #REQUIRED>

<!ELEMENT let (%container;, %value;)>

<!ELEMENT append (%value;)+>
<!ATTLIST append n IDREF #REQUIRED>

<!ELEMENT out (b|lu|mlu)+>

<!ELEMENT modify-case (%container;, %stringvalue;)>

<!ELEMENT call-macro (with-param)*>
<!ATTLIST call-macro n IDREF #REQUIRED>

<!ELEMENT with-param EMPTY>
<!ATTLIST with-param pos CDATA #REQUIRED>

<!ELEMENT clip EMPTY>
<!ATTLIST clip pos CDATA #REQUIRED
               part CDATA #REQUIRED>

<!ELEMENT lit EMPTY>
<!ATTLIST lit v CDATA #REQUIRED>

<!ELEMENT lit-tag EMPTY>
<!ATTLIST lit-tag v CDATA #REQUIRED>

<!ELEMENT var EMPTY>
```

```
<!ATTLIST var n IDREF #REQUIRED>

<!ELEMENT get-case-from (clip|lit|var)>
<!ATTLIST get-case-from pos CDATA #REQUIRED>

<!ELEMENT case-of EMPTY>
<!ATTLIST case-of pos CDATA #REQUIRED
                  part CDATA #REQUIRED>

<!ELEMENT concat (%value;)+>

<!ELEMENT mlu (lu+)>

<!ELEMENT lu (%value;)+>

<!ELEMENT b EMPTY>
<!ATTLIST b pos CDATA #IMPLIED>
```

A.6 DTD for the format specification rules

DTD for the format specification rules. This definition can be downloaded from the web page `http://cvs.sourceforge.net/viewcvs.py/apertium/apertium/apertium/format.dtd`.

Its elements are described in Section 3.6.2.

```
<!ELEMENT format (options,rules)>
<!ATTLIST format name CDATA #REQUIRED>

<!ELEMENT options (largeblocks, input, output,
                   escape-chars, space-chars, case-sensitive)>

<!ELEMENT largeblocks EMPTY>
<!ATTLIST largeblocks size CDATA #REQUIRED>

<!ELEMENT input EMPTY>
<!ATTLIST input zip-path CDATA #IMPLIED
               encoding CDATA #REQUIRED>

<!ELEMENT output EMPTY>
<!ATTLIST output zip-path CDATA #IMPLIED
                encoding CDATA #REQUIRED>

<!ELEMENT escape-chars EMPTY>
<!ATTLIST escape-chars regexp CDATA #REQUIRED>

<!ELEMENT space-chars EMPTY>
<!ATTLIST space-chars regexp CDATA #REQUIRED>

<!ELEMENT case-sensitive EMPTY>
<!ATTLIST case-sensitive value (yes|no) #REQUIRED>

<!ELEMENT rules (format-rule|replacement-rule)+>

<!ELEMENT format-rule (begin-end|(begin,end))>
<!ATTLIST format-rule eos (yes|no) #IMPLIED
                      priority CDATA #REQUIRED>

<!ELEMENT begin-end EMPTY>
<!ATTLIST begin-end regexp CDATA #REQUIRED>

<!ELEMENT begin EMPTY>
```

```
<!ATTLIST begin regexp CDATA #REQUIRED>

<!ELEMENT end EMPTY>
<!ATTLIST end regexp CDATA #REQUIRED>

<!ELEMENT replacement-rule (replace+)>
<!ATTLIST replacement-rule regexp CDATA #REQUIRED>

<!ELEMENT replace EMPTY>
<!ATTLIST replace source CDATA #REQUIRED
                  target CDATA #REQUIRED
                  prefer (yes|no) #IMPLIED>
```

A.7 DTD for the form paradigms

DTD for the format of the paradigm files used in the forms. This definition is included in the package `apertium-lexical-webform`.

```
<!ELEMENT form (entry)+>

<!ATTLIST form
        lang CDATA #REQUIRED
        langpair CDATA #REQUIRED>

<!ELEMENT entry (endings, paradigms)+>

<!ATTLIST entry
        PoS CDATA #REQUIRED
        nbr CDATA #IMPLIED
        gen CDATA #IMPLIED>

<!ELEMENT endings (stem, ending+)>

<!ELEMENT stem (#PCDATA)>

<!ELEMENT ending (#PCDATA)>

<!ELEMENT paradigms (par+)>

<!ATTLIST paradigms howmany CDATA #REQUIRED>

<!ELEMENT par EMPTY>

<!ATTLIST par n CDATA #REQUIRED>
```

Appendix B

Grammatical symbols used in the modules

B.1 Grammatical symbols used in dictionaries

B.1.1 List of symbols

aa	adjective-adjective (function of relative pronoun)
acr	acronym
al	others (for proper nouns)
an	adjective-noun (function of relative pronoun)
ant	antroponym
cni	conditional
cnjadv	adverbial conjunction
cnjcoo	co-ordinating conjunction
cnjsub	subordinating conjunction
def	definite
dem	demonstrative
det	determiner
detnt	neuter determiner
enc	enclitic
f	feminine
fti	future indicative
fts	future subjunctive
ger	gerund
ifi	perfect preterite
ij	interjection
imp	imperative
ind	indefinite
inf	infinitive

itg	interrogative
loc	locative
lpar	([
lquest	¿
m	masculine
mf	masculine-feminine
n	noun
nn	noun-noun (function of relative pronoun)
np	proper noun
nt	neuter
num	numeral - number
p1	first person
p2	second person
p3	third person
pii	imperfect preterite indicative
pis	imperfect preterite subjunctive
pl	plural
pos	possessive
pp	participle
pr	preposition
preadv	preadverb
predet	predeterminer
pri	present indicative
prn	pronoun
pro	proclitic
prs	present subjunctive
ref	reflexive
rel	relative
rpar	)]
sent	. ? ; : !
sg	singular
sp	singular-plural
sup	superlative
tn	tonic
vaux	auxiliary verb
vbhaver	verb *to have*
vblex	lexical verb
vbmod	modal verb
vbser	verb *to be*

B.1.2 Specification of lexical forms

Order for the placement of grammatical symbols in the morphological dictionaries of this system (from left to right in the table). The examples in brackets are from Spanish.

Common adjectives (difícil, rojo)	**PoS**	**Gender**	**Number**
	adj	m	sg
		f	pl
		mf	sp
Interrogative, possessive, indetermined and superlative adjectives (qué, tus, otra, buenísimo)	**PoS**	**Type**	**Gender**	**Number**
	adj	itg	m	sg
		pos	f	pl
		ind	mf	sp
		sup		
Adverbs (siempre, mañana)	**PoS**			
	adv			
Preadverbs (muy, tan)	**PoS**			
	preadv			
Interrogative adverbs (dónde)	**PoS**	**Type**		
	adv	itg		
Adverbial conjunctions (que, así como)	**PoS**			
	cnjadv			
	cnjcoo			
	cnjsub			
Determiners (el, uno, este, mi)	**PoS**	**Type**	**Gender**	**Number**
	det	def	m	sg
		ind	f	pl
		dem	mf	sp
		pos		
Neuter determiners (lo)	**PoS**			
	detnt			
Predeterminers (todos)	**PoS**	**Gender**	**Number**
	predet	m	sg
		f	pl
		nt	sp
Interjections (hola)	**PoS**		
	ij		
Common nouns (casa, perro)	**PoS**	**Gender**	**Number**
	n	m	sg
	n	f	pl
	n	mf	sp
Proper nouns (Pedro, Londres)	**PoS**	**Type**	
	np	ant	
		loc	
		al	

Acronyms (IRPF, INEM)	**PoS**	**Type**	**Gender**	**Number**	
	n	acr	m f mf	sg pl sp	
Numerals (tres)	**PoS**	**Gender**	**Number**		
	num	m f mf	sg pl sp		
Prepositions (de, por)	**PoS**				
	pr				
Interrogative pronouns (quién, qué)	**PoS**	**Type**	**Gender**	**Number**	
	prn	itg	m f	sg pl	
Enclitic, proclitic and tonic personal pronouns (yo, vosotros, ayudar**te**, **te** ayudo)	**PoS**	**Type**	**Person**	**Gender**	**Number**
	prn	enc pro tn	p1 p2 p3	m f mf nt	sg pl sp
Procl. reflexive pron. (se):	prn	pro	ref	p3	mf sp
Tonic reflex. pron. (si):	prn	tn	ref	p3	mf sp
Tonic possessive pron. (mío, suyo)	**PoS**	**Type**	**Subtype**	**Gender**	**Number**
	prn	tn	pos	m f	sg pl
Other tonic pronouns (aquella, nadie, otro)	**PoS**	**Type**	**Gender**	**Number**	
	prn	tn	m f mf nt	sg pl sp	
Pronominal and adjectival relatives (que, cuyo)	**PoS**	**Type**	**Gender**	**Number**	
	rel	nn an aa	m f f	sg pl pl	
Adverbial relatives (como, donde)	**PoS**	**Type**			
	rel	adv			
Verbs (personal forms) (subo, vamos)	**Type**	**Tense and mode**	**Person**	**Number**	
	vblex vbser vbhaver vbmod	cni fti fts ifi imp pii pis pri prs	p1 p2 p3	sg pl	
Verbs (infinitive and gerund) (cantar, buscando)	**Type**	**Form**			
	vblex vbser vbhaver vbmod	inf ger			
Verbs (participle) (dormido, cansadas)	**Type**	**Form**	**Gender**	**Number**	
	vblex vbser vbhaver vbmod	pp	m f	sg pl	

B.2 Categories used in the part-of-speech tagger

B.2.1 Spanish tagger

These are the categories or coarse tags used by the Spanish part-of-speech tagger.

Tag	Description	Closed	Examples
	Simple tags		
PARAPR	Lexicalization of *para* as a preposition	Yes	
PARAVBPRI	Lexicalization of *para* as a lexical verb in present indicative	Yes	
PARAVBIMP	Lexicalization of *para* as a lexical verb in imperative	Yes	
QUECNJ	Lexicalization of *que* as a conjunction	Yes	
QUEREL	Lexicalization of *que* as a relative pronoun	Yes	
COMOPR[1]	Lexicalization of *como* as a preposition	Yes	
COMOREL	Lexicalization of *como* as a relative pronoun	Yes	
COMOVB	Lexicalization of *como* as a lexical verb in present indicative	Yes	
MASADV	Lexicalization of *más/menos* as an adverb	Yes	
MASADJ	Lexicalization of *más/menos* as an adjective	Yes	
MASNP	Lexicalization of *Más* as a proper noun	Yes	
ALGOADV	Lexicalization of *algo* as an adverb	Yes	
ACRONIMOM	Acronym	No	BCH
ACRONIMOF	Acronym	No	ONU
ACRONIMOMF	Acronym	No	ATS
INTNOM	Interrogative pronoun	Yes	quién, cuál
ADJINT	Interrogative adjective	Yes	cuánto, qué
INTADV	Interrogative adverb	Yes	cuándo, dónde
PREADV	Adverb that can precede another adverb or an adjective	Yes	muy, bien, mal
ADV	Adverb	No	nunca, ahí
CNJSUBS	Subordinating conjunction	Yes	que
CNJCOORD	Co-ordinating conjunction	Yes	y, pero
CNJADV	Adverbial conjunction	No	si
DETNT	Neuter determiner	Yes	lo
DETM	Determiner	Yes	el, un
DETF	Determiner	Yes	la, una
DETMF	Determiner	Yes	cada
INTERJ	Interjection	No	ojalá, hola
NOM	Noun	No	casa, coche
ANTROPONIM	Proper noun for person	No	Fernando
TOPONIM	Proper noun for place	No	Alicante

[1]The morphological analyser considers that *como* can be a preposition since it can be replaced with *en calidad de* in some contexts (e.g.- *'Os hablo como director de la película'*).

Tag	**Description**	**Closed**	**Examples**
NPALTRES	Other proper nouns	No	Linux, Seat
NUM	Numeral	Yes	tres, cuatro
PREDETNT	Neuter predeterminer	Yes	todo
PREDET	Predeterminer	Yes	toda
PREP	Preposition	Yes	ante, desde
PRNTNNT	Neuter tonic pronoun	Yes	algo, esto
PRNTN	Tonic pronoun	Yes	ambos, nadie
PRNENCREF	Reflexive enclitic pronoun	Yes	se
PRNPROREF	Reflexive proclitic pronoun	Yes	se
PRNENC	Enclitic pronoun	Yes	me, nos
PRNPRO	Proclitic pronoun	Yes	le, te
VLEXINF	Lexical verb in infinitive	No	cantar, reír
VLEXGER	Lexical verb in gerund	No	hablando
VLEXPARTPI	Lexical verb in participle	No	dicho, cantado
VLEXPFCI	Lexical verb in present, future or conditional indicative	No	digo, diré, diría
VLEXIPI	Lexical verb in imperfect preferite or perfect preterite indicative	No	cantaba, dijo
VLEXSUBJ	Lexical verb in subjunctive	No	hablase, dijeramos
VLEXIMP	Lexical verb in imperative	No	canta, comed
VSERINF	Verb *to be* in infinitive	Yes	ser
VSERGER	Verb *to be* in gerund	Yes	siendo
VSERPARTPI	Verb *to be* in participle	Yes	sido
VSERPFCI	Verb *to be* in present, future or conditional indicative	Yes	soy, seré, sería
VSERIPI	Verb *to be* in imperfect preterite or perfect preterite indicative	Yes	era, fui
VSERSUBJ	Verb *to be* in subjunctive	Yes	fueras
VSERIMP	Verb *to be* in imperative	Yes	sé
VHABERINF	Verb *to have* in infinitive	Yes	haber
VHABERGER	Verb *to have* in gerund	Yes	habiendo
VHABERPARTPI	Verb *to have* in participle	Yes	habido
VHABERPFCI	Verb *to have* in present, future or conditional indicative	Yes	hay, habrán, habría
VHABERIPI	Verb *to have* in imperfect preterite or perfect preterite indicative	Yes	había, hubo
VHABERSUBJ	Verb *to have* in subjunctive	Yes	hubieran
VMODALINF	Modal verb in infinitive	Yes	deber, poder
VMODALGER	Modal verb in gerund	Yes	debiendo
VMODALPARTPI	Modal verb in participle	Yes	podido
VMODALPFCI	Modal verb in present, future or conditional indicative	Yes	puede, deberá, podría
VMODALIPI	Modal verb in imperfect preterite or perfect preterite indicative	Yes	podía, debió
VMODALSUBJ	Modal verb in subjunctive	Yes	pudiese, debiéramos
VMODALIMP	Modal verb in imperative	Yes	poded, debed

Tag	Description	Closed	Examples
ADJM	Adjective	No	gracioso
ADJF	Adjective	No	graciosa
ADJMF	Adjective	No	inteligente
ADJPOS	Possessive adjective	Yes	mío
REL	Relative pronoun	Yes	quien, cuya
RELADV	Adverbial relative	Yes	cuando, donde
Compound tags			
PREPDET	Contraction of preposition and determiner	Yes	del, al
PRCNJ	Multiword made of preposition and conjunction	Yes	a que
PRREL	Multiword made of preposition and relative	Yes	en que
INFLEXPRNENC	Lexical verb in infinitive with enclitics	No	dármelo, cantarlo
GERLEXPRNENC	Lexical verb in gerund with enclitics	No	cantándosela
IMPLEXPRNENC	Lexical verb in imperative with enclitics	No	dímelo
INFSERPRNENC	Verb *to be* in infinitive with enclitics	Yes	serlo
GERSERPRNENC	Verb *to be* in gerund with enclitics	Yes	siéndolo
IMPSERPRNENC	Verb *to be* in imperative with enclitics	Yes	sedlo
INFHABPRNENC	Verb *to have* in infinitive with enclitics	Yes	habérsela
GERHABPRNENC	Verb *to have* in gerund with enclitics	Yes	habiéndole
INFMODPRNENC	Modal verb in infinitive with enclitics	Yes	poderla, deberlo
GERMODPRNENC	Modal verb in gerund with enclitics	Yes	debiéndosela
IMPMODPRNENC	Modal verb in imperative with enclitics	Sí	debédmela
Other tags			
LQUEST	Opening question mark		¿
LPAR	Opening parenthesis or square bracket		(, [
RPAR	Closing parenthesis or square bracket		),]
CM	Comma		,
SENT	Sentence end character		., :, ;, ?, !

B.2.2 Catalan tagger

Due to the similarity of the Catalan tagger categories and the Spanish ones, we list here only the tags that are new or different in the Catalan tagger.

Tag	Description	Closed	Examples
Simple tags			
MOLTADV	Lexicalization of *molt/gaire* as an adverb	Yes	
MOTLPREADV	Lexicalization of *molt/gaire* as an adverb	Yes	
VOLERMOD	Lexicalization of *voler* as a modal verb	Yes	
VOLERLEX	Lexicalization of *voler* as a lexical verb	Yes	
VA	Lexicalization of *va* as a form of the verb *anar*	Yes	

B.2.3 Galician tagger

Due to the similarity of the Galician tagger categories and the Spanish ones, we list here only the tags that are new or different in the Galician tagger.

Tag	Description	Closed	Examples
	Simple tags		
VBIRNPS	Lexicalization of *to go* in infinitive and gerund	Yes	
VBIRPARTPI	Lexicalization of *to go* in participle	Yes	
VBIRPS	Lexicalization of *to go* in the personal forms of indicative and subjunctive	Yes	
VBIRIMP	Lexicalization of *to go* in imperative	Yes	
VHABERNPS	Lexicalization of *to have* in infinitive and gerund	Yes	
VHABERPARTPI	Lexicalization of *to have* in participle	Yes	
VHABERPS	Lexicalization of *to have* in the personal forms of indicative and subjunctive	Yes	
VHABERIMP	Lexicalization of *to have* in imperative	Yes	
APREP	Lexicalization of *a* as a preposition	Yes	
VLEXNPS	Lexical verb: infinitive and gerund	No	achegar, achegándomos
VLEXPS	Lexical verb: personal forms in indicative	No	achegue, achegaré
VSERNPS	Verb *to be*: infinitive and gerund	Yes	ser, seres
VSERPS	Verb *to be*: personal forms in indicative	Yes	fosen, es
	Compound tags		
PREPDETM	Contraction of preposition and masculine determiner	Yes	do, ao
PREPDETF	Contraction of preposition and feminine determiner	Yes	da, ás
PREPDETN	Contraction of preposition and neuter determiner	Yes	do
PREPDETDET	Contraction of preposition and two determiners	Yes	destoutro
PREPPRTNNT	Contraction of preposition and neuter tonic pronoun	Yes	daquilo
PREPPRNTN	Contraction of preposition and tonic pronoun	Yes	daqueloutra
PREPTNTN	Contraction of preposition and two tonic pronouns	Yes	nestoutra
PREPNUM	Contraction of preposition and numeral	Yes	dunha
PREDETDET	Contraction of predeterminer and		

Tag	Description	Closed	Examples
	determiner	Yes	tódalas
INTADVDET	Contraction of adverbial interrogative and determiner	Yes	u-la
DETDETM	Contraction of two masculine determiners	Yes	ámbolos
DETDETF	Contraction of two feminine determiners	Yes	ámbalas
PRNPRN	Contraction of two tonic pronouns	Yes	esoutra
PRNPRN	Contraction of two proclitic pronouns	Yes	chas
CNJCDET	Contraction of co-ordinating conjunction and determiner	Yes	maila
CNJSUB	Contraction of subordinating conjunction and determiner	Yes	cás

Appendix C

Abbreviations used in the text

ANSI American National Standards Institute; when used informally in the expression *ANSI text*, it refers to a text encoded in any of the encodings of one byte per character defined in the standard ISO-8859 [1].

ca ISO 639 two-letter code[1] for Catalan

DTD Document type definition in XML

es ISO 639 two-letter code for Spanish

eu ISO 639 two-letter code for Basque

LF Lexical form (see page 7)

TLLF Target language lexical form

SLLF Source language lexical form

SF Surface form (see page 7)

gl ISO 639 two-letter code for Galician

pt ISO 639 two-letter code for Portuguese

HTML Hypertext markup language

TL Target language

SL Source language

[1] See `http://www.w3.org/WAI/ER/IG/ert/iso639.htm`

RTF Rich text format

MT Machine translation

XML Extensible markup language

POS Part of speech

Bibliography

[1] Unicode. `http://www.unicode.org`.

[2] R. Canals-Marote, A. Esteve-Guillén, A. Garrido-Alenda, M. Guardiola-Savall, A. Iturraspe-Bellver, S. Monserrat-Buendia, S. Ortiz-Rojas, H. Pastor-Pina, P.M. Perez-Antón, and M.L. Forcada. El sistema de traducción automática castellano-catalán interNOSTRUM. *Procesamiento del Lenguaje Natural*, 27:151–156, 2001. XVII Congreso de la Sociedad Española de Procesamiento del Lenguaje Natural, Jaén, Spain, 12-14.09.2001.

[3] D. Cutting, J. Kupiec, J. Pedersen, and P. Sibun. A practical part-of-speech tagger. In *Third Conference on Applied Natural Language Processing. Association for Computational Linguistics. Proceedings of the Conference.*, pages 133–140, Trento, Italia, 31 marzo–3 abril 1992.

[4] A. Garrido, A. Iturraspe, S. Montserrat, H. Pastor, and M. L. Forcada. A compiler for morphological analysers and generators based on finite-state transducers. *Procesamiento del Lenguaje Natural*, (25):93–98, 1999.

[5] Alicia Garrido-Alenda and Mikel L. Forcada. Morphtrans: un lenguaje y un compilador para especificar y generar módulos de transferencia morfológica para sistemas de traducción automática. *Procesamiento del Lenguaje Natural*, 27:157–164, 2001.

[6] Alicia Garrido-Alenda, Mikel L. Forcada, and Rafael C. Carrasco. Incremental construction and maintenance of morphological analysers based on augmented letter transducers. In *Proceedings of TMI 2002 (Theoretical and Methodological Issues in Machine Translation, Keihanna/Kyoto, Japan, March 2002)*, pages 53–62, 2002.

[7] Alicia Garrido-Alenda, Patrícia Gilabert Zarco, Juan Antonio Pérez-Ortiz, Antonio Pertusa-Ibáñez, Gema Ramírez-Sánchez, Felipe

Sánchez-Martínez, Miriam A. Scalco, and Mikel L. Forcada. Shallow parsing for portuguese-spanish machine translation. In A. Branco, A. Mendes, and R. Ribeiro, editors, *TASHA 2003: Workshop on Tagging and Shallow Processing of Portuguese*, pages 21–24, October 2003.

[8] N. Ide. *The XML Framework and Its Implications for the Development of Natural Language Processing Tools*. Luxembourg, 2000.

[9] M.E. Lesk. Lex — a lexical analyzer generator. Technical Report Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[10] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[11] Sergio Ortiz-Rojas, Mikel L. Forcada, and Gema Ramírez-Sánchez. Construcción y minimización eficiente de transductores de letras a partir de diccionarios con paradigmas. *Procesamiento del Lenguaje Natural*, (25):51–57, 2005.

[12] Ferran Pla and Antonio Molina. Improving part-of-speech tagging using lexicalized HMMs. *Journal of Natural Language Engineering*, 10(2):167–189, June 2004.

[13] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[14] E. Roche and Y. Schabes. *Introduction*. MIT Press, Cambridge, Massachusetts, 1997.

[15] E. Roche and Y. Schabes. Introduction. In E. Roche and Y. Schabes, editors, *Finite-State Language Processing*, pages 1–65. MIT Press, Cambridge, Mass., 1997.

[16] J. L. A. van de Snepscheut. *What computing is all about*. Springer-Verlag, New York, 1993.

[17] Patrícia Gilabert Zarco, Javier Herrero-Vicente, Sergio Ortiz-Rojas, Antonio Pertusa-Ibáñez, Gema Ramírez-Sánchez, Felipe Sánchez-Martínez, Marcial Samper-Asensio, Míriam A. Scalco, and Mikel L. Forcada. Construcción rápida de un sistema de traducción automática español-portugués partiendo de un sistema español–catalán. *Procesamiento del Lenguaje Natural*, (32):279–285, 2003. (Actas del XIX congreso de la Sociedad Española de Procesamiento del Lenguaje Natural).