A framework for representing lexical resources

Fabrice Issac
LDI
Université Paris 13

Abstract

Our goal is to propose a description model for the lexicon. We describe a software framework for representing the lexicon and its variations called Proteus. Various examples show the different possibilities offered by this tool. We conclude with a demonstration of the use of lexical resources in complex, real examples.

1 Introduction

Natural language processing relies as well on methods, algorithms, or formal models as on linguistic resources. Processing textual data involves a classic sequence of steps : it begins with the normalisation of data and ends with their lexical, syntactic and semantic analysis. Lexical resources are the first to be used and must be of excellent quality.

Traditionally, a lexical resource is represented as a list of inflected forms that are projected on a text. However, this type of resource can not take into account linguistic phenomena, as each unit of information is independent. This results in a number of problems regarding the improvement or review of the resource. On the other hand some languages such as Arabic, because of the potential large lexicon, lends itself less easily to this kind of manipulation.

Our goal is to propose a model for the description of the lexicon. After presenting the existing theory and software tools, we introduce a software framework called Proteus, capable of representing the lexicon and its variations. The different possibilities offered by this tool will be illustrated through various examples. We conclude with a demonstration of the use of lexical resources in different languages.

2 Context

Whatever the writing system of a language (logographic, syllabic or alphabetic), it seems that the word is a central concept. Nonetheless, the very definition of a word is subject to variation depending on the language studied. For some Asian languages such as Mandarin or Vietnamese, the notion of word delimiter does not exist ; for others, such as French or English, the space is a good indicator. Likewise, for some languages, prepositions are included in words, while for others they form a separate unit.

Languages can be classified according to their morphological mechanisms and their complexity ; for instance, the morphological systems of French or English are relatively simple compared to that of Arabic or Greek.

There are two main branches of morphology, inflectional or grammatical morphology and lexical morphology. The first one deals with context-related variations, as the rules of agreement in gender and number or the conjugation of verbs. The second one concerns word formation, generally involving the association of a lexeme to prefixes or suffixes.

3 Tagging

Text tagging consists in adding one or more information units to a group of characters : the token. This association is firstly performed in a context-free way, that is to say considering only the token, and secondly by increasing the context size : the tagging process is subsequently repeated in or-

Coling 2010: Poster Volume, pages 490–497,
Beijing, August 2010

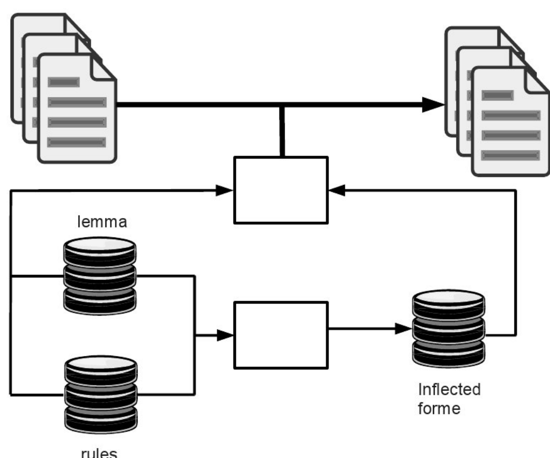

Figure 1: tagging schema

der to merge multiple tokens. Token merging applies to polylexical units, syntactic or para-textual structures.

We distinguish two main types of resources that can be projected on raw texts in order to enrich them. The first of these resources is a set of inflected forms associated with a number of information units (in the example below, the lemma and the morphosyntactic annotation of each form) :

```
abyssal   abyssal  A--ms
abysses   abysse   N-mp
```

The projection of this type of resources in textual corpora is quite simple. After identifying a token, the program only needs to check if the token is included in the resource and add the information units associated with it.

The second type of resources contains a set of rules and a set of canonical forms (usually the lemma but not necessarily). These sets are used jointly, to produce all the inflected forms or to analyse the tokens. Analysis consists in determining, for a given inflected form, which rule was used, on which canonical form, in order to generate it. Then the information to be associated with the inflected form is related to the rule found.

Diagram 1 presents the place of different resources in the tagging process.

4 Tools and resources

Several concepts are related to the use of lexical resources ; here we provide some examples of tools, theoretical as well as computational.

- resources in the form of a frozen list: Morphalou (Romary et al., 2004), Morfetik (Buvet et al., 2007), Lexique3 (New, 2006) ;
- lexical representation formalisms: DATR (Evans and Gazdar, 1996) ;
- inflections' parsers: Flemm (Namer, 2000) ;
- complete software platforms: Nooj (Silberztein, 2005), Unitex (Paumier, 2002) ;
- lexicon acquisition: lefff (Sagot et al., 2006).

4.1 Frozen resources

If this kind of resource is directly used in the tagging process, it raises many maintenance issues. Moreover, in the case of languages with rich morphology, the number of elements becomes too large. These lists are most often the result of inflection engines that use canonical forms and inflection rules to generate the inflected forms.

4.2 Hierarchical lexical representation formalisms

The goal of this type of formalisms is to represent the most general information at the top-level of a hierarchy. There is an inheritance mechanism to transmit, specify, and if necessary, delete information along the structure. It is possible to group under one tag a set of morphological phenomena and their exceptions. Multiple inheritance allows a node to inherit from several different hierarchies.

4.3 Inflections' parsers

They propose a morphological analysis for a given inflected form : they try to apply the derivation rules backwards and test whether the result obtained corresponds to an attested canonical form. The use of canonical forms is optional ; it provides an analysis for lexical neologisms but can cause incorrect results (*Hundred* is not the past participle of *hundre*).

4.4 Software platforms

Unitex / Intex / NooJ are complete environments for performing linguistic analysis on documents. They are able to project dictionaries on texts and to query them. They offer a set of tools for managing dictionaries, both lexical and inflectional. NooJ is the successor of Intex ; among the new features, is the redesign of the architecture of the dictionaries. It proposes handling simple and compound words in a single way. The method is a mix of manipulation of characters and use of morphological grammars.

The inflexion mechanism is based on classic character handling operators as well as on word manipulation operators. Here is the list of some operators:
<B> delete last character
<D> duplicate last character
<L> go left
<R> go right
<N> go to the end of next word form
<P> go to the end of previous word form
<S> delete next character

5 Representation and structuring of inflections: the Proteus model

We introduce a framework capable to represent and structure inflections efficiently, not only in terms of resource creation, but in terms of linguistic consistency too. At the inflection level we propose a simple multilingual mechanism for simple and compound forms. It is used both to generate simple forms and to analyse them. Regarding the lexicon, the model allows for clusters.

We distinguish three levels:

- the inflection level: determine how to produce a derived form from a base form ; the atomic processing unit here is the character (*i.e. local transformation*).
- the conjugation level: determine how to organise family rules effectively in order to avoid redundancy ; the atomic processing unit here is the transformation rule.
- the word level: once the derived form is produced, determine which operation is required to validate the form against non-morphological rules ; the processing unit here is the token (*i.e.* global transformation).

The model was developed to meet the following objectives:

1. A verbatim description of a language does not allow for the analysis of unknown words even if their inflection is regular. We must therefore develop a mechanism that we can use for both analysis and generation. Then we will be able to analyse not only known words but also neologisms.
2. In a lexical database, where, for French, the number of elements reaches one million, the presence of an error is always possible, even inevitable. We must therefore consider an effective maintenance procedure : a dictionary of lemmas linked to a dictionary of inflections and not a read-only resource containing all inflected forms.
3. The concept of word is so complex that we cannot limit a resource to simple words. The model must integrate the management of both simple and compound words. The only limit we set is syntax: the management of idioms, even if it is fundamental, requires the implementation of other tools.
4. The concept of inflection varies depending on the language. We must build a system capable of dealing with all types of affixation (prefixation, suffixation or infixation). The treatment of Arabic is from this point of view a good indicator since it uses all three types of affixation.
5. An inflection rule, applied to a canonical form, is never completely autonomous ; it is part of a group. For instance, we group together all the inflections of a verb type, for all tenses.
6. The transformation is not limited to morphological changes. For instance, phonological phenomena can occur too. More generally there are treatments that cannot be modelled on simple rules.

7. The proposed model is based on a set of simple tools, it is able to easily integrate third-party applications and allows use of dictionaries built in another environment.

5.1 Inflection description

Let f be the inflexion function and f^{-1} its inverse function, then

$$f(canonicalform, code) = inflectedform$$

$$f^{-1}(inflectedform, code) = canonicalform$$

By simplifying the model to the extreme, we use of a rule that generates an inflected form from its lemma. The form is represented as a list of caracters : (i) it shifts characters from the list to a stack or vice versa (ii) and deletes or inserts characters in the list. By default, the operations apply to characters placed at the end of the list of characters or, depending on the operator, at the top of the stack. Most operators allow for the application of the inverse function for the analysis of an inflection. Due to the operator based construction, the function has the following property:

let c_1, c_2 be valid code and x a character string, then

$$f(f(x, c_1), c_2) = f(x, c_1 \bullet c_2)$$

We now present the different operators. They must be sufficiently numerous, to offer the necessary expressive power to represent any kind of inflection, but small enough, not to make the task of creating the rule too difficult.

P (Push) : move a character from the list to the stack

D (Dump) : moves the character of the stack to the list

E (Erase) : deletes a character from the list

/x/ : adds the character x at the end of the list

To simplify code writing, it is possible to indicate the number of repetitions of an operator. Here is an example of code that generates an inflected form from its lemma.

Step	Mot	Pile	Code restant
1	céder		`3PE/è/3D/ais/`
2	cé	der	`E/è/3D/ais/`
3	c	der	`/è/3D/ais/`
4	cè	der	`3D/ais/`
5	cèder		`/ais/`
6	cèderais		

Steps:

1→2 : stack of three characters

2→3 : deleting of a character

3→4 : adding the character *è*

4→5 : dumping three characters from the stack

5→6 : addition of three characters *ais*

This code can inflect french verbs like *céder* (as *révéler*, *espérer*, ...). However, this kind of code does not allow reversing the operation, *i.e.* find the lemma from the inflection : the E (erase) operator, unlike other operators, is not reversible. Therefore we add another operator to erase this function or remove the characters to delete.

`\x\` : erase given character from the list (this rule cannot be applied if the character is not present)

The code of the previous example becomes `3P\é\/è/3D/ais/`.

Since consonant duplication is a common phenomenon, we introduce a specific operator :

C (Clone) : duplicates the last character of the list.

The code `C/ing/` generates the present participle of words such as *run*, *sit* or *stop*

The management of prefixes requires the addition of operators:

`]` (fill stack) : transfers all the characters from the list to the stack

`[` (empty stack) : transfers all the characters from the stack to the list

Operator] can prepare an addition at the beginning of a word since all characters are put in the stack. We are now able to describe the inflexion of the form: *move* $\rightarrow$ *unmovable*. The transformation "remove the character 'e' at the end of a word, add 'un' at the beginning and 'able' end of a word" is coded `\e\]/un/[/able/`. The same code can analyse verb constructions that end with the character *e*.

Processing compound words requires the addition, or rather the transformation, of an operator. The difficulty here is to distinguish the different components of an expression with respect to one or more separators (traditionally in French space, hyphen or apostrophe).

P|**x**| (Push): moves the character of the list to the stack to meet the character x

Changing the stacking operator allows us to access directly an element of an expression or a compound word. Please note that access to different elements of an expression is achieved by stacking and unstacking successively. The code `2P|-|/s/[` allows to form the plural of expressions such as *brother-in-law* : only the third word from the end is pluralized (*brothers-in-law*). To preserve the analysis function of the model, it would be necessary to add, symmetrically, a conditional popping operator (*e.g.* `D|x|`). However, compound words analysis is far more complex, and such an operator could not bring the solution.

5.2 Management of inflexion

We have defined an XML DTD to manage the inflections expressed in code.

```
<flex id="n-y-p" type="final">
    <name>Np</name>
    <info>Noun plural with
          a terminal y</info>
    <code>\y\/ies/</code>
<\flex>
```

The above definition associates n-y-p identifier with the code `\y\/ies/`. A typical inflexion is characterised by:

- an identifier (attribute `id`) is used by the description language ;
- a status (optional attribute `type`) ;
- a name (optional element `<name>`) which corresponds to the tag associated to the inflected form ;
- information (optional element `<info>`) about the inflection ;
- a Proteus inflection code (element `<code>`).

However it is often necessary to combine several transformations : masculine/feminine and singular/plural for nouns and adjectives, persons and tenses for verbs. Take for example the conjugation of a French verb in the first group in the present tense. The prototypical inflection may be given as follows:

```
<flex id="v1ip" type="term">
    <name>Vp</name>
    <info>verbes
        indicatif présent</info>
    <flex id="p1ns">
        <name>1s</name>
        <code>/e/</code>
    </flex>
    <flex id="p2ns">
        <name>2s</name>
        <code>/es/</code>
    </flex>
    <flex id="p3ns">
        <name>3s</name>
        <code>/e/</code>
    </flex>
    <flex id="p1np">
        <name>1p</name>
        <code>/ons/</code>
    </flex>
    <flex id="p2np">
        <name>2p</name>
        <code>/ez/</code>
    </flex>
    <flex id="p3np">
        <name>3p</name>
        <code>/ent/</code>
    </flex>
</flex>
```

In this structure we regroup all the inflections of a given tense. Each inflection is : associated to its own identifier, prefixed with the main identifier, separated with a point, and associated to a name which is also a concatenation. Note that it is the identifier that must be unique and not the name. This mechanism allows for the expression of variants in a paradigm (see below). The previous definition states that, for the first group of the present tense, French verbs require suffixes at the end of the canonical form. Note that this is

a generic definition that can take into account exceptions, and can be apllied to any tense or mood.

identifier	name	code
v1ip.p1ns	Vip1s	/e/
v1ip.p2ns	Vip2s	/es/
v1ip.p3ns	Vip3s	/e/
v1ip.p1np	Vip1p	/ons/
v1ip.p2np	Vip2p	/ez/
v1ip.p3np	Vip3p	/ent/

It is also possible to group inflections with a new element (<op> with the attribute type).

```
<flex id="vig1-1" type="nonterm">
    <name></name>
    <info>first group
      indicative</info>
    <op type="add">
        <item value="v1ip"/>
        <item value="v1ii"/>
        <item value="v1ips"/>
        <item value="v1ifs"/>
    </op>
</flex>
```

To the previous definitions we need to modify the code in order to add a prefix operation: *remove the 'er' at the end of the lemma*. So we added the possibility of code concatenation to a previously defined group. In the example bellow the pos attribute determines if the code to be added is a prefix (p) or a suffix (s). The value attribute indicates the identifier of the structure upon which the operation is applied.

```
<flex id="v1" type="final">
    <name></name>
    <info>"er" verb</info>
    <op type="conc" value="vig1-1">
        <item pos="p">\re\</item>
    </op>
</flex>
```

In some cases, modification has to be performed on a particular inflection. This is done via the application of a *mask* which operates on a group of inflections and changes, possibly selectively, codes of inflexion. A mask is a set of rules applied on code. A regular expression on the identifier (ervalue attribute) performs the selection. We use Proteus code to modify Proteus code. This *mise en abyme* seems inconsistent, since Proteus has been designed to apply on a language element. But it seemed inappropriate to introduce a new syntax.

The definition below allows to add the letter *e* to a form in order to maintain its pronunciation [ʒ].

```
<mask id="m-ge">
 <info>add e after a g</info>
  <item ervalue="v1ip\.p1np">
        ]5D/e/[</item>
  <item ervalue="v1ii\.p([123]ns|3np)">
        ]5D/e/[</item>
  <item ervalue="v1if\.p([12]n[ps]|3np)">
        ]5D/e/[</item>
</mask>
```

The previous definition transforms code as \er\/ons/ in \er\/eons/, \re\/ais/ in \re\/eais/, ... The mask is used in combination with the attribute mask in a inflection definition, as in the conc attribute.

```
<flex id="v1" type="final">
    <name></name>
    <info>verbes en "er"</info>
    <op type="mask" value="vig1-1">
        <item value="m-ge"/>
    </op>
</flex>
```

You can build a complex inflection by using a base and applying masks successively. The inflection of the French verb *neiger* (to snow) can be expressed using two masks. First a mask to take into account the pronunciation of the [ʒ] and a second one, the weather verb mask, which is only used in the third singular person.

5.3 Applications

The examples bellow show the different capabilities of the model.

5.3.1 Neologisms in French

The formation of French inflections should not create significant problems. For the most common languages, simple forms are less than 1 million. Therefore, most systems use this set of inflected forms and supply modules to guess unknown words, only when they arise. This experiment is used to validate the model and to analyse unknown forms.

The example below shows how we analyse an unknown form.

```
anticonservationnistes =(/s/)=>
anticonservationniste =(]/anti/[)=>
conservationniste =(/niste/)=>
conservation =(\e\/ation/)=>
conserve
```

The algorithm tries to apply a code and reiterates the process on the result until we obtain an

attested form. The set of rules provides a potential analysis of the unknown word. Note that the rules used allow to determine the part of speech. In the example, the analysed word can be a plural noun or an adjective.

5.3.2 Arabic verbs

Arabic is a Semitic language ; it uses a semitic root to derive all the words used. For example from the root كتب (which refers to the writing) it is possible to produce verb (write), noun (desk) or adjective (written).

With these lemmas it is possible to agglutinate prefixes and suffixes. The rules are very regular in morphology but also very productive. We build all inflexion from the semitic root. So we have the schema: root → radical → inflected form. We then define inflexion for prefix/suffix (identifier `pass1term`), a mask for the radical (identifier `pass1radical`) and a definition which combine both.

```
<mask id="pass1radical">
 <info>add radical past</info>
 <item ervalue=".">
   ]/2P\/´\/D\/´\/D/[+
 </item>
</mask>
```

```
<flex id="pass1" type="nonterm">
  <name>Vis</name>
  <info>passe</info>
  <op type="mask" value="pass1term">
    <item value="pass1radical"/>
  </op>
</flex>
```

The problem encounter with arab text is the possible non use of all vowels. In fact they are rarely used, generally in pedagogical or religious text. This mean that the context is fundamental to interpret a text, a vowel is added only to remove an ambiguity. However we decide to describe language fully vowelled and to manage this specificity in an earlier stage.

The objective is to provide a resource used during an lexical analysis. This can be done in two ways[1]:

[1]We used here a transliteration version of arab writing to be more clear.

5.3.3 Old French

We are developping (author reference) an Old French resource, as exhaustive as possible. One difficulty is to consider the various alternatives, dialectal or chronological. This *proto-morphological* problem complicates the development of the dictionary nomenclature. We solved this problem by introducing an arbitrary "language Phantom" and by adding one level to the composition of the nomenclature, in the form of a label named hyperlemma. All derivations are from this entity using Proteus rules. All variants are generated from this item by application of successive masks.

The example below shows the successive masks applied on the inflection rules to account for the variations of the imperfect tense. Each mask is named *modifxxx* and corresponds to the modification of the Proteus rule for each century *xxx*.

```
<flex id="vg1i-5" type="final">
      <name>Vii</name>
      <info>first group
            imparfait</info>
    <op type="mask" value="v1ii">
        <item value="modifXI"/>
        <item value="modifXII"/>
        <item value="modifXIII"/>
        <item value="modifXIIIa"/>
        <item value="vrber"/>
    </op>
</flex>
```

6 Implementation

This framework is not only a theoretical tool ; it is designed to be implemented in a tagging software as an autonomous module. Based on abstract descriptions (Proteus code and XML language), it allows the resource creator to focus on linguistic aspects. It is simple enough to be easily expressed in any computational language.

The platform described here is developed in Python, which allows a very compact coding and can be used for both generation and analysis.

7 Conclusion

Our work is part of a set of tools and resources dedicated to the analysis of natural language. We have presented a model for the representation of inflections coupled with a language to structure the transformation rules. Compound words are

handled in the same way as simple words. The proposed model also allows simple word identification in both analysis and resource generation functionality. We have presented three examples of the use of the model, each introducing a specificity: French, Old French and Arabic. In the near future we expect to begin work on the Korean, Polish and Greek.

The best way to improve the framework is to create real, *i.e.* exhaustive linguistic resources. The development of the framework can be considered from several ways.

The Proteus code and the XML language description need stability. In our opinion, addition of operations to take into account some language specificities would complicate the model without adding any significant improvement. These modifications will take place during the third stage, the word level, where post-treatments are applied. For instance, the tonic accent in Greek can move along the last three syllables of a word and affects the use of the diaeresis mark in diphthongs.

As far as the analysis functionnality is concerned, we are considering to develop specific heuristics for each language in order to guide the choice of rules.

References

Buvet, Pierre-André, Emmanuel Cartier, Fabrice Issac, and Salah Mejri. 2007. Dictionnaires électroniques et étiquetage syntactico-sémantique. In Hathout, Nabil and Philippe Muller, editors, *Actes des 14e journées sur le Traitement Automatique des Langues Naturelles*, pages 239–248, Toulouse. IRIT Press.

Evans, Roger and Gerald Gazdar. 1996. Datr: A language for lexical knowledge representation. *Computational Linguistics*, 22(2):167–216.

Namer, F. 2000. Flemm : Un analyseur flexionnel du français à base de règles. *Revue Traitement Automatique des Langues*, 41(2).

New, Boris. 2006. Lexique 3 : Une nouvelle base de données lexicales. In Mertens, P., C. Fairon, A. Dister, and P. Watrin, editors, *Verbum ex machina. Actes de la 13e conférence sur le Traitement automatique des langues naturelles*, Cahiers du Cental 2,2, Louvain-la-Neuve. Presses universitaires de Louvain.

Paumier, Sébastien, 2002. *Manuel d'utilisation du logiciel Unitex*. Université de Marne-la-Vallée.

Romary, Laurent, Susanne Salmon-Alt, and Gil Francopoulo. 2004. Standards going concrete : from lmf to morphalou. In Zock, Michael, editor, *COLING 2004 Enhancing and using electronic dictionaries*, pages 22–28, Geneva, Switzerland, August 29th. COLING.

Sagot, Benoît, Lionel Clément, Éric Villemonte de la Clergerie, and Pierre Boullier. 2006. The lefff 2 syntactic lexicon for french: architecture, acquisition, use. In *LREC'06*, Gênes.

Silberztein, Max. 2005. NooJ's dictionaries. In Vetulani, Zygmunt, editor, *LTC'05*, pages 291–295, Poznań, Poland, April.