Natural Language Morphology Representation

by

Mikaël P.A. Roussillon

A Thesis Submitted in Partial Fulfilment of
the Requirements for the Degree of

Master of Computer Science

in the Graduate Academic Unit of Computer Science

This thesis is accepted by the
Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

December, 2004

Pour mes parents, qui ont toujours cru en moi.

Abstract

This thesis defines Lightweight Morphology, an alternative to stemming, which creates inflection and derivations from an input word using (1) a set of pattern matching rules producing morphological variants, (2) rules in Java and (3) an exception table to handle exceptions of a language. A language (LiteMorph) was developed to represent natural language morphology specifications for Lightweight Morphology. A French specification was created using LiteMorph, requiring 526 rules, 41 rule sets and 16,842 exception table words. A comparison between an exact query, stemming and Lightweight Morphology was performed. Using a differential recall measure on a collection of 533 documents (Hansard proceedings of the 36th parliament of Canada), we showed that Lightweight Morphology has, on average, 3.9 times more queries retrieving fewer irrelevant documents than stemming. The French version has, on average, 2.5 times more queries retrieving more relevant documents compared to stemming. Two new measures (reflexivity and transitivity) of morphological consistency were defined and tested. The English and French LangLMs have reflexivity scores around 0.9 and transitivity scores under 0.09.

Acknowledgements

I thank my supervisor, Dr. Bradford Nickerson, for giving me the opportunity to work on such an interesting project, for the many discussions that led to this thesis, for his guidance and inspiration.

I would like to thank Dr. Stephen Green, from Sun Microsystems, for all the handy explanations he provided me and for helping me in solving some of my problems and Dr. William Woods, also from Sun Microsystems, for creating such an incredible information retrieval system, and for the many comments and information he provided me.

I am thankful to Dr. Anna Maclachlan, for introducing me to linguistics, and for the many resources she was able to provide me.

Special thanks to the Faculty of Computer Science and Linda Sales for their help on many occasions.

Finally, I want to show my sincere gratitude to my family and my friends for their support and their affection.

Table of Contents

List of Figures

List of Tables

Glossary

$\cap^A_B$ (or $A \cap B$) One of the differential recall measure defined as the number of relevant documents returned by both A and B, where A and B are two different IR systems. The same measure can be computed with the number of irrelevant documents, p. 77.

Δ^A_B (or $A - B$) One of the differential recall measure defined as the number of relevant documents returned by A but not B, where A and B are two different IR systems. The same measure can be computed with the number of irrelevant documents, p. 77.

abstract syntax tree Tree where a terminal or non-terminal of the grammar is represented as a node, p. 54.

ad hoc retrieval Retrieval of text documents given a text query from a user. The retrieved documents can be ranked, p. 1.

affix Morpheme that provides additional meaning. See also circumfix, infix, prefix, suffix, p. 8.

affixation Act of adding an affix to a stem. Circumfixation, infixation, prefixation and suffixation can also be used to specify the kind of affix added, p. 8.

ambiguity Refers to phrases and words that can have different meanings, p. 3.

circumfix Affix that precedes and follow the stem, p. 8.

collection Set of documents used to satisfy users requests, p. 1.

conjuguaison (conjugation) Set of inflexions of the verbs, p. 57.

default rule set Specific type of rule set containing the first pattern-matching rules to be tried. See also rule set, p. 23.

derivation Word formation from a stem with a grammatical morpheme that changes the meaning or the use, p. 8.

differential recall Measure comparing two systems on the intersection and difference of relevant documents each system retrieved, p. 77.

document Unit of text indexed in a information retrieval system and available for retrieval, p. 1.

document corpus See collection, p. 1.

ending rule set Specific type of rule set containing all rules that process input words with the same ending. See also rule set, p. 23.

exception table Lightweight Morphology's hard coded set of morphological variants, p. 37.

F-measure Measure combining precision and recall. See precision and recall, p. 76.

finite state automata Automata that has a finite set of states, edges going from a state to another and each edge is labeled with a symbol. One state is the start state, certain of the states are final states. Finte state automata are usually used to represent regular expressions, p. 40.

finite state transducer Two level finite state automata. Each edge has two symbols, one being the symbol to match to go from a state to another, the other being used to create the output, p. 40.

grammaire (grammar) Set of rules to build correct statements, p. 57.

homonymy Relation between words that have the same form but unrelated meanings, p. 3.

infix Affix that is inserted in the stem, p. 8.

inflection Word formation from a stem with a grammatical morpheme that reflects grammatical features, p. 8.

information retrieval (IR) Technology for finding information from indexed materials in response to a user query, p. 1.

information retrieval system (IR system) Program or set of programs whose goal is to perform information retrieval. Usually consisting of an indexer and a query processor, p. 13.

Java Compiler Compiler (JavaCC) Parser generator and lexical analyzer generator producing Java code, p. 53.

java rule set Specific type of rule set containing all rules that process input words with the same ending. See also rule set, p. 34.

LangLM processor Program that can create morphological variants given the rules provided in a LangLM, p. 21.

Language Specification for Lightweight Morphology (LangLM) Set of rules and exception list used to perform Lighweight Morphology, p. 21.

lemma Single abstract word representing a set of lexical forms having the same stem, major part of speech and word sense, p. 10.

lexeme Individual entry in a lexicon, most of the time a lemma, p. 10.

lexical analyzer Program breaking an input into tokens, given a specification of tokens, p. 53.

lexicon A finite set of lexemes, p. 10.

Lightweight Morphology (LM) Morphological analysis technique which tries to produce morphological variants from a word without lexicon, p. 14.

LiteMorph Language to specify a complete LangLM, p. 43.

lower language Output word of a finite state transducer, p. 40.

morpheme Smallest unit of meaning in language, p. 8.

morphological analysis Analysis of the morphology of words to derive rules that govern word creation, p. 10.

morphology Study of the formation of words by combining morphemes, p. 9.

morphotactics Type of morphology rules explaining which classes of morphemes can follow other classes of morphemes, p. 10.

ontology Hierachical organization of knowledege of distinct objects into subcategories of concepts, p. 11.

orthographe (spelling) Set of rules that define the way of writting words in a language, p. 57.

orthographic rules Type of morphology rules that model the changes that occur to a word when two morphemes are combined, p. 10.

$P_I(r)$ **(interpolated recall-precision at level r)** Precision defined $\max_{R(x)\geq r}(P(x))$ where x is the number of relevant documents returned by the IR system (TREC measure), p. 84.

$\overline{P}$ **(average precision)** Precision calculated after each relevant document is retrieved and averaged over the number of relevant documents retrieved (TREC measure), p. 85.

$P(d)$ **(precision after d documents have been retrieved)** Precision with a cutoff at d documents (TREC measure). See also R-precision, p. 85.

paradigm Linguisitic: set of different lexical units or different forms of a word that can commute in the same linguistic context. Grammar: set of different forms of a verb, p. 58.

parser Program checking that a text has a syntax following a given a grammar, p. 53.

part of speech Grammatical categorization of a word, p. 8.

pattern-matching rule Lightweigh Morphology rule defining a pattern a word has to match in order to apply the rule and a list of morphological variations that are to be applied on the previously matched word, p. 25.

polysemy Word that has a diversity of meanings, p. 3.

precision (P) Measure judging the accuracy of an information retrieval system, p. 76.

prefix Affix that precedes the stem, p. 8.

query User's information need expressed as a set of terms, p. 2.

R-precision ($P_R(d)$**)** Precision calculated with the first r documents retrieved by a query, where r is the total number of relevant documents for the query, p. 77.

recall (R) Measure judging of the ability of an information retrieval system to retrieve relevant documents from a collection, p. 75.

regular expression Expression standing for a set of strings using special characters to represent some patterns, p. 25.

relevance Subjective judgement on the relation between query and a retrieved document, p. 75.

root Word without any affixation, p. 8.

rule set Feature of Lightweight Morphology to group pattern-matching rules. See also default rule set, ending rule set and javaruleset, p. 22.

spelling rules See orthographic rules, p. 10.

stem Morpheme that carries the main meaning, p. 8.

stemmer Set of rules that will perform stemming, p. 11.

stemming Morphological analysis technique which tries to produce stems from words, p. 11.

subsumption Relation between meaning of words. A more general term subsumes a more specific term, p. 4.

suffix Affix that follows the stem, p. 8.

surface form Spelling of a word implicitly containing morphological information. For example, an *-s* ending is usually the surface form for the plural in English or French, p. 41.

synonymy Relation between words that have the same meaning, p. 4.

taxonomy Science of ontology-like organization, usually creating tree-like structures as arrangements, p. 11.

term Lexical item that can possibly occur in a collection, p. 2.

upper language Input word of a finite state transducer, p. 40.

vagueness Phrase or word that does not carries a precise meaning, p. 3.

visitor Design pattern to separate the syntax from the interpretation when traversing a data structure, p. 54.

word Ambiguous term defining an association of characters, p. 10.

Xerox Finite State Morphology (XFSM) Set of tools developped by Xerox using finte state automata to preform lexical/morphological analysis, p. 40.

CHAPTER 1

Introduction

Information retrieval is a wide ranging area that deals with storage and retrieval of any kind of media. In this thesis, our focus is on the retrieval of text documents (the retrieved documents can be ordered with a ranking algorithm) given a text query from a user. This type of task is known as **ad hoc retrieval**. We explore the use of morphological analysis on words to improve information retrieval.

1.1 Background on Information Retrieval

Before going into the details of information retrieval, we will introduce some terminology useful to information retrieval (terminology quoted from [23]).

Document is 'the unit of text indexed in the system and available for retrieval'. A document can take various shapes, like a web page, a newspaper article, a paragraph or a sentence.

Collection or Document corpus is 'a set of documents being used to satisfy users' requests'.

Term is ‘a lexical item that can possibly occur in a collection’.

Query is ‘a user’s information need expressed as a set of terms’.

1.1.1 Different approaches for different sizes of collections

The way information retrieval is approached is different depending on the document corpus one wants to retrieve information from. When this document corpus is the Internet (assuming that a great number of pages have been indexed), and provided a good search algorithm is employed, users are almost sure that the term(s) they provided as the query will be present in the document they are seeking, or at least that the returned documents are somewhat relevant to them.

When the size of the corpus is smaller, the users cannot expect to always retrieve the information they seek with the terms in a query. For example, if users issue a query with *car* and the corpus has only documents indexed with *cars*, they will never retrieve the information they are seeking. This first problem comes from systems that don’t use morphological analysis on query words. Furthermore, if the document contains neither *car* nor *cars* but *vehicle*, the users will not find the document until they find the right word (a synonym or a related word) for their idea. This second problem is an ontology problem, where there is a need to attach meaning to a word, and retrieve words with similar meaning.

1.1.2 Natural language implicit problems for information retrieval

The previous considerations are an introduction to the problems involved with natural languages, because they are **ambiguous** and **vague**. **Ambiguity** refers to phrases (or even words) that can have different meanings. The phrase ‘I made her duck’ (an example given in [23]) can have (at least) five different meanings:

1. I cooked a duck (animal) for her;
2. I cooked a duck (animal) belonging to her;
3. I created the duck (a representation of an animal) she owns;
4. I caused her to quickly lower her head or body;
5. I waved my magic wand and turned her into undifferentiated duck (animal).

These different meanings come from different ambiguities existing in natural languages. The ambiguities that are of interest for us are **homonymy**, a relation between words that have the same form with unrelated meanings and **polysemy** a word that has a diversity of meanings, which can be summarized into **lexical ambiguities**. For example, *duck* can be a verb or a noun (homonymy) and *make* can mean either *create* or *cook* (polysemy). The other possible ambiguity is **semantic ambiguity**, describing the way words are semantically analyzed to produce a meaningful phrase. This will be of minor interest in this thesis since semantic ambiguity uses word sense disambiguation in a context, and not morphological analysis of the word.

Vagueness is related to ambiguity. The phrase ‘I want to eat Italian food’ [23] is vague since we don’t really know what the speaker really wants to eat when referring to Italian food — it can be pizza or pasta or something else. This can be seen as an example of **synonymy**, where different words are related to the same meaning.

More general than synonymy is the notion of **subsumption**. A more general term A subsumes a more specific term B. For example, *vehicle* subsumes *car* and *truck* since they are both a kind of vehicle. Another example is *subsumption* subsumes *synonymy*.

This thesis considers words as our field of interest, since our focus is morphological analysis, but not the association of words in phrases. The context in which those words are used is of lesser interest (even though it can be important for information retrieval) for the simple reason that queries on ad hoc retrieval are often constructed of one to three words and they do not necessarily form a semantic phrase.

1.2 Motivation

Computational linguistics is a very interesting topic since it unites two worlds that are quite different in appearance : computer science and natural languages. The aim of this thesis is to provide a bridge between the two fields. Who else than a linguist, expert in natural language morphology and in ontology, can define morphological analysis rules (Lightweight Morphology and Heavyweight Morphology) for a natural language? Computer scientists could do that but not without the constant counselling of linguists.

Sun Microsystems information retrieval system is written Java and Lisp. The morphology definitions for any natural language is done in one of those two languages (Lightweight Morphology in Java, Heavyweight Morphology in Lisp). Although these are powerful computer languages, natural language morphology definitions do not need all the features (and complexities) from Java and Lisp. Also, one should not be constrained to a particular computer language when defining morphologies (new and obsolete computer languages are common, but natural languages are here to stay for quite a long time). Providing a representation for natural language morphology is a necessary step since

1. It produces a language that abstracts the underlying computer science language;
2. It produces a language that is not bound to the underlying computer science language.

The second challenge of this thesis is to test the controversial hypothesis 'linguistic knowledge can improve information retrieval'. The elements that will be explored are:

1. Using linguistic knowledge in a deeper way than it can be used in stemming;
2. Linguistic knowledge, especially morphological analysis for highly inflectional languages, can improve information retrieval;
3. Basing results of the usefulness of linguistic knowledge over only one language (English) is not a satisfactory way to proceed.

To achieve these goals, we define clearly what Lightweight Morphology is, using a context free grammar. We introduce LiteMorph, a language that can specify a Lightweight Morphology for a natural language, and we implement a specification for French in LiteMorph. We finally test the usefulness of Lightweight Morphology in information retrieval.

This thesis is organized as follows: Chapter 2 presents some background on morphological analysis for information retrieval, Chapter 3 presents Lightweight Morphology and its representation. Chapter 4 presents LiteMorph, a language to specify a Lightweight Morphology for a natural language. Chapter 5 presents an application of a Language specification for Lightweight Morphology (LangLM) for French using LiteMorph. Chapter 6 introduces some measures on the quality of clusters obtained from morphological analysis and tests the benefits of using Lightweight Morphology for information retrieval. Chapter 7 presents future directions for Lightweight Morphology representation (with LiteMorph) and testing (with introduced measures), and Heavyweight Morphology representation.

CHAPTER 2

Improving Information Retrieval with Morphological Analysis

The main hypothesis of this thesis is that morphological analysis (and more generally linguistic knowledge) can improve information retrieval. This hypothesis is quite controversial since different studies confirm or infirm the usefulness of morphological analysis for information retrieval. We will see in the following chapters that the conclusion of the usefulness of morphological analysis is often biased by the fact that tests are made with English, a language that presents few inflectional forms.

2.1 Morphological and lexical terminologies

The following terminology definitions are in part quoted from [23].

2.1.1 Morphological terminology

A **morpheme** is the smallest unit of meaning in language. For example, *car* consists morphologically of a single morpheme (the morpheme *car*) whereas *cars* consists of two morphemes *car* and *-s*. We can therefore divide the morphemes in two categories: the **stems** which provides the main meaning, and **affixes** that provide additional meanings to a stem. Different kind of affixes exist: **prefixes** that precede the stem, **suffixes** that follow the stem, **circumfixes** that do both and **infixes** that are inserted inside the stem. The act of adding an affix to a stem is called **affixation** and the result can be another stem. A word without any affixation is called a **root**. Figure 2.1 gives two examples of decomposing words.

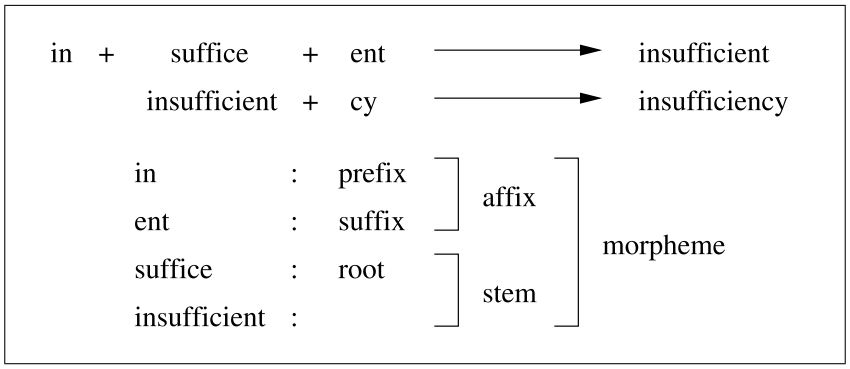


Figure 2.1: Examples of decomposition of words using morphological terminology.

Part of speech tagging is a grammatical categorization of words. Many parts of speech exist but the most common ones are nouns, verbs, adjectives, adverbs and articles. Word formation from morphemes is divided into two classes: **inflection** and **derivation**. Inflection is the combination of a stem with a grammatical morpheme that results in

1. changing the form of the word (as in paradigm or declension);

2. retaining the part of speech;

3. reflecting grammatical features such as number, gender or person.

Verbal systems are a good example of inflection. Derivation is the combination of a stem with a grammatical morpheme that results in

1. deriving a new word by changing meaning or use;

2. often changing the part of speech.

Nominalization (forming a noun from a verb or adjective) is a good example of derivation. Figure 2.2 presents an example of inflection and derivation.

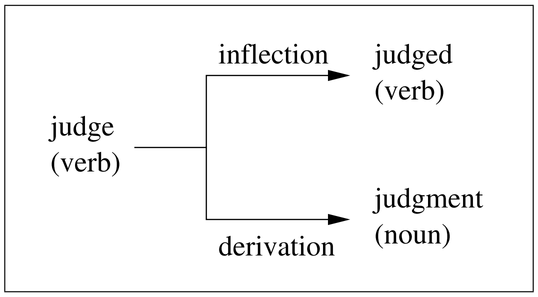


Figure 2.2: Example of inflection and derivation of the verb *to judge*.

Compounding is the action of combining two roots to create a compound word. We define the **head** of a compound word as the root that carries the principal meaning of the word. Usually, the head is the last root. Figure 2.3 present some examples of compound words.

Morphology[1] is the study of the formation of words by combining morphemes. More generally, morphology is the explanation of the formation of a word in terms

[1]From Greek *morphê*, form, shape.

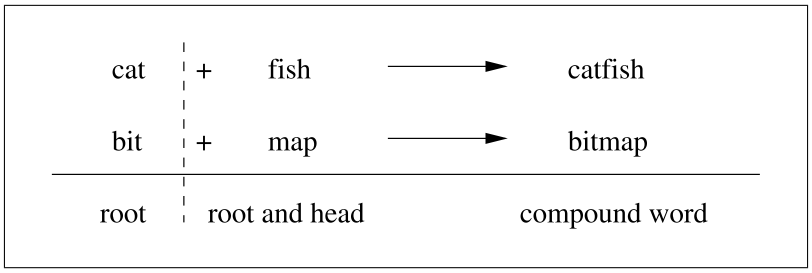


Figure 2.3: Examples of compounding.

of morphemes. **Morphological analysis** consists of analyzing the morphology of a word, to identify the rules that govern word creation. The rules are divided into two classes: **morphotactics**, explaining which class of morphemes can follow other classes of morphemes (e.g. the plural morpheme follows the noun in English) and **orthographic rules** (or **spelling rules**), to model the changes that occur to a word when two morphemes are combined (e.g. *city* + *s* → *cities* but not **citys*[2]). Inflection and derivation are part of both morphotactics and orthographic rules.

2.1.2 Lexical terminology

Word is too ambiguous to be of any use for lexical terminology. Instead, we will use the notion of **lexeme**, an individual entry in a lexicon. Most of the time, the lexeme will be a **lemma**, a single abstract word representing a set of lexical forms having the same stem, the same major part of speech and the same word sense. The key entries in any language dictionary are lemmas (e.g. if you are looking for *sings*, you will look for the lemma *sing* representing the infinitive). A **lexicon** is a finite set of lexemes. Others notions that are useful to provide relations among lexemes are the previously defined notions of homonymy, polysemy and synonymy.

[2]Every non-existing word quoted in this thesis will be prefixed by *.

An **ontology** refers to the hierarchical organization of knowledge of distinct objects into subcategories of concepts. A **taxonomy** is the science of such organization, usually creating tree-like structures as arrangements.

2.2 Stemming

Morphological analysis has a well-known technique called **stemming** that is defined as producing stems from inflected (and possibly derived) forms. Two famous **stemmers** are the Lovins stemmer [28] and the Porter stemmer [32]. A stemmer consists of a series of rules that will strip affixes from a word until there are no more affixes to remove. Each stemmer differs in the number of rules it uses, how the rules are tried and which affixes are removed. An example of a stemmer is shown in Figure 2.4.

```
if word ends in $ies$ but not $eies$ or $aies$ then
    $ies \longrightarrow y$
end if
if word ends in $es$ but not $aes$, $ees$ or $oes$ then
    $es \longrightarrow e$
end if
if word ends in $s$ but not $us$ or $ss$ then
    $s \longrightarrow \epsilon$
end if
```

Figure 2.4: A simple stemmer, the S-stemmer, removing *s* plural in English (from [18]).

Quoting Krovetz [24], stemming can be seen as (see Figure 2.5)

1. a mechanism for automatic query expansion (adding terms that were not provided in the query);

2. a mechanism for clustering words (stems are shared among related words);

3. a mechanism to normalize a query by using a concept (the search in the collection will be made with the stems, not the words from the query).

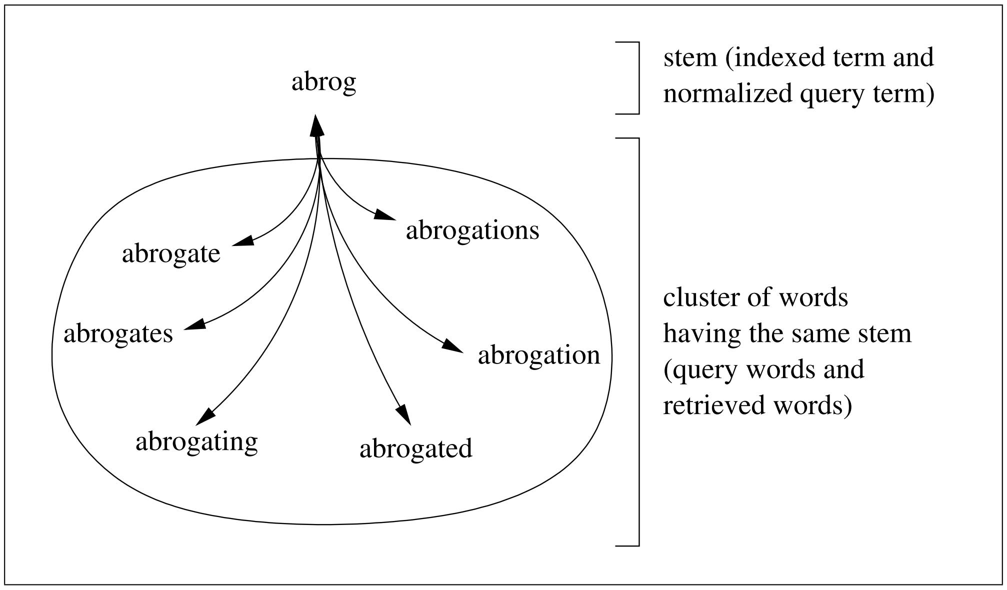

abrog
stem (indexed term and normalized query term)
abrogate
abrogations
abrogates
abrogation
abrogating
abrogated
cluster of words having the same stem (query words and retrieved words)

Figure 2.5: Relationships introduced by stemming.

The major interest of stemming is that, while reducing the number of terms indexed by indexing stems (there are fewer stems than terms occurring in a document), we hope that the stem will retrieve more relevant documents. While stemming seems interesting (especially for languages with numerous morphological variants [30]), it presents serious limitations which led to the conclusion that morphological analysis was not useful for information retrieval (based on the English language) [18]. The problems, as pointed out by Krovetz [24], are the following :

1. The result of removing affixes is a stem and not necessarily a root. *Iteration* will be turned into the stem *iter* where the root is *iterate*. Therefore, different

words with different meanings can result in the same stem or words with the same meaning can result in different stems;

2. The stemmer does not take into account the meaning of a word. For example, *gravitation* will be turned into *gravity*, ignoring the sense of *gravity force* and producing a stem with the meaning *serious*.

The problem is not that morphological analysis is a bad thing but that, as for today's knowledge, stemming has some drawbacks that makes it less useful for information retrieval. In [24, 25, 41], different approaches are presented (taking full advantage of the knowledge of a language or using a machine-readable dictionary in conjunction with a stemmer) showing that by going beyond a simple stemming step, morphological analysis is very useful for information retrieval.

2.3 Beyond simple suffix stripping

The approach used by Sun Microsystems **information retrieval system** [41] is to provide morphological analysis and an ontology at two different stages: the indexer and the query processor.

2.3.1 Architecture of Sun Microsystems information retrieval system

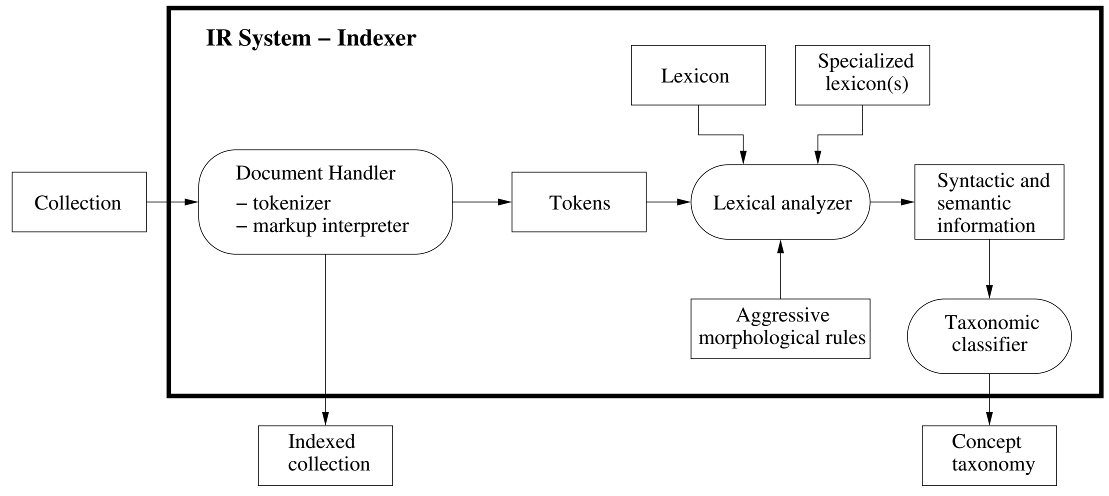
The Sun Labs Search Engine [15] provides morphological analysis both on indexing and querying. The query processor tries to obtain morphological variants of the query terms with two morphological analysis techniques: a stemmer and Lightweight Morphology. One can also choose not to perform morphological analysis on the query terms or specify which technique has to be used for each term of the query.

The indexer's goal is to create a classical index of the documents along with building an optional taxonomy of concepts. Aggressive morphological analysis assisted by a lexicon is used to build the taxonomy. The taxonomy can then be used by the query processor to obtain subsumed concepts of the terms provided in the query. Figure 2.6 presents an architectural overview of the engine.

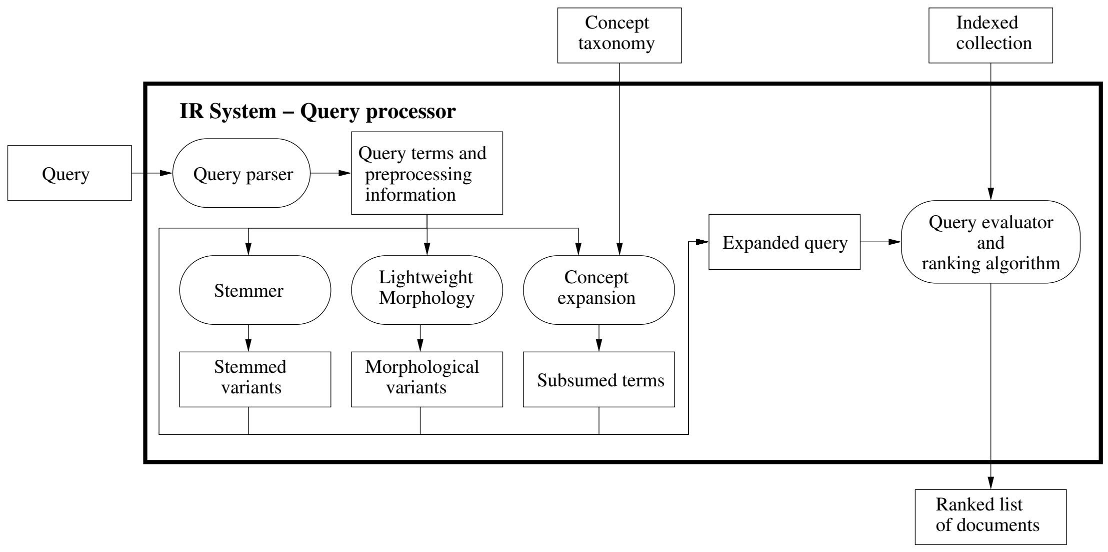
2.3.2 Lightweight Morphology

Lightweight Morphology is a morphological analyzer of terms provided in queries. It consists of producing morphological variants from a query term, whereas stemming tries to reduce terms to a unique stem. This can be seen, from a mathematical point of view, as creating the **equivalence class** of the word. If we have a noun as input, we want the word's inflections and derivations (and the inflections of the derivations) as part of the query. For example, for the French word *étudiant*, we would want *étudiants*, *étudiante* and *étudiantes*, along with *étudier* and all its inflected forms as an intrinsic part of the query.

Lightweight Morphology's goal is to build a cluster of words that approximate the equivalence class. An example of such a cluster can be found in Figure 2.7. The analogy to the mathematical concept of equivalence class is not total: during Lightweight Morphology processing we can create non-existent words or we can have a word appearing in different equivalence classes but with a different meanings (e.g. *found* $\longrightarrow$ *find* or *found* $\longrightarrow$ *founder*). The non-existent words are not a problem



(a) Indexer.



(b) Query processor.

Figure 2.6: Sun Microsystems information retrieval system architecture.

because if they do not exist, they will (hopefully) not appear in an indexed text and never be retrieved.

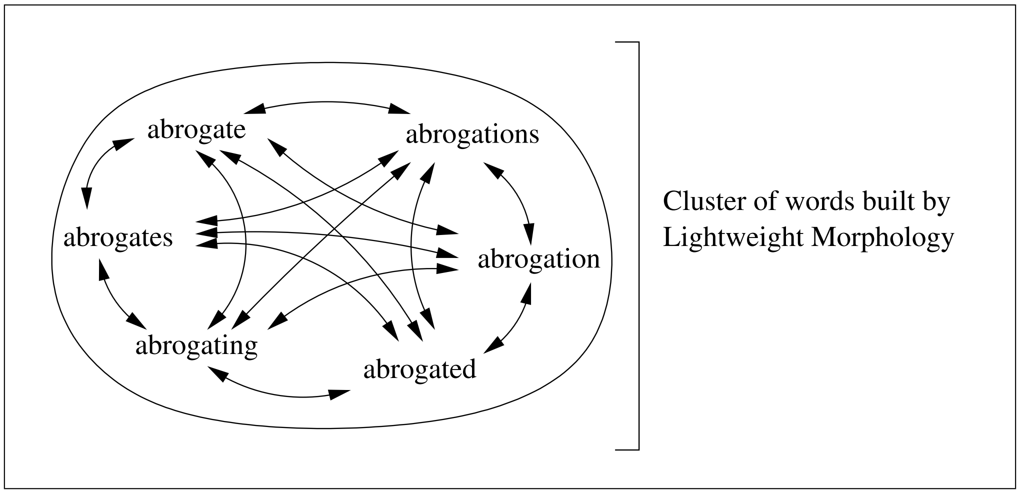


Figure 2.7: Relationship introduced by Lightweight Morphology.

Lightweight Morphology shares some of the properties of stemming: automatic query expansion and clustering. Lightweight Morphology, however, does not normalize a query. This is one of its strengths; normalizing means having a single representative for a class. In natural languages, this representative can fall short in representing its whole class.

2.3.3 Conceptual Indexing

As previously pointed out, information retrieval systems can be improved by providing morphological analysis at a higher level and solving part of the paraphrase problem (i.e. how to formulate a search for a concept as a query, until the terms appear in a relevant document indexed by the IR system). Lightweight morphology provides some morphological analysis, or better said, uses morphological information to produce possible morphological variants.

Conceptual indexing [38] goes beyond producing morphological variants. It tries to organize knowledge with a taxonomy using the notion of subsumption.

To build this conceptual index (a taxonomy of subsumed terms), a core lexicon containing about 40,000 subsumption axioms (mostly for roots and irregularly inflected forms) and an aggressive morphological analysis [38, 39] (or Heavyweight Morphology) of about 1200 rules dealing with prefixes, suffixes, inflections and compounding are used. The role of the morphological analysis is to identify syntactic features along with semantic relationships. The core lexicon will handle the cases where the aggressive morphological analysis fails.

The taxonomy is used at query time by automatically adding more specific terms to the query. Adding more general terms is also possible but not suitable. For example, a query containing *vehicle* will probably look for documents containing *car* and *truck*.

2.3.4 Heavyweight Morphology

Aggressive morphological analysis (or Heavyweight Morphology to contrast with the Lightweight Morphology) is relying on rules. A typical Heavyweight Morphology rule is presented in Figure 2.8. The rule is divided between conditions and inferences. The first line of Lisp is a condition/action performing the following: if the word ends in *fish*, remove this ending and let `root` be the stripped word (e.g. *catfish* will be turned into *cat*). The next two lines are conditions, checking if `root` is a plausible root (e.g. it has at least a vowel in it) and if the root is an adjective or a noun. The second condition also creates a category `nmsp`, 'a category indicating a word that has a mass sense, a singular count sense, and can also be used as a plural (e.g. Goatfish are funny-looking).' [39], in which the word is assigned.

The following lines are the inferences: given that the previous conditions are true, the `root` of the word is a false root, the word is defined as a kind of *fish*, the prefix of the word is `root`, *fish* is the real root of the word and the word belongs to an *es* inflectional paradigm.

```
((f i s h) (kill 4)
   (test (plausible-root root))
      (cat nmsp (is-root-of-cat root '(adj n))
           eval (progn (mark-dict lex 'false-root root t t)
                (mark-doct lex 'kindof 'fish t t)
                (mark-dict lex 'has-prefix root t t)
                (mark-dict lex 'root 'fish t t)
                '-es)))
```

Figure 2.8: -*fish* Heavyweight Morphology rule written in Lisp (from [39]).

The rules are ordered from the most specific to the most general following an heuristic. As in Lightweight Morphology, the process will stop at the first rule encountered that is successful.

CHAPTER 3

Lightweight Morphology

Lightweight Morphology (LM) should be seen more as a methodology than a program (although there is a Java implementation). As a methodology, it should provide a formalism that is independent of the underlying implementation. Lightweight Morphology is a concept that was introduced by William Woods [40], but lacked a formal representation. Specifications for English, German and Spanish were already existing. The Java implementation is in part the work of William Woods and is used in the Sun Labs Search Engine [15]. This chapter introduces what this methodology consists of and presents a representation to specify a Lightweight Morphology for any natural language using affixes.

3.1 Generating Morphological Variants

Lightweight Morphology is a methodology that is similar to stemming because it uses natural language morphological analysis. If both approaches use morphological information, LM and stemming results are opposite.

In stemming, the goal is to strip affixes from a word to reduce related words (either by inflection or by derivation) into a common stem. The LM approach is rather different since it focuses less on finding the right affix to strip and more on the construction of a word with reasonable affixes the root could have. Also, it does not try to reduce morphologically related words into a common stem but instead it explicitly creates those morphologically related words. Of course, affixes are important in LM, since word creation is essentially ruled by affixation, but affix stripping can sometimes be too weak to generate the root of the word.

The term *lightweight* refers to the fact that this methodology does not have access to a lexicon or dictionary to perform word look up, so no *a priori* part-of-speech decision is made. Instead, the methodology relies on the fact that the morphology of the word (ending letters, letter at a certain position) is sufficient to guess the part-of-speech of the word, and that non-valid morphological variants (e.g. *control* $\longrightarrow$ **controlest*) will be naturally filtered by the collection (see Figure 3.1).

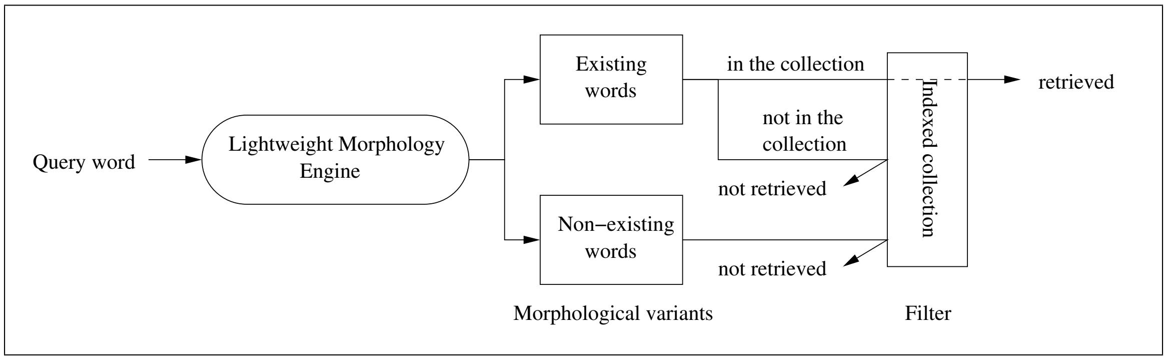


Figure 3.1: Lightweight Morphology variant production and filtering.

3.1.1 Components of Lightweight Morphology

We need to distinguish two components of Lightweight Morphology: the overall architecture needed to implement a Lightweight Morphology (of limited interest for the following discussion) and the architecture needed to specify a Lightweight Morphology for a specific natural language.

3.1.1.1 Architecture of Lightweight Morphology

A Lightweight Morphology engine is divided between the natural language specifications, called **Language Specification for Lightweight Morphology** (LangLM), whose responsibility is to represent how morphological variants have to be created for a defined natural language and the **LangLM processor** that is responsible for creating the variants using this representation (see figure 3.2).

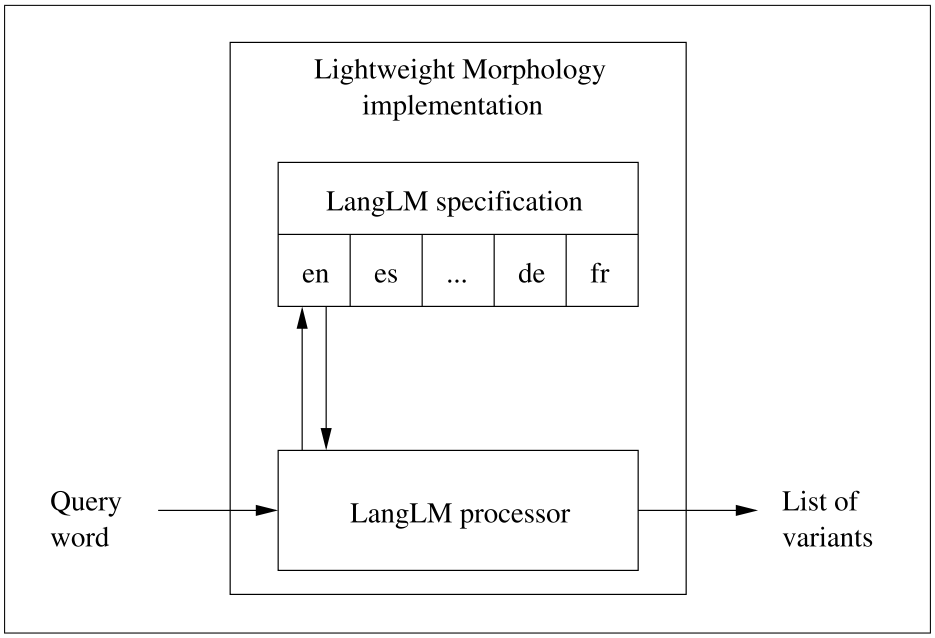


Figure 3.2: Lightweight Morphology implementation architecture.

3.1.1.2 Defining a Language Specification for Lightweight Morphology

LangLM is a set of three interacting parts. The first one (mandatory) is a set of rules that define patterns the word must match and the morphological variations that are applied to the matched word. The second one (optional) is a way for the user to define their own rules in Java, thus completing the expressiveness of LM. The third one (optional, but strongly suggested) is an exception table, which is designed to capture all the exceptions of a language that would be too complicated or too tedious to implement as rules (e.g English irregular verb *think* inflecting to *thought*).

We will present in the following section how the rules are structured, describe each component in detail and how the three components interact with each other.

Two formalisms exist to write a LangLM — writing Java code directly or using the LiteMorph language (see Chapter 4). The two formalisms are very close, since the LiteMorph formalism is based on the Java formalism. We decided to present LM with the LiteMorph formalism because it avoids talking about some Java considerations. Section 4.3.1 will describe the Java formalism. In the following sections, if a paragraph begins with [**Java implementation**] we are talking about the Java formalism.

3.1.2 Aggregation of rules with a rule set

Before going into the details of how to write rules, we define here the notion of a **rule set**. As we will see in Section 3.1.6, ordering the rules is important. Moreover, some rules should only be tried in certain conditions, for example when creating a noun derived from a verb. LM provides a way to perform such operations. Each rule is encapsulated and ordered in a rule set. The **default rule set** is the first one to be tried, and the others are used only if an explicit call is made to them. We can somewhat change this behaviour by defining an **ending rule set**, a rule set where all rules that process input words with the same ending are grouped together. In this case, if a word has this ending, the default rule set will not be tried first, but the rule set defined with this ending will. If the rule set is neither a default rule set nor an ending rule set, then it is a common rule set, which can be called but has no special behaviour.

A Java rule set also exists. It is a specific kind of rule set containing Java code instead of pattern-matching rules (see Section 3.1.4).

As further explained in Section 3.1.6, the order of the rules inside a rule set is important, since Lightweight Morphology stops at the first successfully matching rule it encounters inside a rule set. The order of the rule sets does not matter.

Here are summarized the four existing rule sets, and the grammar for the rule set is given in Figure 3.3:

Default rule set The rule set to try first when given a new input word to process. Defined with:

`DEFAULT RULESET rule_set_name { rule_1 ... rule_n }`

Ending rule set The rule set to try first if the new input word to process ends by the character sequence defined by the rule set. Defined with:

`RULESET rule_set_name ENDING ending_letters { rule_1 ... rule_n }`

Rule set A common rule set that can be called. Defined with:

`RULESET rule_set_name { rule_1 ... rule_n }`

Java Rule set A rule set defined with Java. Defined with:

```
JAVARULESET Java_code ENDJAVARULESET
```

1	*RuleSetDefault*	::=	**RULESET DEFAULT** *Id* **{** { *Rule* }+ **}**
2	*RuleSetEnding*	::=	**RULESET** *Id* **ENDING** *Id* **{** { *Rule* }+ **}**
3	*RuleSetNormal*	::=	**RULESET** *Id* **{** { *Rule* }+ **}**
4	*JavaRuleSet*	::=	**JAVARULESET** { *JavaCode* }* **ENDJAVARULESET**
5	*Id*	::=	*Letter* { *Letter* | *Digit* | **-** }*

Figure 3.3: Rule sets grammar (LiteMorph formalism).

For example, a rule set containing rules to process words ending with *-s* (the plural ending for English) will be defined by `RULESET sEnding ENDING s { ...}`.

[**Java Implementation**] In the Java version of LM, each rule set is defined as a string array, except for the Java rule set. Each identifier (variable name) defining the string array is then mapped to another identifier which is the real name of the rule set via the method `defRule(String real_rule_set_name, String[] array_of_string_name)`. The distinction between the different kinds of rule sets is done with the name of the rule set by (1) prefixing the name by ':' if it is a rule set, (2) not prefixing if it is an ending rule set and defining the name as the ending pattern, and (3) defining the default rule set with the name ':unnamed'. Java rule set valid names are defined in the `computeMorphArgs()` method.

[**Java Implementation**] For the previous example, the Java implementation would first define the following array:

```
String[] sString = { "rule_1", ...  , "rule_n"};
```

and then call

```
defRule("s", sString)
```

3.1.3 Generating variants with pattern-matching rules

A **pattern-matching rule** is made of the following elements:

1. On the left hand side of the rule, a pattern the word has to match in order to apply the rule. The pattern is defined with regular expressions, and some mechanism is provided to interact with the right hand side.
2. On the right hand side of the rules, a list of morphological variations that are to be applied in order to obtain the variants.
3. The left hand side is separated from the right hand side with the '->' production symbol.

The grammar for the pattern-matching rule is given in Figure 3.4 and an example of a simple pattern-matching rule for English is given in Figure 3.5. In this example, the word *timeless* matches the left hand side pattern (the word has to end with *less* and the letter before *less* has to be a vowel), and the right hand side production creates *time*, *timer*, *timers*, **timest*, *timed*, *timing*, *timings*, *timely*, **timeness*, **timenesses* , **timement*, **timements*, and **timeful* (remove *less* from *time* and append the list of endings).

Pattern-matching rules offer more functionality than **regular expression** rules. Pattern-matching rules have knowledge of the previous states (the previously matched characters) to make a decision for the current state whereas regular expression rules can only make a decision based on the current state. Moreover, while performing the

```
1   Rule                  ::=  PatternElements -> ModificationPatterns ;
2   PatternElements       ::=  [ FirstPattern ] { LeftPatternEnd }* [ LeftPat-
                               ternInside ] { LeftPatternEnd }* [ LastPattern
                               ]
3   FirstPattern          ::=  [ BeginWord ] { LeftAnchoredPattern }* Begin-
                               Delimiter
                          |    BeginWord
4   BeginDelimiter        ::=  - ␣
5   BeginWord             ::=  #
6   LeftPatternEnd        ::=  [ . ] LeftAnchoredPattern
7   LeftPatternInside     ::=  < { LeftAnchoredPattern }* >
8   LeftAnchoredPattern   ::=  OpSet Letters | &
9   OpSet                 ::=  [ | ] [ ˜ ] [ * | + | ? ]
10  Letters               ::=  UnOrderedList | OrUnorderedList | LetterVari-
                               able
11  LastPattern           ::=  EndDelimiter { LeftAnchoredPattern }* [ End-
                               Word ] | EndWord
12  EndDelimiter          ::=  + ␣
13  EndWord               ::=  #
14  ModificationPatterns  ::=  RightPattern { , RightPattern }*
15  RightPattern          ::=  ReapplyPattern [ & ] [ EndSubstitution | Inside-
                               Substitution ]
                          |    > Mode UnOrderedList > UnOrderedList / _ [
                               UnOrderedList ]
                          |    & [ EndSubstitution | InsideSubstitution ]
                          |    * [ EndSubstitution | InsideSubstitution ]
                          |    { EndSubstitution | InsideSubstitution }
16  ReapplyPattern        ::=  [ TRY ] ( [ Id ] )
17  EndSubstitution       ::=  UnOrderedList _ [ - ] UnOrderedList | Un-
                               OrderedList _ | _ [ - ] UnOrderedList | [ - ] Un-
                               OrderedList | _
18  InsideSubstitution    ::=  < [ UnOrderedList ] > [ ContextPattern ]
19  ContextPattern        ::=  / [ EndSubstitution ]
20  Mode                  ::=  * | < | >
21  Id                    ::=  Letter { Letter | Digit | - }*
22  UnOrderedList         ::=  Letter { Letter | Digit }*
23  OrUnorderedList       ::=  Letter { | { Letter | Digit } }+
24  LetterVariable        ::=  $ Letter { Letter | Digit | - | _ }*
25  Letter                ::=  { A | ... | Z | a | ... | z }
26  Digit                 ::=  { 0 | ... | 9 }
```

Figure 3.4: Pattern matching rule grammar (LiteMorph formalism).

```
.aeiou + l e s s ->  _,s,er,ers,est,ed,ing,ings,ly,ness,nesses,
                     ment,ments,ful;
```

Figure 3.5: Example of LM pattern matching rule for English.

pattern-matching, some tagging is done in order to know how and where to generate the morphological variations.

[**Java Implementation**] The Java version of LM uses almost the same grammar for pattern-matching rules. The major differences are (1) rules are strings and placed inside quotes so no ';' is needed to end a rule, (2) a call to rules is made by specifying the kind of rule set to call (with a ':' or '!' prefix), and (3) a blank can have a more important meaning.

3.1.3.1 Left hand side to define a pattern

The left hand side of the pattern-matching rule has two purposes: (1) define a pattern on which the rule has to apply, and (2) define the parts on which the morphological modifications have to be applied.

The character sequences are defined as follows:

Unordered letters A group of letters separated with a space stands for a concatenation. For example, 'x y z' stands for 'x' followed by 'y' followed by 'z';

Or Unordered letters (*alternation*) A group of 2 or more letters with no blank stands for a choice between letters. For example, 'xyz' stands for 'x' or 'y' or 'z';

'&' (*double operator*) Intended to repeat the last letter encountered in the pattern. For example, 'mn &' stands for 'm' or 'n' followed by the repetition of the same previously matched letter;

Letter variable A letter or a group of letters prefixed with a '$' is a letter variable, previously defined. For example, one can define '$Vowel = aeiou' and then use '$Vowel' variable instead of 'aeiou';

'␣' (*blank*) is used to separate character sequences or some special operators.

The different operators that help to define a pattern by modifying the meaning of a character sequence are the following:

'#' (*boundary operator)* Defines the beginning and the end of a word, depending on where it is placed. If no character precedes '#' then it defines the beginning of a word; if no character follows '#' then it defines the end of a word; the other cases are errors. For example '# x' means a word beginning with 'x', 'x #' a word ending with 'x';

'.' (*anywhere operator*) Used as a prefix operator of a letter (or a group of letters) to indicate that the letter (or the group of letters) can be encountered anywhere in between the preceding and the following patterns. For example 'x .y z' means a word containing an 'x' with a 'z', and a 'y' between the 'x' and the 'z', thus 'xyz' and 'xdeyfz' both match the previous pattern. Without being preceded by this symbol, a letter (or a group of letters) has to be encountered where it is defined in the pattern, as with 'x y z' (so 'xdeyfz' is not recognized by this pattern);

‘|’ (*or operator*) Used as a prefix operator of a group of letters to indicate a choice between letters. For example ‘|xyz’ stands for ‘x’ or ‘y’ or ‘z’ and it is equivalent to ‘xyz’. This operator can also be used inside a group of letters between each letters, for an equivalent meaning. For example ‘x|y|z’ stands for ‘x’ or ‘y’ or ‘z’;

‘˜’ (*not operator*) Used as a prefix operator of a letter (or a group of letters) to indicate that the letter (or the group of letters) should not be encountered. For example, ‘# ˜n’ consists of all the words that do not begin with ‘n’.

‘?’ (*zero-one operator*) Used as a prefix operator of a letter (or a group of letters) to indicate that the letter (or the group of letters) should be encountered zero or one times;

‘+’ (*one-many operator*) Used as a prefix operator of a letter (or a group of letters) to indicate that the letter (or the group of letters) should be encountered one or more times;

‘*’ (*zero-many operator*) Used as a prefix operator of a letter (or a group of letters) to indicate that the letter (or the group of letters) should be encountered zero or more times;

The following operators describe how to define a pattern that will interact with the right hand side of the rule:

‘-␣’ (*begin substitution marker operator*) Indicates the start of a word that has to be stripped. Everything before this operator is like the rest of the pattern matching rule (except no anywhere operator and no inside substitution operator are allowed after this operator). Therefore, ‘y z - .wx’ designs a word that contains ‘w’ or ‘x’ and that starts with ‘yz’. The word that is passed to the right hand side is the word stripped of ‘yz’. Note that ‘-’ has the same meaning as ‘#’ at the beginning of a rule (if ‘#’ is not followed by the ‘-␣’ operator with a substitution pattern);

‘+␣’ (*end substitution marker operator*) Indicates the ending of a word that has to be stripped. Everything after this operator is like the rest of the pattern matching rule (except no anywhere operator and no inside substitution operator are allowed after this operator). Therefore, ‘.wx + y z’ designs a word that contains ‘w’ or ‘x’ and that ends with ‘yz’. The word that is passed to the right hand side is the word stripped of ‘yz’. Note that ‘+ ->’ has the same meaning as ‘#’ at the end of a rule (if ‘#’ is not preceded by the ‘+␣’ operator with a substitution pattern);

‘< pattern >’ (*inside substitution marker operator*) Used at most once before the ‘+␣’ operator to define a pattern that can be modified by the right hand side of the rule. For example, ‘< n > |bmp + s’ indicates a word ending with ‘nbs’, ‘nms’ or ‘nps’, where the ‘n’ can be modified by the right hand side of the rule.

3.1.3.2 Right hand side to define morphological variants

The right hand side is a listing of morphological variations to apply to the identified pattern. The most common way to create morphological variants is to define morphological variations to append at the end of the matched word or to substitute at the ending of the matched word (which has been marked on the left hand side of the rule with the '+␣' operator) with some morphological variations. The matched word (which contains markers on where modifications can take place) is called the **root term**.

On the right hand side of the rule, the meaning of a character sequence is different. The meanings are as follows

Ordered letters 'abc' stands for 'a' followed by 'b' followed by 'c', so 'abc' will be appended at the end of the root term. 'a b c' is an error;

'&' (*double operator*) As the first character of a character sequence, it specifies the doubling of the last character of the root term. The '&' must appear only at the beginning;

'_' (*empty operator*) Stands for the empty string.

LM provides other operations to perform morphological variations, as follows:

'leftadd_rightadd' (*prefix-suffix substitution operator*) Append 'leftadd' before the root term and 'rightadd' after the root term. 'leftadd' and/or 'rightadd' can be an empty string.

'< morphological_variation >' (*inside substitution operator*) Refers to the same operator on the left hand side of the rule. Different possibilities are given with this operator:

> **'< .. >' or '< .. >/' or '< .. >/_'** Substitutes the matched character sequence on the left hand side by the one defined in the right hand side. No other operation is carried out;

'< .. > /rightadd' or '< .. > /_rightadd' Substitutes the matched character sequence on the left hand side by the one defined in the right hand side, and 'rightadd' will be appended at the end of the root term;

'< .. > /leftadd_rightadd' Substitutes the matched character sequence on the left hand side by the one defined in the right hand side, 'rightadd' will be appended at the beginning of the root term and 'rightadd' will be appended at the end of the root term;

'(rule_set_name) morphological_variation' (*call operator*) Used to call the rule set (see Section 3.1.2) defined inside the parenthesis, and to apply the rules defined inside the rule set on the root term modified by the operations (referred as 'morphological_variation') following the '(rule_set_name)' operator. If no rule set name is provided, the default rule set will be called. So '(MySet)able' will append 'able' to the root term and then apply the rules in the rule set 'MySet' on this modified stem.

[**Java implementation**] In the Java version of the LM, if the rule set name is prefixed by ':' (:MySet), we are calling a rule set; if the rule set name is prefixed by '!' (!MySet), we are calling a Java rule set; finally if we do not prefix the rule set name (MySet), we are calling an ending rule set.

'TRY' (*try operator*) One can prefix the call operator with *TRY*, so if the whole rule does not produce any variant, the next rule will be tried. This can be useful when calling a user defined rule in Java using a small lexicon of exceptions and the matched word is not one of this exception. For example, '+ e r -> TRY(:ErVerb), able' will try the next rule if the call to 'ErVerb' does not produce variants, else it will stop;

'*' (***reapply operator***) As the first character of a character sequence, it indicates to call with the root stem modified by the character sequence following the operator the proper ending rule set (if existing) or the default rule set. Calling '()' (no rule set name specified) or '*' will not always produce the same effect. For example '+ s -> *_' will strip the 's' from a word ending with 's' and call the proper rule set on this root term (an ending rule set if appliable on the root term, a default rule set if not). '+ s -> ()_' will call the default rule set;

'> mode >> pattern > modif /_leftadd' (***pattern substitution operator***) Operator that will append 'leftadd' to the root stem (if left add is not empty) and then search for 'pattern' that will be substituted by 'modif'. Three substitution modes defined by 'mode' are available:

'*' substituting every appearance of 'pattern'

'<' substituting the left most appearance of 'pattern'

'>' substituting the right most appearance of 'pattern'

Each of the ways to define morphological variants can be juxtaposed on the right side of the rule by separating each morphological variant with a comma (',') and indicating the end of the rule with a colon (';').

3.1.4 Java rule set: user-defined rules with Java

Sometimes, pattern-matching is not enough to describe morphotactics. LM provides an escape mechanism to call a rule that is defined in Java. The user defined rules in Java are a special type of rule set that are written in Java instead of the rule formalism introduced in the previous section.

A **java rule set**, from a pattern-matching rule point of view, behaves just like an ordinary rule set; that is, it can be called by a rule exactly the same way a rule calls a rule set and it returns a set of morphological variants. A Java rule set can also call pattern matching rule sets.

3.1.4.1 Writing Java code

The method where the Java code for every Java rule set will be written is called `computeMorph`. The prototype of the method (with the description of each variable) is the following:

```
String[] computeMorph(String input, String arg, int depth,
    String prefix, String suffix)
```

where

`input` is the input stem to process,

`arg` is the name under which the user defined rule is called,

`depth` is a variable to get the recursion depth,

`prefix` is a variable to pass the prefix of the current stem,

`suffix` is a variable to pass the suffix of the current stem, and

Output should be an array of strings where each element of the array is a morphological variation.

When using the LiteMorph language (see Chapter 4), there is no need to write down the prototype of the function. This is done at compile time and everything between `JAVARULESET` and `ENDJAVARULESET` will be inserted in the body of this function. One has to know what variables can be used, and what types of objects have to be returned.

As we can see, if one chooses to create two Java rule sets named A and B, the associated Java code for both rules is inside the `computeMorph` method, but the call is made with a different `arg` value. The selection of the code to execute will be made by an `if-else` or a `switch` statement on the value of `arg`.

3.1.4.2 Calling a rule set

To call a rule set with Java code, one has to use the *morphWord* method whose prototype is as follows:

```
void morphWord(String word, int depth, String ruleSetName,
    Vector variants)
```

where

`word` is the input stem to process,

`depth` is a variable to get the recursion depth,

`ruleSetName` is the rule set to call, and

`variants` is a vector to store the morphological variations that are produced.

The LiteMorph language (see Chapter 4) provides a shortcut to this function:

```
void applyrule(String word, String ruleSetName, Vector variants)
```

The compiler automatically generates a call to the function with the value of `depth` incremented by 1. It is highly recommended to use this shortcut since it will check if 'ruleSetName' is an existing rule set.

3.1.4.3 Defining Table and List

One can also manipulate two useful data structures that we call Table and List. Table consist of multiple entries where each entry consist of a list of words. The Table data structure allows one to retrieve every word of an entry from any of the words in the same entry. Two kinds of tables exist, one which normalizes the number of words per entry and one which does not. Normalized Tables are mainly used to store a restricted paradigm of irregular verbs. List is an easier data structure that contains a list of words. This structure is mainly used to store a list of affixes. One can also create new lists by concatenating other lists.

[**Java implementation**] A Table is in fact an array of strings then transformed to a special kind of Hashtable. The purpose of tables is to store multiple entries were an entry is composed of multiple words. Each word of the entry is then a key to retrieve the entry. A List is just a string were each word is separated with a space.

The following list along with the grammar in Figure 3.6 define the usage of tables and lists:

Normalized Table Contains the same number of words per entry as defined in the header. Defined with:

`TABLE { [ header_1, ... , header_n ] entry_1 ... entry_m}`

`entry_i` is `word_1, ... , word_n;` (If n headers are defined, only n words are allowed)

General Table Contains any number of words per entry. Defined with:

`TABLE { entry_1 ... entry_n}`

`entry_i` is `word_1, ... , word_j;` (There is no constraint on the number j of words)

Lists Contains a list of words. Defined with:

`LIST { word_1; ... ; word_n;}`

```
1  Id          ::=  Letter { Letter | Digit | - }*
2  Word        ::=  Letter { Letter | Digit | - }*
3  TableDef    ::=  TABLE Id { [ [ Id { , Id }* ] ] { Word { , Word }* ; }+ }
4  ListDef     ::=  LIST Id { { Word ; }+ }
5  ListConcat  ::=  LIST Id = Id { + Id }* ;
```

Figure 3.6: Table and List grammar (LiteMorph formalism).

3.1.5 Exception table

Every language has exceptions or irregularities, and some are so exceptional that they cannot be encoded as rules. Most of the time, those exceptions are intensively used words that earned a very irregular pattern over the centuries. Classical examples that apply to many languages are ‘to be’ and ‘to go’. Table 3.4 highlights some of those morphological variations.

LM provides a mechanism to write this kind of exception via an **exception table** (a special case of the tables described in the previous Section). The exception table can be seen as a kind of lexicon where morphological inflections and derivations of a

Table 3.4: Some language exceptions of *to go* and *to be* in English, French and Spanish.

verb	**English**	**French**	**Spanish**
to be	be, am, are, was, were	être, suis, es, sommes, êtes, fus, serai	ser, soy, eres, somos, fui, seré
to go	go, goes, went, gone	aller, vais, allons, irai, aille	ir, voy, iba, fui, fuese

word are grouped together. For example, an entry for the verb *to go* would look like 'go, goes, went, gone, going, goings, goer, goers;'. If *gone* is the input word all the words from the entry of *to go* will be returned, since *gone* is one of the *to go* entries. The way to build an exception table is the following:

`EXCEPTIONS { entry_1 ... entry_n }`

The exception table can also be used for words where rules produce incorrect morphological variations (morphological variations that are real words but are not related to the input word), or for words that do not require processing by rules, like adverbs.

Sometimes, a word can have different meanings, and both of those meanings are correct. For example *found* can be related to *to find* or to *to found*. LM does not discriminate one of the forms. If there are two entries in the exception table, one with morphological variations of *to find* and one with morphological variations of *to found*, and if the input word is *found*, morphological variations of both entries will be returned. If *find* is the input word, however, only morphological variations from the *to find* entry are returned.

As we will see in 3.1.6, if an input word matches a word in the exception table, all the related words from the exception table are returned, but no rules are tried. This is the desired behaviour, since an exception must not be processed by rules if we know it is an exception.

[Java implementation] In the Java version of LM, the exception table is an array of string where each element is an entry. Also, words are separated by blanks (‘ ’) instead of comas (‘,’).

3.1.6 Lightweight Morphology flow

The order in which each component is processed is important, since not all the rules are tried and since the exception table has a higher precedence than the rules. The flow of LM is as follows:

1. Check if a word in the exception table matches the input word. If a word matches, then all the morphological variants from each entry containing the input word are retrieved from the exception table and no rule will be tried. If this step produced variants, exit and return the morphological variants, else go to 2.

2. Check if an ending rule set is defined and if the input word ends with this predefined ending. In this case, the starting rule set will be the ending rule set, otherwise the starting rule set will be the default rule set. The starting rule set becomes the current rule set. Go to 3.

3. If the current rule set is a pattern matching rule set, go to 4. If it is a Java rule set, go to 5.

4. See if the input word matches a pattern in the current rule set. The rules are tried from first to last. When a rule succeeds, no further rules are tried (unless the *TRY* operator is used) and the matching rule does everything asked on the right hand side. Go to 6.

5. Execute the Java code on the input word. If the method returns `null`, the Java rule set will be considered to have failed. Go to 6;

6. If the rule explicitly calls a specific rule set, then the current rule set will be this specific rule set. Also, the input word for this specific rule set will be a modified stem (a word resulting from the morphological variation operation defined in the calling rule). Go to 3 if a call to a rule set is made, else exit and return all morphological variants found so far.

3.2 Xerox Finite State Morphology representation

Xerox Finite State Morphology (XFSM) [3] is a set of tools using **finite state automata** to perform lexical/morphological analysis. We investigated the possibility of using XFSM to implement a Lightweight Morphology. The complete results of this investigation is available in [33]. This section will only cover the most important points.

A **finite state transducer** is a two level finite state automata where the first level, called the **upper language**, is the input word and the second level, called the **lower language**, is the output. This difference between input and output is purely artificial since the lower language can be the input and the upper language can be the output.

Finite state transducers are used to map a set of letters into another set of letters. Different uses are made of finite state transducers: translation from a natural language to another (the translation is bidirectional so it does not matter which one is the upper language), morphotactics (see Figure 3.7 where from the **surface form** *jolies* we can produce the analysis *+adj +fem + plur*) or pronunciation.

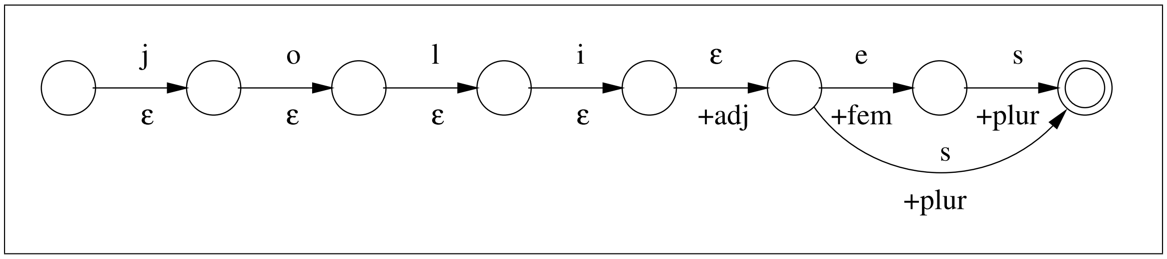


Figure 3.7: Morphological finite state transducer for the French word *joli*.

One interesting feature of XFSM is its representation of replacement rules. That is exactly the kind of behaviour we are facing with the pattern matching rules in Lightweight Morphology, and the syntax used in XFSM is quite similar to the one used in Lightweight Morphology. Our goal was to see if the replacement rule formalism could successfully represent the pattern matching rules from LM and if the finite state technology produced the same set of variants. We did not try to create rule 'thinking' in XFSM. We used a straightforward approach, translating a subset of the English LangLM pattern matching rules into a set of rules in the XFSM formalism that we thought would have the same behaviour. Figure 3.8 contains an example of such a transformation.

Different problems arose when testing this representation. The finite state automata produced from the XFSM rules suffered from a 'recursion problem', producing words like **controlnessesed* showing that first *-nesses* was appended to control, then *-ed* was appended to **controlnesses*. It is indeed very hard to control how variants

```
.aeiouy + ->  s,er,ers,est,ed,ing,ings,ly,ness,
              nesses,ment,ments,less,ful;
```

(a) LiteMorph formalism.

```
[0] ->  [s]|[e r]|[e r s]|[e s t]|[e d]|[i n g]|[i n g s]|
        [l y]|[n e s s]|[n e s s e s]|[m e n t]|[m e n t s]
        |[l e s s]|[f u l] || $[a|e|i|o|u|y] _ .#.
```

(b) XFSM formalism.

Figure 3.8: Translating LiteMorph formalism to XFSM formalism.

are produced with XFSM. We tried different approaches to constrain the context in which the rule should apply but without useful results. Lightweight Morphology offers complete control on how variants should be produced from a pattern. XFSM tries to produce as many variants as it possibly can following the patterns. This includes having the morphological variants produce other morphological variants.

Some attempts were made to control the production of variants, mainly by implementing a kind of rule set, but the time efficiency dropped drastically.

The other problem of XFSM is that is does not support the notions of exception lists, rule sets and user defined rule sets (it was not designed to produce multiple morphological variants). Even if XFSM could successfully represent a subset of the pattern matching rules, it would fall short in trying to support the missing features. Our conclusion is that XFSM is not suitable to represent a Lightweight Morphology.

CHAPTER 4

LiteMorph: a language for Lightweight Morphology

LM has already been implemented by Sun Microsystems in Java. In Sun's version, there is no separation between the core engine of the Lightweight Morphology, (the *LangLM processor* that will parse and execute the rules) and the language modules (*LangLM* that define the rules for a language). Each LangLM is defined in a class that extends the Lightweight Morphology engine class. There are two problems with this approach:

1. A new LangLM has to be defined in Java. Some methods have to be redefined (overloaded) and some variables have to be initialized in a certain way that deal with Java syntax and knowledge of the parent class. This complexity is not needed and can be automated.
2. LangLM is written in Java, so there is no error checking for rules (syntax, use of non-initialized variables) until the code is compiled and the engine is run.

4.1 LiteMorph as a language for Linguists

Since the framework for LM was already present and since it appears not possible to easily use another representation for LM (see Section 3.2), we decided to ease the writing of a LangLM by introducing a formal representation. The aim is to have a representation that is close enough to the existing representation (i.e. the pattern matching rule syntax) but that would avoid Java syntax (i.e. defining rule sets inside arrays of string) and unnecessary complexity for linguists.

The best representation we could think of was to create a language that would be translated into its equivalent in Java. The main benefits of this approach are:

1. No unnecessary Java syntax has to be used when writing a new LangLM or modifying an existing one.
2. Most of the initialization in the Java code (i.e. creation of predefined methods, call to specific methods, initialization of some variables) can be automated at compile time.
3. Error checking can be performed at compile time, especially the pattern matching rule syntax (see Section 3.1.3), avoiding run-time debugging to detect such errors.
4. LM can be implemented in different computer object-oriented languages (e.g. C++, Python) and the existing LangLM can be translated (with the appropriate compiler) to the computer language the LM framework is using.
5. There is no need to rewrite a LM framework to perform testing.

4.2 Abstracting Java with LiteMorph

We abstracted the Java code into the LiteMorph language via the following steps:

1. Defining a context free grammar for the pattern matching rules.
2. Adding expressiveness to the grammar by adding features such as rule set support or language definition.
3. Including support for Java and control integration with LiteMorph.

4.2.1 LiteMorph pattern matching rules grammar

This step is the most important one and also the most complicated. The sources to define the grammar are of four kinds: (1) documentation to build a Java Lightweight Morphology [43], (2) existing LangLM in English, German and Spanish, (3) Java source code to parse the pattern matching rules, and (4) William Woods, creator of Lightweight Morphology. The first two sources appeared to be the most useful, along with William Woods to check if our grammar was as he intended LM to be.

Our grammar construction method was straightforward. First, we tried to build the grammar with the available documentation. Then, we tested our grammar by parsing the existing LangLM. We iterated the steps of building the grammar and testing until we could successfully parse all three existing LangLMs. Not all documented features of the pattern matching rules are used in the different LangLM and the documentation does not fully explain the allowed syntax. We had to produce a 'test LangLM' to see if the grammar could parse the document features that were not used. This 'test LangLM' was also used to see which undocumented syntax was accepted by the Java LM.

The resulting grammar (which can be seen in Figure 3.4) successfully parsed the three existing LangLMs.

4.2.2 Extending LiteMorph grammar

Parsing pattern matching rules is a big step but not enough to cover a full LM. There are multiple features that need to appear, namely:

Language definition The language definition of the LangLM, consisting of an identifier. The identifier is used to define the name of the class that contains the LangLM (the English LangLM will have the class name 'LiteMorph_en').

Debugging options LM has run-time debugging options that can be turned on or off in the LangLM.

Letter variable initialization As we have seen, pattern matching rules can use letter variables and we should be able to initialize the value of such variables.

Exception table One of the features of LM defined in Section 3.1.5.

Rule sets One of the features of LM defined in Section 3.1.2.

Comments Useful for every language.

The features are added either as keyword-value pairs or as keywords and can be seen in Figure 4.1.

The grammar follows the order of the previously introduced features that need to be included. The grammar also follows the formalism that was introduced in Section 3.1. The ∞ symbol refers to 'anything' so any kind of token can be matched by this symbol. The '\n' has the usual meaning of new line. Therefore one can see that the comments are defined using the usual C (or Java) syntax, but will not appear in the definitive LiteMorph grammar because they are removed at the lexical analysis step.

```
1   Language            ::=  LANG = Id ;
2   Options             ::=  DEBUG ; | TRACE ;
3   DefLetterVariable   ::=  LetterVariable = UnOrderedList ;
4   Exceptions          ::=  EXCEPTIONS { { ExceptionsList }+ }
5   ExceptionsList      ::=  Word { , Word }* ;
6   RuleSetNormal       ::=  RULESET Id { { Rule }+ }
7   RuleSetEnding       ::=  RULESET Id ENDING Id { { Rule }+ }
8   RuleSetDefault      ::=  RULESET DEFAULT Id { { Rule }+ }
9   JavaRuleSet         ::=  JAVARULESET { Modifiedjavacode }* END-
                             JAVARULESET
10  Id                  ::=  Letter { Letter | Digit }*
11  LetterVariable      ::=  $ Letter { Letter | Digit | - }*
12  UnOrderedList       ::=  Letter { Letter | Digit }*
13  Word                ::=  Letter { Letter | Digit | - }*
14  Comments            ::=  // ∞
                        |    /* { ∞ [ \n ] }+ */
```

Figure 4.1: Extension to LiteMorph grammar.

4.2.3 Adding Java support to LiteMorph

One of the most interesting features is the possibility to use Java code to define our own morphological rules. While this is quite easy to do when LangLM is using a Java formalism (everything is using Java syntax so Java rule sets are just part of it), it is challenging to implement such features in LiteMorph in a linguist-friendly way. The challenges include:

- Create a Java parser.
- Include a grammar (Java) in another grammar (LiteMorph) with possible collision.
- Deal with the Interaction between Java rule sets and other rule sets (check that called rule sets exist).
- Interact between Java rule sets and defined tables and lists.

We decided that the best way to approach such problems was by building a second parser for the Java code with existing resources (an open source Java grammar). We avoided a lengthy LiteMorph grammar with all the possible collisions that could arise from mixing two different grammars. We modified the Java grammar to include built-in methods (such as `applyrule()`) and statements (e.g. `RULE(rule_set_name)`) that ease the interaction between Java rule sets and other rule sets.

The interaction between tables or lists and Java rule sets appeared impossible to resolve in a satisfactory way. If it was fully integrated in a LiteMorph formalism, many of the features of the underlying data structures of those objects would have been lost, or we would have hidden too much from the underlying data structures. We decided not to provide a particular modification to the grammar except a double 'initialization' with `TABLE table_name` or `LIST list_name` to be used in the Java code to check for the existence of the variable names. This initialization is optional and does not provide the features of a symbol table (check for each variable its type, value and scope).

4.2.4 Putting it together

The resulting LiteMorph grammar is presented in Appendix A. The grammar also includes tokens.

4.3 Translating LiteMorph into Java

The previous sections introduced some concerns about the Java implementation. This section focuses on the link between the LiteMorph formalism and the Java formalism to write LangLM, presenting how LiteMorph should be translated into Java code.

4.3.1 Java LightweightMorphology implementation

The Java implementation is separated in two classes `LiteMorph` and `LiteMorphRule` (see Figure 4.2). The `LiteMorph` class is an abstract class and each new LangLM specification should inherit from `LiteMorph`. The convention is to name the new subclass `LiteMorph_xx` where `xx` stands for the two letter ISO language code (e.g. `en` for English, `fr` for French). The `LiteMorphRule` class is a part of the LangLM processor. More precisely, this class is responsible for creating morphological variants from pattern matching rules, being a 'pattern-matching rule processor'. The other part of the LangLM processor is in `LiteMorphRule`.

We will present how one should implement the subclass of `LiteMorph` to define a new LangLM specification, by going through the previously defined concepts with LiteMorph formalism.

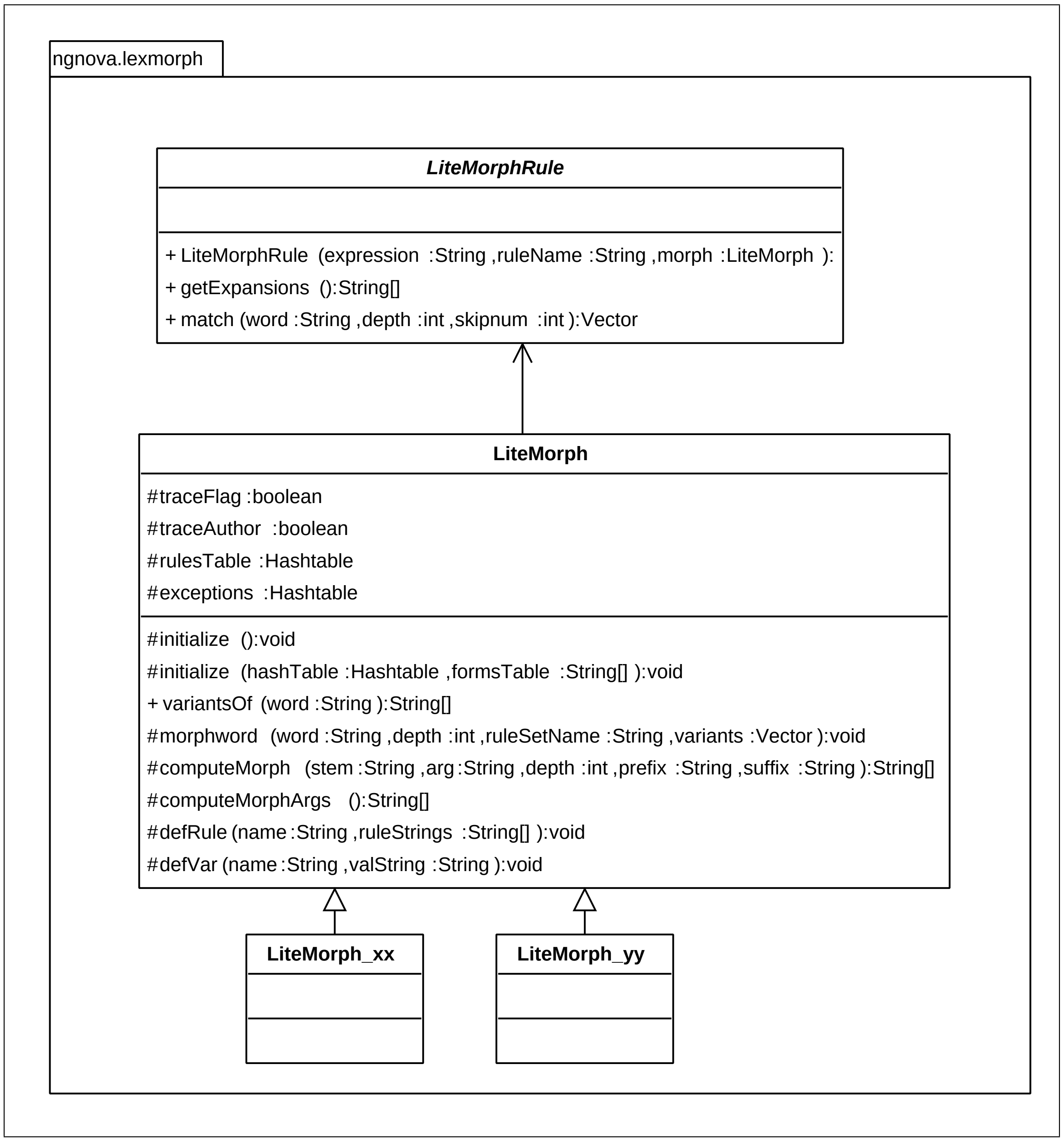

ngnova.lexmorph
LiteMorphRule
+ LiteMorphRule (expression :String ,ruleName :String ,morph :LiteMorph):
+ getExpansions ():String[]
+ match (word :String ,depth :int ,skipnum :int):Vector
LiteMorph
#traceFlag :boolean
#traceAuthor :boolean
#rulesTable :Hashtable
#exceptions :Hashtable
#initialize ():void
#initialize (hashTable :Hashtable ,formsTable :String[]):void
+ variantsOf (word :String):String[]
#morphword (word :String ,depth :int ,ruleSetName :String ,variants :Vector):void
#computeMorph (stem :String ,arg :String ,depth :int ,prefix :String ,suffix :String):String[]
#computeMorphArgs ():String[]
#defRule (name :String ,ruleStrings :String[]):void
#defVar (name :String ,valString :String):void
LiteMorph_xx
LiteMorph_yy

Figure 4.2: Lightweight Morphology Java implementation UML diagram.

Rule sets (except Java rule sets) A rule set is defined as an array of string in the `initialize()` method. Each element of the array is a pattern matching rule. The variable of the array of string has to be mapped to its rule set name with the method `defRule(name, ruleStrings)` storing the association between rule set name and rules in the `rulesTable` hashtable. The rule set name is responsible for the behaviour of the rule set. If it starts with ':' then it is a normal rule set, else (with the exception of '!', used for the Java rule sets) it is an ending rule set. For the ending rule set, the rule set name defines the ending pattern. For example a normal and an ending rule set (ending is *s*) will be defined as follows:

```
public void initialize() {
...
String[] normalRuleSet = { "rule_1", ...  , "rule_n"};
String[] sRuleSet = { "rule_1", ...  , "rule_m"};
defRule(":ruleSet1", normalRuleSet); // rule set name:  ruleSet1
defRule("s", sRuleSet); // rule set name and ending pattern:  s
...
}
```

Java rule sets Java rule sets are defined in the `computeMorph` method and the Java rule sets names are defined in the `computeMorphArgs` method. The method `computeMorph` knows which Java rule set name should be used with the `arg` parameter. The `computeMorphArgs` is used only to check calls from a pattern matching rule to a Java rule set. See Section 3.1.4 for available methods in Java rule sets.

Exception table Exceptions lists are defined in an array of string in the method `initialize()`. Each exception entry (set of morphological variants) is an element of the string array. The exception list is then transformed in the Hashtable `exceptions` calling the `initialize(hashTable, formsTable)` method. An exception table will be defined as follows:

```
public void initialize() {
...
String[] exceptionTable = { "entry_1", ...  , "entry_n"};
initialize(exceptions, exceptionTable);
...
}
```

Letter variable Letter variables are initialized calling `defVar(name, valString)` method in the `initialize()` method and storing the association in the Hashable `rulesTable` . A letter variable will be defined as follows:

```
public void initialize() {
...
defVar("$Vowel", "aeioy");
...
}
```

Note: In the LiteMorph to Java compilation, we replace each letter variable by its actual value.

Table Defining a Table should be done by first specifying a new `private static Hashtable` object as an attribute of the current class. The process is then exactly the same as defining an exception table.

List Defining a List should be done by defining a `private static String[]` as an attribute of the current class.

4.3.2 Creating a compiler compiler

Now that we have a working grammar, we need a tool to transform the LiteMorph grammar into a program that will be able to parse LiteMorph files. This kind of tool is called a compiler compiler. One well known compiler compiler is the duo Lex and Yacc [27] (or their open source equivalent Flex and Bison). The output of Lex is a **lexical analyzer** written in C, a program that will break the input into individual tokens. Yacc will produce a **parser** written in C from a grammar, a program that will check that a document is valid given the grammar.

For our project, we decided to use **JavaCC** [22] with a modern compiler approach described in [1]. The advantages of JavaCC over Lex/Yacc are the following:

1. The parser generator and lexical analyzer generator are combined in the same tool. Language lexical and grammar specifications are defined in a single file.
2. JavaCC is an LL(k) parser generator (this is an advantage for our grammar).
3. JavaCC is written in Java and outputs Java code, natively supporting Unicode characters.
4. Abstract syntax tree building can be automated with JJTree [22] or JTB [21].
5. JavaCC is able to switch to different lexical states, therefore having different interpretation of tokens depending on the state.

We converted the grammars (LiteMorph grammar and Modified Java 1.4 grammar) into JavaCC grammars (see Appendix B). The definition of tokens is problematic with the LiteMorph grammar since identifier tokens (a letter followed by any kind of character) and word tokens (a letter followed by a letter or a digit) are overlapping. Using lexical states is a solution but it is better in such cases to rewrite the grammar so as to correctly define an identifier or a word. We defined different tokens where one token was included in the other (e.g `LETTER1` with only letters and digit allowed and `LETTER2` with letters, digit, '-' and '_' allowed) and assigned one or more of those tokens to a grammar production, one named *Id* and the other one called *UnorderedLetters*. The *Id* production contains all tokens that match an identifier definition; the *UnorderedLetters* production contains all tokens that match a word definition.

This problem drove the use of capital letters for keywords so if someone uses *try* as a word in an exception table or as a pattern in a pattern matching rule, the compiler will not complain about *try* being a keyword. This problem also influenced us in separating the LiteMorph grammar from the modified Java grammar (avoiding words to be analyzed as Java keywords).

Two possibilities can be used to perform the LM to Java compilation step. The first one is to include lexical and semantic actions directly in the JavaCC code. The second solution is to generate an **abstract syntax tree**, a tree where each node corresponds to a terminal or non-terminal of the grammar. A set of visitors (using the **visitor** pattern [13]) is used to traverse the abstract syntax tree and perform interpretation depending on the type of visitor and the type of node. The second approach is preferred since we separate the interpretation of the grammar from its syntax.

We used JTB to automatically build abstract syntax trees. We wrote our own visitors to perform the different interpretations. For each grammar, one visitor is needed to initialize symbol tables and other initialization information, and a second visitor is used to translate the LiteMorph code into Java code, following the Java formalism defined in 4.3.1.

4.3.3 Architecture of LiteMorph2Java compiler

Figure 4.3 presents the global architecture to build LM2Java and a functional LM engine.

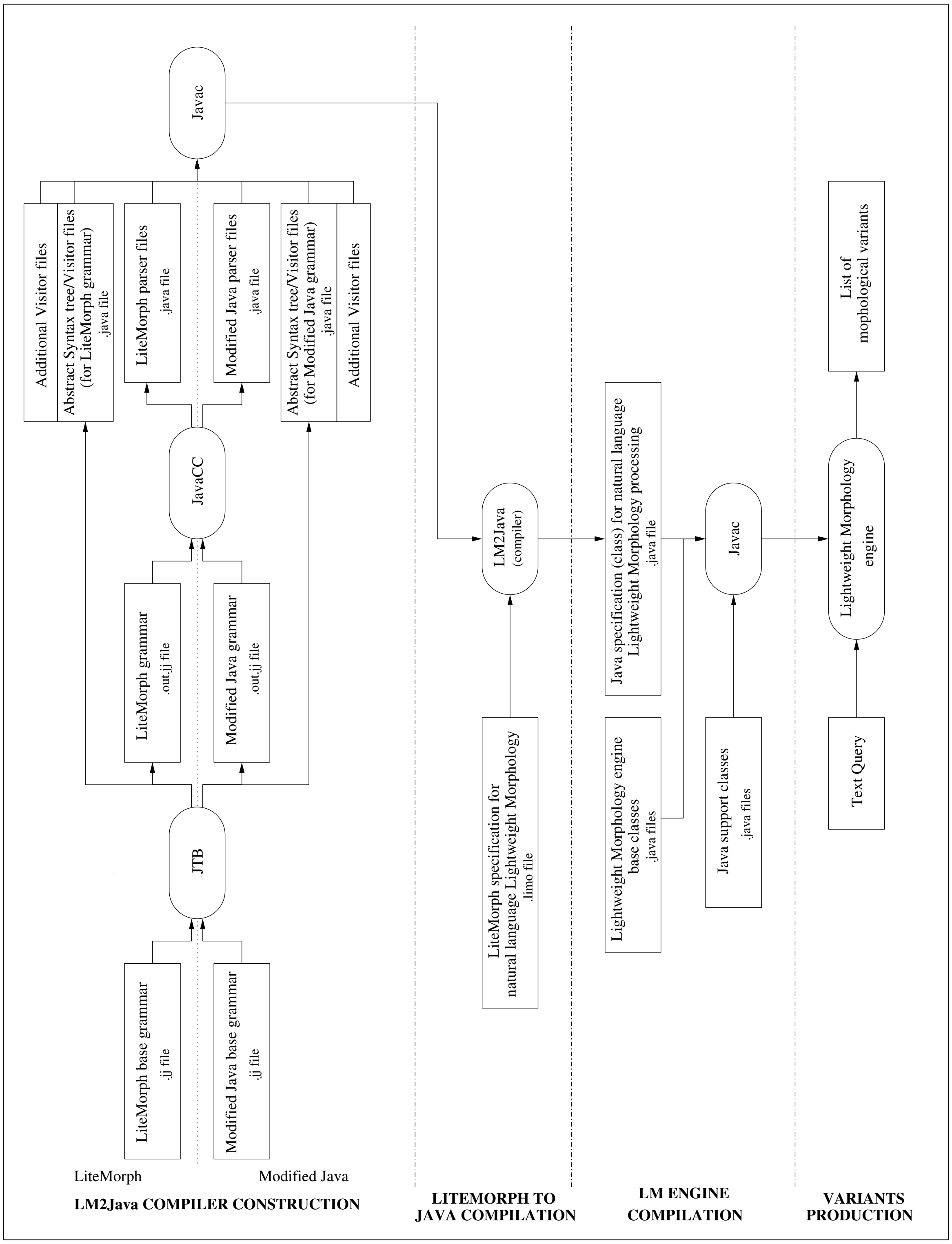


Figure 4.3: LM2Java building architecture.

CHAPTER 5

French Lightweight Morphology

French can be a difficult language to learn. The reason it is difficult is summarized in this well know French sentence 'l'exception qui confirme la règle'[1]. Almost every French grammatical rule is accompanied by its list of exceptions, a few words that will follow a different rule. We focus on different components of French morphology that are useful for French LangLM specification.

5.1 Essentials for French morphology

This section presents the different morphological and grammatical considerations useful for building a French Lightweight Morphology.

The knowledge of French can be divided in three (overlapping) categories: '**grammaire**', '**orthographe**' and '**conjuguaison**'. 'Grammaire' (grammar in English) is the set of rules used to build correct statements. 'Orthographe' (spelling in English) is the set of rules that define the way of writing words in a language. 'Conjugaison'

[1]the exception that confirms the rule

(conjugation in English) is the set of inflexions of the verbs. All forms of a same verb constitute its **paradigm**. As we previously pointed out, these three categories are overlapping. For instance, the 'orthographe' is divided into '**orthographe lexicale**' (lexical spelling) the way to spell a particular word independently from its context in a phrase and '**orthographe grammaticale**' (grammatical spelling) the way to graphically indicate variable elements from a word (e.g. plural, conjugation of a verb).

5.1.1 Considerations on French inflection

The two categories we focus on for Lightweight Morphology are conjugation and spelling (more precisely, grammatical spelling). Indeed, the most prolific way to produce morphological variants in French is by building the paradigm of every verb (about 45 morphological variants for each verb), and the result of (masculine, feminine) $\times$ (singular, plural) for nouns and adjectives (each noun or adjective has in general 4 morphological variants, some can have up to 6 morphological variants). These two ways of creating morphological variations represent the inflectional morphology of French, and should be the core of every Lightweight Morphology.

In French, the verbs are usually classified in three groups plus the two auxiliary verbs *avoir* (to have) and *être* (to be). The first group contains all verbs whose infinitive ending is *-er* except for *aller* (to go). They are the most numerous and almost all neological verbs are constructed following this conjugation. The second group contains all verbs whose infinitive is ending with *-ir* and whose stem remains constant by inflection. There are more than 300 verbs in this group and some neological formations are constructed on this model. The third group contains all irregular verbs. The infinitive endings encountered in this group are *-ir*, *-oir* and *-re* (and the only irregular verb ending with -*er*, *aller*).

5.1.2 Considerations on French derivation

The other interesting topic for Lightweight Morphology is the derivational morphology. Although very interesting since it can reunite a verb with its noun and its adjective (e.g. in French *manger* $\longleftrightarrow$ *mangeur* $\longleftrightarrow$ *mangeable*), this process is often more difficult to formalize since there is no standard rule to perform such an operation. To convice oneself of this, one should look at an etymology book of a language and see how derivations are produced. For French, different influences can be identified: Latin (the main influence), Greek (Greek and Latin influenced each other), Italian, Spanish, English and many more. Such an example is given by *fluctuation* and *flottaison*, both originating form the Latin word *fluere, fluctum*, which gave the French word *flotter* (to float). More generally, different suffixes can have the same meaning. For example the suffixes -*aison*, -*ision*, -*ation* and -*ition* have the same meaning when appending to a verb: marking the action or the result, that is the state. Also *ambition* is derived from the verb *ambitionner* (not **amber*) but *abolition* is derived from the verb *abolir*.

It is possible to identify productive suffixes. By productive suffixes we mean suffixes that (1) will produce a meaningful noun or adjective when appended to a word stem, and (2) will produce a meaningful verb when stripped from a noun or an adjective (the verb is lexically related to its generator). Some examples of these suffixes in French are -*able* (producing an adjective whose meaning is ‘that can be’) and -*ion* (producing a noun whose meaning is ‘the resulting action of’).

5.1.3 Considerations on French exceptions

As we pointed out before, exceptions are an essential part of the French language. Some of these exceptions can be rules by themselves: the feminine form of a noun is generally formed by adding -*e* at the end of the masculine noun (*apprenti* $\longrightarrow$ *apprentie*). Masculine nouns ending in -*eur* can have three different kinds of feminine noun, by transforming -*eur* into -*euse* (*vendangeur* $\longrightarrow$ *vendangeuse*), -*rice* (*instituteur* $\longrightarrow$ *institutrice*) or *eresse* (*chasseur* $\longrightarrow$ *chasseresse*). This behaviour is quite common in French where there is no certainty over how inflection has to be done depending on the ending pattern (*adjoint* $\longrightarrow$ *adjointe*, but *chat* $\longrightarrow$ *chatte*, *confus* $\longrightarrow$ *confuse*, but *bas* $\longrightarrow$ *basse*).

Other exceptions are found in verbs. The third group contains both very used verbs (*aller*, to go, *voir*, to see, *faire*, to do) and obsolete verbs or forms of verbs (*ouïr*, to hear, *falloir*, to be necessary, *gésir*, to lie), mainly used in expressions or only in some tenses. The Bescherelle [4], a well-known resource for French conjugation, counts 65 verbs used as paradigm models to inflect about 370 verbs. It is not trivial to build the 370 paradigms from the 65 models since (1) verbs are derived from the model verb by prefixation which can sometimes be complicated (the prefix *re-* can be found under different forms: *tenir* $\longrightarrow$ *retenir*, *souvenir* $\longrightarrow$ *ressouvenir*, *asseoir* $\longrightarrow$ *rasseoir*, *admettre* $\longrightarrow$ *réadmettre*, *épandre* $\longrightarrow$ *répandre*) and (2) a model can be used for totally different verbs (*sentir*, to smell/to feel, and *partir*, to leave, are inflected in the same way).

It is also interesting to introduce some consideration on etymology for derivations. One interesting question is whether a word is a morphological derivation or a morphological generator. For example, the French word *mettre* (to put) has the inflected form *mis* for the past participle[2] which by another inflection produces the feminine past participle *mise* (put). *Mise* (a bet) is also a noun, an example of nominalization and a perfect example of polysemy. What is more interesting is the derivational neologism *miser* (to bet, to gamble) from *mise*. Note that *mettre* is an irregular verb and *miser* is a regular verb (like all neologism verbs). These two verbs have different meanings but a common morphological form *mise* that comes from the etymology. Summarizing, we have

$$\begin{tikzcd}
\text{mettre} \arrow[r, "\text{inflection}", "\text{(tense)}"'] \arrow[rrr, bend left, "\substack{\text{derivation}\\\text{(nominalization)}}"] & \text{mis} \arrow[r, "\text{inflection}", "\text{(gender)}"'] & \text{mise} \arrow[r, "\text{derivation}", "\text{(nominalization)}"'] & \text{mise} \arrow[r, "\text{derivation}", "\text{(neologism)}"'] & \text{miser}
\end{tikzcd}$$

but the meaning relation would be

$$\begin{tikzcd}
\text{mettre} \arrow[r, leftrightarrow] \arrow[rr, leftrightarrow, bend left, "\times" marking] & \text{mise} \arrow[r, leftrightarrow] & \text{miser}
\end{tikzcd}$$

Finally, some exceptions cannot be encoded as rules. Such examples can be found with nouns where the feminine is not based on the masculine. This is particularly used in feminines of animals (e.g. *bouc* $\longrightarrow$ *chèvre*, goat in English).

5.1.4 French spelling rectifications

A living natural language is a language that adapts itself to the modifications introduced by its usage, in the following ways:

[2]it can also be an adjective, but that is not the point

1. Harmonizing the actual pronunciation of words and their lexical spelling,
2. Adapting the grammar to effective habits,
3. Modifying the meaning of its lexeme, and
4. Adding or borrowing lexemes.

From a morphological point of view, only the first point is of real interest. The other points can also impact a lightweight morphology in the following different manners:

- Two lexemes sharing the same etymological roots can have different meanings. For example *près* (near) and *presser* (to press) come from the same Latin root *permere, pressum* (to tighten);
- A derivational word can lose its etymological meaning (*pomme* $\longrightarrow$ *pommade* but a pommade is not exclusively made from apple and grease anymore);
- Lexemes from a foreign language can follow a different grammatical spelling (e.g. *rugbyman* has the plurals *rugbymans* and *rugbymen* in French);

The last spelling rectification was recommended by *le Conseil supérieur de la langue française* in 1990 [7]. Arrivé in [2] presents an interesting analysis of spelling rectification in general, and the last one in particular. Since this rectification is not a reform, both spellings can be used nowadays (the one used before 1990 and the one introduced with this rectification). Lightweight Morphology has to be able to generate spellings arising in all documents, both past and present.

5.2 Representing French LangLM with LiteMorph

French and Spanish are both languages derived from Latin. The existing Spanish LangLM was used as a base to build the French Lightweight Morphology. The previous considerations decided the way the French LangLM should be built:

1. Irregular verbs (3$^{\text{rd}}$ group verbs) should be hard coded in the exception table. The list of irregular verbs is finite (370 verbs) and the prefixation is sometimes too complicated to be encoded as (Java) rules (see Figure 5.1).

2. Articles and adverbs, along with some inflectional irregularities, should be included in the exception table (see Figure 5.1).

3. Adjective and noun inflections are similar so they are grouped in four rule sets, one for feminine each pair (gender, number), for example (, plural) (see Figure 5.2).

4. Inflection of verbs for simple tenses is divided in rule sets according to the usual French verb classification (1$^{\text{st}}$/2$^{\text{nd}}$ group and tense, see Figure 5.3).

5. Bidirectional derivation (verb $\longleftrightarrow$ noun and verb $\longleftrightarrow$ adjective) is made with the suffixes -*age*, -*ance*, -*ment*, -*eur*, -*ion*, and -*able*. Each directional derivation is grouped in a rule set (see Figures 5.4 and 5.5).

6. The default rule set role is to capture the more general patterns and each specific rule set is used to produce accurate morphological variants (see Appendix C).

Grouping each kind of derivation into a separate group of rule sets makes it easy to add or remove a specific derivation for processing. Rule sets are also helpful to

```
EXCEPTIONS {
...
asseoir, assoir, assieds, assied, asseyons, asseyez, asseyent,
   asseyais, asseyait, asseyions, asseyiez, asseyaient, assis, assit
   , asssîmes, assîtes, assirent, assiérai, assiéras, assiéra,
   assiérons, assiérez, assiéront, assiérais, assiérait, assiérions,
   assiériez, assiéraient, asseye, asseyes, assisse, assisses, assît
   , assissions, assissiez, assissent, asseyant, assise, assises,
   assois, assoyons, assoyez, assoyent, assoyais, assoyait, assoyions
   , assoyiez, assoyaient, assoirai, assoiras, assoira, assoirons,
   assoirez, assoiront, assoirais, assoirait, assoirions, assoiriez,
   assoiraient, assoie, assoies, assoyant;
beau, bel, beaux, bels, belles;
çà;
...
}
```

Figure 5.1: Sample entries from the French LangLM exception table

group conceptually identical morphotactics (verbs inflection grouped in paradigms, noun and adjectives inflection grouped by gender and number). Other examples of rules and rule sets are given in Appendix C.

The following resources were used to build the French LangLM:

- French verbs paradigms: *Bescherelle* [4].
- Lexical spelling and grammar: *le Bled* [5], *le Grevisse* [16], *le Dictionnaire de l'Académie Française* [12, 11] and their online versions [9, 10].
- Etymology: *les Racines et la signification des mots Français* [8].

```
RULESET NomAdjSingFem {
   // generates adjectives and nouns
   // assumes the input is singular feminin form of adjective or noun

   .$Letter + è r e -> er,ers,ère,ères; // (printanier, printanière)
   .$Letter i + g n e -> n,ns,gne,gnes; // (bénin, bénigne)
   .$Letter e t + t e -> _,s,te,tes; // (violet, violette)
   .$Letter + è t e -> et,ets,ète,ètes; // (complet, complète)
   .$Letter + a l e -> al,aux,ale,ales; // (initial,initiale)
   .$Letter + e l l e -> eau,el,eaux,els,elle,elles;
   .$Letter e i l + l e -> _,s,le,les; // (spirituel,spirituelle, pareil,pareille
      )
   .$Letter + e r e s s e -> eur,eurs,resesse,eresses; // (vengeur,
      vengeresse)
   .$Letter + o r e s s e -> eur,eurs,oresse,oresses; // (vengeur,
      vengeresse)
   [...]
   .$Letter + a s s e -> e,es,asse,asses; // (blondasse, blonde)
   .$Letter + e s s e -> _,e,s,es,esse,esses; // (adj finissant par −e,
      nom finissant par −e pauvre,pauvresse)
   .$Consonant + s s e -> s,x,sse,sses; // (bas,basse, faux,fausse)
   .$Letter + e u s e -> eur,eux,eurs,euse,euses; // (rieur,rieuse)
   .$Letter s + e -> _,e,es; // (ras,rase)
   .$Consonant l|n + & e # -> _,s,&e,&es; // (net,nette, gentil,gentille, pâlot
      ,palotte)
   .$Letter + è v e -> ef,efs,ève,èves; // (bref,brève)
   .$Letter + v e -> f,fs,ve,ves; // (naïf,naïve)
   .$Letter + r i c e -> eur,eurs,rice,rices; // (créateur,créatrice)
   .$Letter + e r e s s e -> eur,eurs,eresse,eresses; // (vengeur,
      vengeresse)
   .$Letter + c e -> x,ce,ces; // (doux, douce)
   .$Consonant + c q u e -> c,cs,cque,cques; // (grec, grecque)
   .$Consonant + q u e -> c,cs,que,ques; // (turc,turcque)
   .$Letter g + u e -> _,s,ue,ues; // (long,longue)
   .$Letter + e -> _,s,e,es; // (adj finissant par −e, joli,jolie, règle par
      défaut)
   .~s|x|z # -> s; // (absurdité)
}
```

Figure 5.2: Rule set to process singular feminine nouns and adjectives with sample rules.

```
RULESET IrPres {
   //generate present forms for −ir verbs, from verb stem
   //Assume input has stripped infinitive −ir

   .$Letter # -> is,it,issons,issez,issent; // (finir)
}
```

Figure 5.3: Rule set to create indicative present form for 2nd group verbs.

```
RULESET AgeNounToVerb {

   .$Letter + i s s a g e -> (RegIrParadigm)_, // (attérissage)
   (RegErParadigm)iss; // (glissage)
   .$Letter + c a g e -> (RegErParadigm)qu; // (blocage)
   .$Letter + a g e -> (RegErParadigm)_, // (abordage)
   (RegIrParadigm)ag; // (avantage)
}
```

Figure 5.4: Rule set to process -*age* derivation from noun forms to verb forms.

```
RULESET IrAgeNoun {

   .$Letter # -> issage,issages;

}
```

Figure 5.5: Rule set to process -*age* derivation from verb forms to noun forms.

CHAPTER 6

Testing

Testing is an essential part of building a successful LangLM. One has to check that the rules produce correct morphological variants and avoid spurious morphological variants. We do not consider meaningless morphological variants (words that are not part of the language) since they do not affect the way LM should work.

The usefulness, in terms of information retrieval, of Lightweight Morphology in general, and the French LangLM in particular, is another important part of this investigation.

6.1 Validating the rules

The difficulty of validating a LangLM comes from the enormous number of words available in a language (theoretically infinite). If a French dictionary reports 50,000 lexemes, the French language will probably contain between 400,000 and 1,000,000 words when including inflections and non-indexed lexemes. Even if one carefully restranscribes the information provided in French resources (mainly lexical spelling and verb paradigms), it is impossible to say if the rules are correct for every word. Looking at the morphological variants of 1,000,000 words is an impossible task, both in terms of time (if we suppose that we can check the validity of morphological variants of a word in 10 seconds, it would take between 3 to 7 1/2 years), repetition (if during the testing process we modify one or more rules, we have to check again all tested words) and knowledge (we have to know for each word its valid morphological variants).

Two solutions can be used. The first one, sampling, consists of taking a limited number of words and checking if the morphological variants are correct. The second one tries to introduce measures that indicate how the LangLM behaves.

Our collection used for testing is the aligned Hansard of the 36th parliament of Canada from 1997 in French and English [14], referred in the following sections as the Hansard. It consists of 533 documents and a total of 19,999,604 tokens for the English version and 22,801,063 tokens for the French version.

6.1.1 Sampling

Sample testing is tedious work for Lightweight Morphology. It should be considered as part of the debugging step but it does not provide extensive information on the validity of the rules. Sampling was performed for the French LangLM has follows:

1. Select a corpus of words from the Hansard constituting the sample to test.
2. Run the Lightweight Morphology engine on the sample.
3. Examine the output with three goals in mind:

   (a) Check if the variants are correct (spelling, morphological validity);

   (b) Check that no unrelated words are part of the variants;

   (c) Check if variants are missing (inflectional and derivational).

4. Modify or add rules, then retry the Lightweight Morphology on the sample.

Of course, the sample should not always be the same, avoiding a biased LangLM. Sampling does not provide any measurable information on the quality of a LangLM.

6.1.2 LangLM measures

We pick up on our analogy of LM as an equivalence class. A set of morphological variants is not properly an equivalence class (for example, *found* is both in the equivalence class of *to find* and *to found*) but for the majority of the words, it can be considered as such.

The goal of LM is to approximate each equivalence class by building a cluster that explicitly defines the equivalence class. We want to see *how well a LangLM builds clusters to approximate each equivalence class.* For this, we define a relation and different properties of a relation.

Definition 1 *A relation R over a set E is such that $R \subset E \times E$. We will note $x\mathcal{R}y$ to say $(x, y) \in R$.*

Definition 2 *A relation is reflexive if $x\mathcal{R}x$.*

Definition 3 *A relation is symmetric if $x\mathcal{R}y \Longrightarrow y\mathcal{R}x$.*

Definition 4 *A relation is transitive if $x\mathcal{R}y$ and $y\mathcal{R}z \Longrightarrow x\mathcal{R}z$.*

Definition 5 *An equivalence relation is reflexive, symmetric and transitive. The equivalence class of $x \in E$ according to an equivalence relation $\mathcal{R}$, denoted $\mathcal{C}_{\mathcal{R}}(x)$ or $\bar{x}$, is defined as:*

$$\mathcal{C}_{\mathcal{R}}(x) = \{y \in E; x\mathcal{R}y\}.$$

In our case, E is the set of all existing words of a language (lexemes and inflections) and $\mathcal{R}$ is a morphological relation. LM tries to generate an approximation of $\mathcal{C}_{\mathcal{R}}(x)$ for every word x. We focus on the relation $\mathcal{R}_{LM}$ (or $\mathcal{LM}$), the one that will produce the cluster $\mathcal{C}_{\mathcal{R}_{LM}}(x)$ (or $\mathcal{C}_{\mathcal{LM}}(x)$ if there is no confusion about the relation) for every word x. We can use the previous definition of an equivalence relation to build measures of a LangLM. Only two characterizations are useful; the symmetric relation and the transitive relation. The reflexive relation is evident since we automatically include a word into its set of variants.

The first measure is based on symmetry, measuring how much LangLM is symmetric. We classify each pair (x, y) into three categories:

- $x\mathcal{LM}y$ and $y\mathcal{LM}x$ then (x,y) is a symmetric pair,
- $x\mathcal{LM}y$ and not $y\mathcal{LM}x$ then (x,y) is a linked pair, and
- not $x\mathcal{LM}y$ and not $y\mathcal{LM}x$ then (x,y) is not a linked pair.

For example, consider the two following rules (assuming no other rule is tried):

```
.aeiouy l + l e r ->  &,&s,&est,&ed,&ing,&ings,ly,lely,&ness,
                      &nesses,&ment,&ments,&ful,&ers,&ered,
                      &ering,&erings,&erly,&erness,&ernesses,
                      &erments,&erless, &erful;
```

```
.aeiouy l l + ->  s,er,ers,est,ed,ing,ings,y,ness,nesses,ment,
                  ments,-less,ful;
```

From *caller*, the first rule will produce the existing words *call*, *calls*, *called*, *calling*, *callings*, and *callers*. From *call*, the second rule will produce the existing words *calls*, *caller*, *callers*, *called*, *calling*, and *callings*. Words *caller* and *call* are a linked and reflexive pair.

We measure the symmetry ($S_{\mathcal{LM}}$) of a LangLM by studying the linked pairs with the following formula:

$$S_{\mathcal{LM}} = \frac{\text{Number of symmetric and reflexive pairs}}{\text{Number of linked pairs}} \tag{6.1}$$

Ideally, $S = 1$ so we have a fully symmetric relation. We could achieve this result if $x\mathcal{LM}y$ and $y\mathcal{LM}x$ for every $(x, y) \in E \times E$, meaning we have only one set of morphological variants, or by having bigger sets of unrelated variants. However, this is not what we desire.

Our second measure will be based on transitivity, measuring how much LangLM creates bridges between different clusters. We define a footbridge word as a word y such as $x\mathcal{LM}y$ and $y\mathcal{LM}z$ but not $x\mathcal{LM}z$. For example *found* is a footbridge word connecting *find* and *founder*. For this, we measure the transitivity $T_{\mathcal{LM}}$ as:

$$T_{\mathcal{LM}} = \frac{\text{Number of footbridge words}}{\text{Number of words of E}} \tag{6.2}$$

Ideally, $T_{\mathcal{LM}} = 0$ so we do not connect two different clusters, therefore we have an equivalence class. Such a result is of course impossible to achieve (think of the natural footbridge word *found*). Again, one could lower $T_{\mathcal{LM}}$ by classifying a footbridge word into one of its clusters or by creating bigger but meaningless clusters (e.g. a cluster where *find* can get *founder*).

The symmetry and transitivity measures are based purely on morphological variants produced by LM. A third characteristic to measure the quality of each cluster constructed for a word x can be defined. The quality is measured by the related meaning of the cluster to the word x, not by related morphology. Morphological analysis is performed to add meaning to the query word. We can define a measure of related meaning $RM_{\mathcal{C}_{\mathcal{LM}}(x)}$ in a cluster as:

$$RM_{\mathcal{C}_{\mathcal{LM}}(x)} = \frac{\text{Number of related words to } x \text{ in } \mathcal{C}_{\mathcal{LM}}(x)}{\text{Number of words in } \mathcal{C}_{\mathcal{LM}}(x)} \tag{6.3}$$

For example, the French LangLM will produce *attention* (caution) from *attenter* (to make an attempt) by doing the common derivation of adding the *-tion* suffix to a verb. We are creating two words that are in the same cluster but with different meanings, since even their etymology is different. The other example is *found* that will produce *find* and *founder*. Since we are only interested in the related meaning of produced variations to *found*, it is not important that *find* and *founder* are not related in meaning.

The difficulty is to define how two words are related in meaning (if $x\mathcal{LM}y \iff x\mathcal{R}y$). The solution is to use a lexical semantic net that can identify related meanings and related derivations between x and y. WordNet [29] presents a partial solution to this problem.

6.1.3 LangLM measure results

As presented in the previous section, LangLM measures are supposed to get access to a corpus containing all existing words of a language. Fortunately, we can use a sampling approach to estimate $S_{\mathcal{LM}}$ and $T_{\mathcal{LM}}$. We used a reduced set of words E, consisting of all the terms that appear in the Hansard (French Hansard to test the French LangLM, English Hansard to test the English LangLM). The process is simple: we retrieve all terms (consisting only of letters) from the Hansard and compute the morphological variants for the term. If a variant is not in the collection, it is ignored. We also looked at the difference introduced by selecting only lower case terms (removing proper nouns).

For the symmetry measure, we keep a count of linked pairs (term $\longleftrightarrow$ variant). If a linked pair appears twice, it is a reflexive pair. We use these values to compute the symmetry estimate $\widetilde{S}_{\mathcal{LM}}$. The results for the English and French LangLM are given in Table 6.1.

Table 6.1: Reflexivity estimate for English and French LangLM.

LangLM	**Terms**	**Linked pairs**	**Reflexive pairs**	$\widetilde{\mathbf{S}}_{\mathcal{LM}}$
English	55,323	39,661	34,918	0.8804
English (Lower Case)	39,931	37,145	33,145	0.8923
French	76,031	287,968	259,516	0.9011
French (Lower Case)	59,395	282,694	255,528	0.9039

The transitivity measure is calculated by identifying all the footbridge words among the terms. For every term x appearing in the collection, we have access to the morphological variants $Y_x = y_x^1, \ldots, y_x^m$ produced by the term. Each morphological variant y_x^i is a term so we also have access to the morphological variants $Z_{y_x^i} = z^1_{y_x^i}, \ldots, z^n_{y_x^i}$ it produces. Therefore, a footbridge word is defined as $\{y \in Y_x \mid \exists z \in Z_y, z \notin Y_x\}$. Table 6.2 presents the results for the transitivity estimate $\widetilde{T}_{\mathcal{LM}}$.

Table 6.2: Transitivity estimate for English and French LangLM.

LangLM	**Terms**	**Footbridge words**	$\widetilde{\mathbf{T}}_{\mathcal{LM}}$
English	55,323	4,072	0.0736
English (Lower Case)	39,931	3,579	0.0896
French	76,031	5,132	0.0674
French (Lower Case)	59,395	4,426	0.0745

6.1.4 LangLM conclusion

English and French offer similar results, with symmetry scores being around 0.9 and transitivity scores below 0.09. The lower/upper case distinctions in both languages offer few differences between the measures. These results are near the optimal values.

These measures provide interesting overall results on the quality of LangLM. For example, only 10% of pairs formed by two words are not symmetric. True non-symmetric pairs are supposed to be even lower. Some spurious words exist in the Hansard resulting in adding a node and an edge that should not exist. Similar considerations hold for the transitivity measure. The most interesting measure, related meaning, could not be computed because of the complexity involved in its definition.

6.2 Improvement in information retrieval system

The usefulness of LM in information retrieval systems was tested with two approaches: (1) using differential recall on single term queries over the Hansard and (2) using TREC topics and relevance judgements to get recall and precision measures. On the first approach, we tested both French and English languages. In the second approach, we only tested English since no TREC collection set is available for French.

The following abbreviation will be used in the next sections: **LM** (Lightweight Morphology), **S** (Stemmer), **W** (Wildcard) and **EQ** (Exact Query).

6.2.1 Evaluating information retrieval systems

Evaluating an information retrieval system is based on the subjective notion of **relevance**, a judgement of a human to decide whether a document is relevant to the query or not. This notion of relevance has much to do with the previously defined notion of homonymy, polysemy and synonymy. Let's consider the word *tear* and two of its meanings: (1) tear from a teardrop, and (2) tear from something that is torn. One can consider that documents retrieved from meaning (1) and (2) are relevant since no one (computer or human) can differentiate the two meanings when only given *tear* as a query. On the other hand, if the user thought of the meaning (1) for the query *tear*, all the returned documents retrieved from meaning (2) will be classified as non-relevant. This simple example shows the difficulty of defining the notion of relevance of a document in response to a query.

Two major quantifying measures are used to evaluate an information retrieval system. The first one is a **recall** measure, judging the system's ability to retrieve relevant documents from the collection. The recall (R) is given by the formula [23]:

$$R = \frac{\text{Number of relevant documents returned}}{\text{Total number of relevant documents in the collection}} \tag{6.4}$$

Since a system could achieve 100% recall by only returning all the documents in the collection, we define a second measure, namely **precision**, which measures the accuracy of the system. This value indicates how many of the documents returned are relevant for a given query. The precision (P) is given by the formula [23]:

$$P = \frac{\text{Number of relevant documents returned}}{\text{Number of documents returned}} \tag{6.5}$$

We can see that precision and recall are related, and improving one of them often results in decreasing the other one (by making the system more or less selective). A third measure, called the **F-measure**, can be used to balance precision and recall using a parameter β. F-measure (F) is defined as follows [23]:

$$F = \frac{(\beta^2 + 1)\, PR}{\beta^2 P + R} \tag{6.6}$$

If $\beta = 1$, precision and recall are given the same weight. If $\beta > 1$ (resp. $\beta < 1$) precision (resp. recall) is favoured.

In information retrieval systems, it is more important to rank the documents than to make explicit relevance judgements. This is very understandable since users do not want all relevant documents to their query, but the most relevant ones. Therefore, many systems will retrieve more documents than the relevant ones (improving the recall by deliberately decreasing the precision) but compensate by providing a relevance ranking. The recall and precision measure can still be calculated by, for example, specifying cutoffs in the ranking. The **R-precision** for example is the precision calculated for a cutoff at r documents, where r is the total number of relevant documents for the query. Also, one can decide to evaluate the system by checking if the most relevant documents are ranked in the first documents.

Another difficulty is deciding how to compare two information retrieval systems. Is a system having a higher recall score better than a system having a higher precision score? The F-measure can help to combine recall and precision into a single measure. Another solution is to calculate a **differential recall** where the interest is to see how many documents the two systems retrieve in common and how many documents a system retrieved but the other system did not. In differential recall, we compare two systems A and B and calculate:

$$\begin{aligned} A \cap B \text{ (or } \cap_B^A \text{)} &= \text{Number of relevant documents returned by both } A \text{ and } B \\ A - B \text{ (or } \Delta_B^A) &= \text{Number of relevant documents returned by } A \text{ but not } B \\ B - A \text{ (or } \Delta_A^B) &= \text{Number of relevant documents returned by } B \text{ but not } A \end{aligned}$$

One can also complete the differential recall measure by using an irrelevance criterion instead of the previous relevance criterion to see how much ‘garbage’ each system returns.

This section presented some useful notions and measures to evaluate information retrieval systems and pointed out the difficulty of providing a rigorous testbed which needs a collection, a set of queries and their associated relevant documents. TREC (Text REtrieval Conference) [19, 35, 37] provides such a testbed, which consists of sets of documents accompanied by sets of queries and relevance judgements.

6.2.2 Differential recall

The goal of differential recall is to compare two different information retrieval systems A and B by checking which documents were retrieved by only one system. We focus on a comparison between two different query preprocessing techniques A and B. Our analysis is based on the 'difference' score (Δ_B^A and Δ_A^B) from differential recall measures. For the relevance criterion, if $\Delta_B^A > \Delta_A^B$, A is awarded 1 point (A is retrieving more relevant documents than B). For the irrelevance criterion, if $\Delta_B^A > \Delta_A^B$, B is awarded 1 point (B is retrieving less irrelevant documents than A).

We compared LM with a stemmer algorithm, an exact query and a wildcard query. The Porter stemmer [32] is used for English, while a stemmer created by Martin Porter [31] is used for French. For the wildcard query the allowed query operators are '?' (zero or one of any character) and '*' (zero or more of any character).

6.2.2.1 Differential recall preprocessing

For each language, we indexed our collection with an indexer provided by Sun Microsystems (the tokenization and indexing issues are not covered here). Two different word query sets were chosen to test the different techniques.

The first query set is composed of 100 randomly selected words from an Ispell [20] dictionary. If a selected word has no relevant or irrelevant hit in the collection with at least one of the techniques used, it is discarded and another word is randomly selected.

For the second query set, we classified the words appearing in the Hansard by frequency and selected the first 100 most frequent and meaningful words. We consider that a meaningful word is a word that is not common. Common words can be an article (e.g. *the*, *a*), a common verb (*be*, *go*) or any word that would not make any sense in a query as a single word (e.g. *however*, *first*).

As a result, each query is a single word. For each word, we had to create an equivalent wildcard query. We removed the ending of the word and replaced it by a '*' operator, so as to provide a kind of stem. For example, the word *slaying* will have *slai*[1] as the stem and *slay** as the wildcard query.

We also wanted to see the impact of processing the derivations for French. Two LangLMs were used for this purpose; one processing derivations for the suffixes -*age*, -*ance*, -*ment*, -*eur*, -*ion*, and -*able*, and one that does not. We denote the second LangLM as **LM′**. Both LangLM exception tables include morphological variants for inflection and derivation.

6.2.2.2 Differential recall results

For each query word, we classified the variants produced by the different techniques that appeared in the indexed collection as either relevant or irrelevant. Our relevance criterion is the correctness of the variant regarding the input word. We do not consider the correctness of the variant in the context of a collection. For example, from *tear*, we consider that *torn* is relevant, but *tearmann* is irrelevant. *Tear* is classified as relevant either in the context of a teardrop or of something that is torn. A sample of query words and their classification list is given in Appendix D. The complete

[1]The Porter stemmer has a rule to transform y in i, so *try* and *tries* has the same stem *tri*. *Slai* is the proper stem with the Porter stemmer.

differential recall measures for each query word are given in Appendix E.

The relevant win-lose scoring a - b between two techniques A and B is computed as follows:

- $a =$ Number of queries where $\Delta_B^A > \Delta_A^B$ (with relevant criterion)
- $b =$ Number of queries where $\Delta_B^A < \Delta_A^B$ (with relevant criterion)

The irrelevant win-lose scoring a - b between two techniques A and B is computed as follows:

- $a =$ Number of queries where $\Delta_B^A < \Delta_A^B$ (with irrelevant criterion)
- $b =$ Number of queries where $\Delta_B^A > \Delta_A^B$ (with irrelevant criterion)

Classification is used for calculating the differential recall measures. Results of the win-lose score are given in Table 6.3 for the randomly selected words and in Table 6.4 for the frequency selected words. Bold values indicates the 'winning' technique.

Table 6.3: Differential recall Win-Lose scores for 100 randomly selected words.

LangLM	Criterion	LM - S	LM′ - S	LM - W	LM - EQ	S - EQ
English	Relevant	**24** - **24**	N - A	8 - **44**	**82** - 0	**81** - 0
English	Irrelevant	**4** - 1	N - A	**39** - 0	0 - **1**	0 - **4**
French	Relevant	**35** - 13	25 - **36**	11 - **33**	**90** - 0	**93** - 0
French	Irrelevant	**5** - 2	5 - **1**	**26** - 1	0 - **2**	0 - **5**

Table 6.4: Differential recall Win-Lose scores for 100 frequency selected words.

LangLM	Criterion	LM - S	LM′ - S	LM - W	LM - EQ	S - EQ
English	Relevant	11 - **17**	N - A	8 - **33**	**71** - 0	**73** - 0
English	Irrelevant	**21** - 4	N - A	**63** - 2	0 - **9**	0 - **25**
French	Relevant	**17** - 7	12 - **13**	**23** - 14	**73** - 0	**70** - 0
French	Irrelevant	**20** - 5	**19** - 3	**50** - 7	0 - **11**	0 - **23**

6.2.2.3 Conclusion

First, we can see the usefulness of morphological analysis by comparing LM to EQ and to W queries. Compared to EQ, LM provides more relevant documents on 70 to 90 queries but introduces more irrelevant documents on 1 to 11 queries. Most of the time, performing morphological analysis on terms will retrieve more relevant documents. In a few cases, more irrelevant documents will be retrieved.

When comparing to W, we can see the advantage of performing morphological analysis: the user does not need to think about the query. Quickly stating an efficient wildcard query is not easy since we need to think of the kind of related words we will retrieve with the query. For example, we think of *pity* from *pitier* and create the query *pit**, but it will retrieve unrelated words like *pitbull*. If it is easy to often retrieve more relevant documents with a straightforward wildcard query (e.g. '*' will always get all the relevant terms), it is also easy to retrieve more irrelevant documents. Morphological analysis has a strong advantage since the user does not need to think about putting wildcard operators at the right place to ensure retrieval of many related and few unrelated words.

Compared to S, English LM is better, not always by retrieving more relevant documents on a query basis but by always retrieving fewer irrelevant documents.

Retrieving fewer irrelevant documents will result in less confusion for the users. We can see that for English, LM has a small advantage over S on the first query set by retrieving fewer irrelevant documents in 4 queries and more in only 1. On the second query set, S retrieves more relevant documents on 17 queries (11 for LM). However, LM does not retrieve so many irrelevant documents: in 21 queries, S retrieves more irrelevant documents and retrieves fewer irrelevant documents in only 4 queries. The Porter Stemmer for example, found the stem *intern* from *international* allowing variants such as *internalize* or *interned*.

French LM performs better than S in every case, retrieving more relevant documents and fewer irrelevant documents on a per query basis. French LangLM does not find verbs from nouns ending with a consonant (e.g. *accorder* from *accord* or *travailler* from *travail*) reducing its relevant retrieval efficiency.

Processing derivations in French is important. Ignoring the suffix derivations included in the full LangLM decreases the relevant documents retrieved, making LM′ less efficient than S for returning relevant documents. LM′ also has a minor impact on the irrelevant documents retrieved.

Concerning the test collection, we encountered 'spurious' terms that complicated the relevant/irrelevant classification. In both the English and the French Hansard, there are spelling errors (e.g. **financiére*), neologisms (e.g. *parliamentarianism*), French words in the English Hansard (e.g. *affairs/affaires*), English words in the French Hansard (e.g. *germes/germs*) and spacing errors between words (e.g. **comitteeadopted*). A solution would have been to look for every 'spurious' term in context inside the collection, so as to classify it as either relevant or irrelevant given a supposed meaning. These errors often appear in a document containing a relevant term, minimizing the range of the error. We instead used an approach of classifying a word as relevant if the meaning was correct, even if some spelling errors were present, and as irrelevant if the meaning was doubtful.

6.2.3 Recall and precision with TREC

TREC was used to get recall and precision scores for LM, stemming and exact query. In no way are single morphological analysis techniques aimed at competing with a full information retrieval system. Comparing the following results with TREC competitor results is pointless, and only a comparison between the three previously described techniques is made.

6.2.3.1 TREC queries preprocessing

We used the TREC collection volume 1 and 2 [35] (known as Disks 1 & 2 or TIPSTER Volumes 1 & 2) as the collection to index and topics 51 to 200 as the queries. The collection consists of 741,856 documents from 5 English-language sources. The documents contain approximately 185 millions words for disk 1 and 131 million words for disk 2 and include, for example, news article from the Wall Street Journal (1987–1992) and abstracts from the Department of Energy. The collection was indexed with the same indexer used for differential recall.

Two sets of queries were used. The first set (set 1) of queries was slightly preprocessed, by straightforwardly selecting the ‘title’ of each topic to be a query. We manually preprocessed the queries to remove all articles and meaningless words and included expanded acronyms (e.g. *U.S.* into *United States*). For example, query 51 is“Airbus Subsidies.”

The second set (set 2) aimed at providing an extensive list of words as a query. For each topic we selected the following entries: ‘domain’, ‘title’, ‘description’, ‘summary’ and ‘concepts’. We processed the text to remove common words with a stop list [34] and to remove punctuation symbols. Each resulting query is a list of unique and meaningful words. For example, query 51 expands to a list of 64 terms and represents a type of comprehensive query likely from a research librarian.

A sample of the queries used for this first and second set is provided in Appendix F.

6.2.3.2 TREC results

The results were obtained using the `trec_eval` program [36]. The measures are based on recall and precision, defined in Section 6.2.1. TREC measures use cutoffs to calculate different precision and recall scores, using a set of relevance judgements. We denote as L the number of relevant documents defined in the relevance judgements for a query. For each query and each technique (LM, S, EQ), a ranked list of the first $M =$ 1,000 documents retrieved by the IR system is considered. The ranking was provided by the Sun Labs Search Engine using our morphological variants and a ‘smart OR’ operator between words. The ‘smart OR’ performs the sum of weights of each term retrieved, instead of the usual ‘OR’ which takes the maximum weight of the term retrieved. Each ranked list is automatically compared to the relevance judgements to compute the following measures (a deeper explanation of these measures are given in Appendix F, Section F.3):

$P_I(r)$ — **Interpolated recall-precision at level r** The precision defined $P_I(r) = \max_{R(x)\geq r}(P(x))$. x is the number of relevant documents returned by the IR system.

$\overline{P}$ — **Average precision** The precision is calculated after each relevant document is retrieved and averaged over the number of relevant documents retrieved.

$P(d)$ — **Precision after d documents have been retrieved** The precision with a cutoff at d documents.

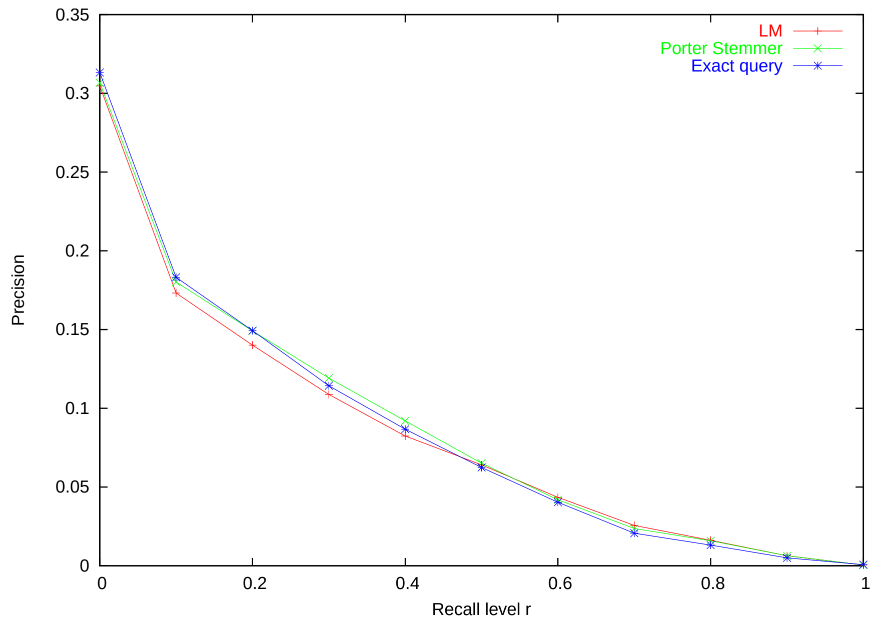
$P_R(d)$ — **R-precision** The precision with a cutoff at d documents, $d \leq \min(L, M)$.

The presented measures in Table 6.5 and Figures 6.1 and 6.2 are averaged over the number of queries (150 in our case), using the notation $\overline{X}$. For example, $\overline{\overline{P}}$ refers to the average over all 150 queries of the average precision measures (see Appendix F, Section F.3)

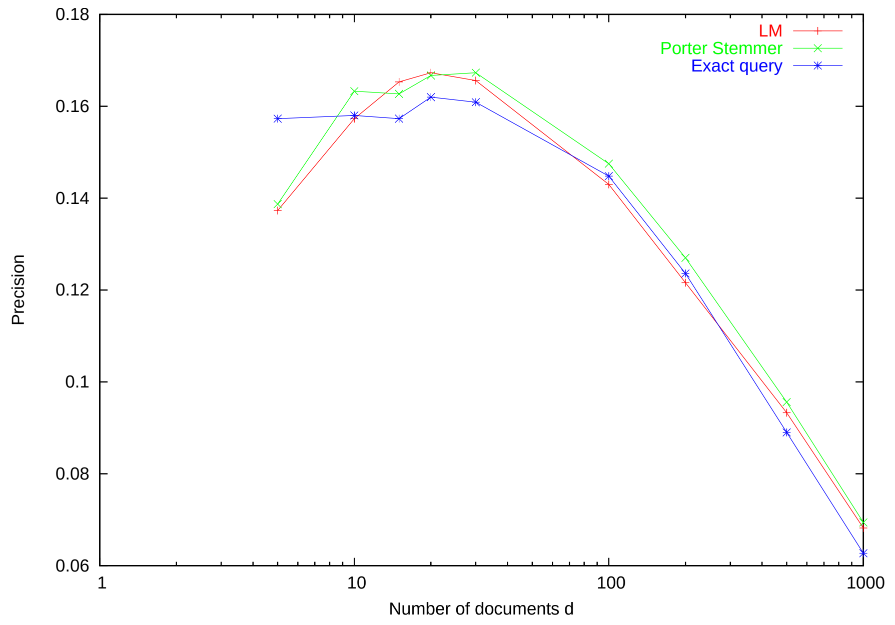
The relevance judgements for topics 51 to 200 indicates a total of 37,836 relevant documents over the 741,856 documents defining the collection. Table 6.5 presents an overview of the retrieval results for the two query sets and the three techniques used. Figures 6.1 and 6.2 are graphs presenting precision scores for the interpolated recall-precision average and for the cutoffs at different number of documents returned. The data to build these graphs is given in Appendix F.

6.2.3.3 Conclusion

Surprisingly, Lightweight Morphology and Stemming do not provide a significant advantage on information retrieval compared to an exact query. On the contrary,

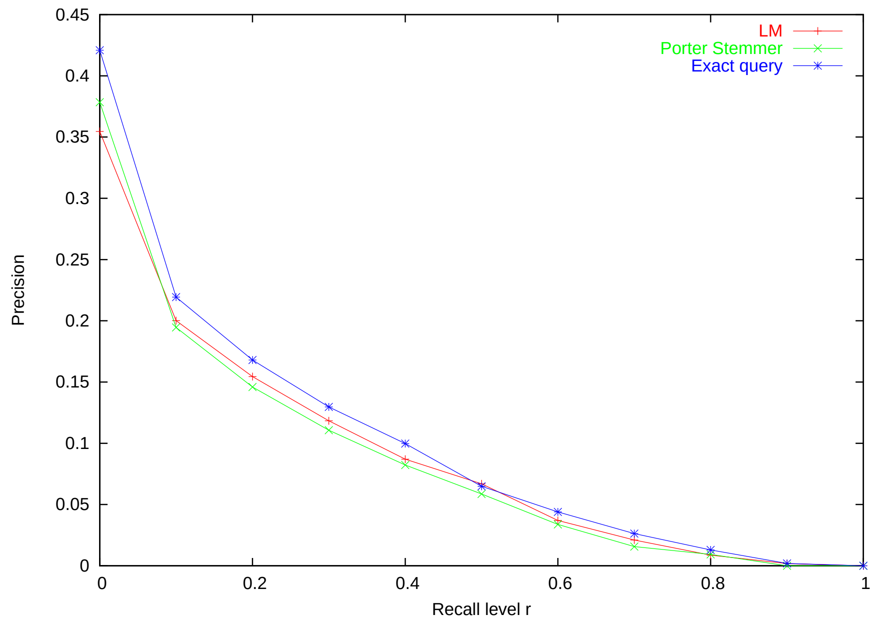


(a) Average interpolated recall-precision $\overline{P_I(r)}$.

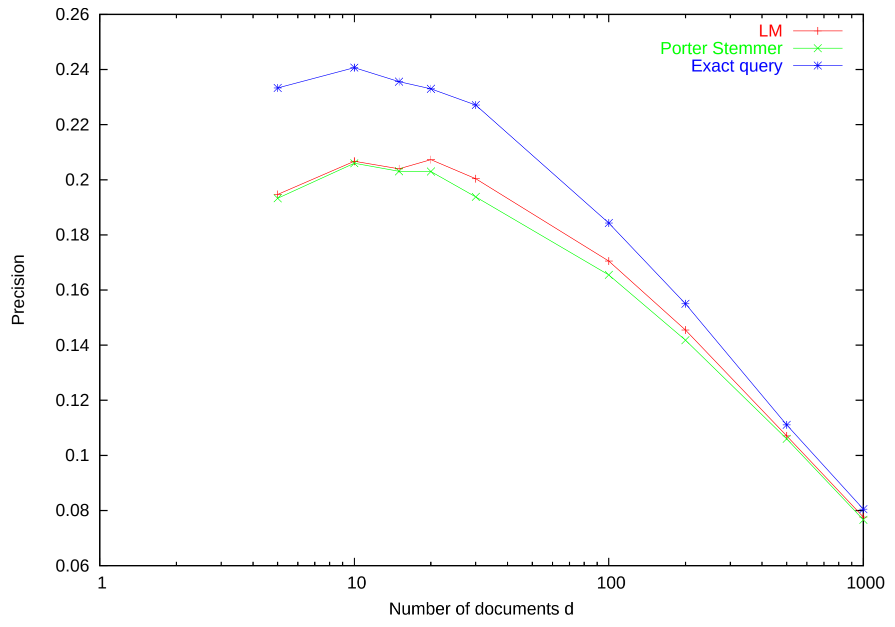


(b) Average precision after d documents retrieved $\overline{P(d)}$.

Figure 6.1: TREC average precision graphs for topics 51 to 200 with set 1 queries.



(a) Average interpolated recall-precision $\overline{P_I(r)}$.



(b) Average precision after d documents retrieved $\overline{P(d)}$.

Figure 6.2: TREC average precision graphs for topics 51 to 200 with set 2 queries.

Table 6.5: TREC global results for topics 51 to 200.

	Set 1			**Set 2**		
	LM	S	EQ	LM	S	EQ
Documents	147,476	147,399	147,162	150,000	150,000	150,000
Relevant documents	10,227	10,416	9,399	11,639	11,487	12,080
$\overline{\overline{P}}$	0.0743	0.0770	0.0757	0.0802	0.0761	0.0898
$\overline{P_R(x)}$	0.1269	0.1322	0.1291	0.1386	0.1354	0.1506

they seem to worsen (slightly) the results provided with an exact query. This is even more confusing as LM and S retrieve more relevant documents than EQ (around 1000) with only a few more total documents (around 300) for the first set of queries.

Adding terms to a query (essentially semantically different terms) as performed in the second query set improved the overall precision score of the methods. The results between Lightweight Morphology and Stemming are still similar, and they improved the results for the exact query by about 18%. The results obtained are quite paradoxical. Between the first set and the second set we added terms (4 to 50 times more terms and morphological variants of the terms) to the queries and the precision score was improved. Stemming and Lightweight Morphology automatically add terms (2 to 20 morphological variants per term) to the queries but the precision score does not increase as much as for the exact query (no increase for stemming, 8% increase for Lightweight Morphology).

Different causes can explain this behaviour:

1. TREC relevance judgements are based on a pooling technique. LM and S could have retrieved relevant documents that were not in the relevance judgement

set, thus counting possible relevant documents as irrelevant ones. This is even more possible since, quoting the overview of TREC-2 [19], ‘Pooling proved to be an effective method. There was little overlap among the 31 systems in their retrieved documents.’ The incompleteness of the relevance judgements was also noticed in [42].

2. The ranking algorithm does not take full advantage of LM and S. Thus, irrelevant documents can have a better ranking than relevant documents, affecting the precision score.

3. The queries are badly formulated. Since the same set of queries is used for the three approaches, this only explains why the results for the three approaches are so low compared to TREC participants’ results.

4. S and LM add ‘noise’ to the query. While this can be true for S when looking at the differential recall measures, it is less understandable for LM.

The lack of TREC data available for French is unfortunate since differential recall measures demonstrated promising results for information retrieval of French documents.

CHAPTER 7

Future research and improvements on natural language representation

7.1 Lightweight Morphology

Lightweight Morphology, especially its LiteMorph representation, is in the early stages of development. LangLMs for different languages (besides English, French, German and Spanish) have yet to be defined. Some problems with Lightweight Morphology and LiteMorph are present, and are described below.

7.1.1 Framework implementation improvement

As Figure 4.3 shows, the implementation is based on a Java implementation. The LiteMorph language is only hiding as best it can the underlying Java implementation from a user to ease the definition of a LangLM. This implies two problems:

1. The Lightweight Morphology engine needs to be compiled every time a LangLM is added or modified (with the two steps LM2Java to convert a LiteMorph into

Java and javac to compile the Java code into byte code);

2. Java rule sets rely heavily on Java and the design of LiteMorph is not optimal. Compilation errors can appear at two stages: LiteMorph to Java compilation and Lightweight Morphology engine compilation. Improving or adding LangLMs would be easier if second stage compilation errors are reported in the first stage.

These two problems can be redefined in terms of problems a linguist will encounter:

1. Even if a linguist does not have to write Java code, it is still their responsibility to integrate the resulting Java code into the Java LM, including the compilation of the LM code. Some knowledge of Java is still needed.
2. The relation between LiteMorph and Java is still strong because LiteMorph was built on top of the Java version of LM. This is particularly noticeable in the Java rule sets and the tight integration with Java that can be problematic. Again, some knowledge of Java is needed.

We use a formalism to write LangLM that is strictly[1] defined with a grammar and explanation of all operators. The framework should be adapted to LiteMorph.

Building a new architecture that dissociates the LangLM processor from LangLM specifications is needed. Ideally, the LangLM processor should read LiteMorph files as input. One could also think of a compiled format that would be read by the LangLM

[1]Strictly in the sense that LiteMorph is properly documented, not in the sense that the language will not improve (on the contrary)

processor. The LangLM processor should define an API that at minimum performs the following jobs:

1. Define the natural language to use (and therefore the LiteMorph file to read);
2. Define the query word to do morphological analysis on;
3. Provide a structure that returns the morphological variants.

The new architecture needs another language to define the Java rule set (which should be called user rule set). This topic is presented in the following section. The real challenge in this new architecture is how to integrate the user rule set. The user rule set should be defined in a language that is translated into the underlying programming language. The most elegant solution for this problem is to define a new language for the user rule set and provide an API in the LangLM processor that will match the features of the user rule set language. A compiler is used afterwards to convert the user rule set into the proper calls to the API.

7.1.2 LiteMorph Language improvement

LiteMorph language design relies on the Java Lightweight Morphology Framework. We present features that are not present or that should be modified in Lightweight Morphology.

7.1.2.1 User-defined rule set

The LiteMorph language can be further improved. First, the Java rule set is far from being perfect. The underlying implementation in Java makes it less than convenient. The use of the Table and List data structures is based on the knowledge of the underlying representation in Java. Errors that occur in the Java rule set are at best found in the LM2Java compilation step (lexical and syntactic errors) or at worst found in the Java compilation step (improper call to object methods or use of undefined variables) or at run-time (for Java run-time errors).

This should be avoided since debugging the last kind of error needs knowledge of Java and knowledge of how LiteMorph is converted into Java.

The whole set of Java features is used in user-defined rule sets but this is unnecessary. User-defined rule sets in LM represent morphological analysis that cannot be done with pattern-matching rules. The user-defined rule set should be defined using its own language with advanced string manipulation and proper data structures (like the predefined Table and List) with a simple syntax. One could use a scripting language like Groovy [17], a JRE compliant scripting language.

7.1.2.2 Exception table improvements

Consider a word x producing a morphological variant y from a pattern matching rule. It is possible that y appears in the exception table. Two cases can arise:

1. y should be valid and possibly retrieve all entries from the exception table. For example, *miser* should retrieve *mise* even if it is in the exception table.

2. y should not be valid since y is in the exception table but not x. In other words, there is no morphological relation between x and y. For example, *fraise* (strawberry) will produce *frais* (cool). However *frais* is in the exception table in the entry *frais, fraîche, fraîches* and should not be retrieved by *fraise*;

These behaviours need an exception table lookup every time a morphological variant is produced by pattern matching rules.

The opposite behaviour can also appear. If a word x is in the exception table but x is also a regular word, then it should be processed by the pattern matching rules to produce *Y1* = the set morphological variants from the exception table and *Y2* = the set of morphological variants from pattern matching rules. To quote the previous example, if *mise* is produced, it should also produce inflections from *miser*. The way the exception table works obliges one to encode a form that can be covered by pattern matching rules as an exception entry. This behaviour is not suitable since it can lead to a cascading of regular forms being hard coded in the exception table.

Some words (especially verbs) are not always completely irregular. Lightweight Morphology should provide a mechanism to call special rules so it can avoid writing all the entries. This can be done at compilation time, where the special rules are called in order to produce the whole entry. For example, suppose that we want to create entries for the French verbs ending with *-enir* in the exception table, like *tenir*, *venir* or *intervenir*. Writing those entries as ‘t(enirVerbParadigm)’, ‘v(enirVerbParadigm)’ and ‘interv(enirVerbParadigm)’, where ‘(enirVerbParadigm)’ contains the inflections for *-enir* verbs will (1) facilitate the use of the exception table, (2) avoid typos and (3) ease the recursive modification of the same kinds of entries.

7.1.2.3 Weighted Lightweight Morphology

When considering the process introduced by Lightweight Morphology, we do not differentiate the variants produced. A term will produce a set of variants that are supposed to be equivalent. Some variants, however, are more related to the initial term than other variants. For example, inflections are closely related to the term (having the word as singular or plural does not change the meaning of the word) whereas derivations are less related to the term (a derivation can introduce some variation in meaning).

To highlight this difference, we can extended Lightweight Morphology to introduce weights for each variant produced, where the weight represents how much the variant is related to the term it comes from. A simple weighting system is to provide a weight of 1 for inflections and 0.5 for derivations.

Different applications can be made of this weighting system. One can add the weight of the variants to compute the relevance score of a document. One can also decide not to consider morphological variants having a weight less than a certain threshold.

7.1.3 Improving LangLM authoring

Tools that help a user to define LangLMs have to be developed. Here we present two possible tools that are useful.

7.1.3.1 Improving the LiteMorph compiler

As we pointed out before, the Lightweight Morphology implementation framework should be improved, and these improvements should be reflected in the compiler. The aim is to catch all possible errors on the compilation step.

Some dynamic 'errors' should also be detected by the compiler. For example, when a rule set is never called, the compiler should raise a warning. A second example is to detect unreachable rules inside a rule set. This second type of error can have great consequences on the way Lightweight Morphology behaves. Putting the most general case before the most specific case will result in producing wrong variants since the specific case is never reached.

7.1.3.2 LiteMorph-friendly Integrated Development Environment

Writing a LangLM can be tedious because of all the information that is included in a single file. Going from Java to LiteMorph made this process less tedious since LiteMorph focuses on natural language morphology, not Java syntax. Further progress can be made by providing an Integrated Development Environment (IDE) for LiteMorph. The most basic feature should be syntax coloration. Other helpful features should include detecting multiple appearance of the same word in an exception table entry, sorting and searching efficiently the exception table and displaying available rule sets to call.

7.1.4 Improving French LangLM

The French LangLM can be further improved in many ways. First, one could be more precise on the definition of the pattern matching rules to be more selective and avoid spurious morphological variants. Also, one could look to add more derivations to the current LangLM. The improvement of French LangLM, beyond using resources for French (e.g. grammatical, spelling, etymology), should be done by constructing statistical data on French, such as 'how many words ending with *-ation* are derived from a verb and how many are not?'. This would help identifying useful derivations.

A litigious point on the current French LangLM is the inclusion of irregular verbs directly into the exception table. While it is easier to do that than defining proper rules, and while it is probably faster on run time performance, it becomes challenging to manage this list and add derivations. Spelling errors can also appear in this list[2]. Finally, including words in this exception list can also block creation of valid morphological variants (e.g. *mise* will produce the derivation of *mettre* but not of *miser*).

Finally, some derivations are not correctly processed. Derivational endings are not suffixes in some words and generate spurious variants, like creating *attenter* from *attention.* This behaviour is hard to avoid without a lexicon or should require more rules to prevent the creation of such variants. On the contrary, some derivations are not implemented, such as *-if/-ive* suffix. Some noun to verb conversion is not performed, when the ending of the word is a consonant, as for *accorder* from *accord* or *conseiller* from *conseil.*

[2]no human is perfect, including myself

7.1.5 Measuring and testing Lightweight Morphology

Chapter 6 introduced some ways to measure Lightweight Morphology and to test its usefulness. These measures are not complete and some results were not obtained.

An interesting question is the representational power of Lightweight Morphology. Can LM be used to create a stemmer? It seems that, by making heavy use of user-defined rule sets, this could be achieved. If this is true, LiteMorph and Lightweight Morphology can be used to perform what it was designed for (creating morphological variants), it can also be used to encode different kinds of stemmers.

7.1.6 Automatic generation of LangLM with machine learning and data mining

Building a LangLM can be complicated because of the amount of data that has to be encoded in it. The different resources are grammar, conjugation and etymology references and dictionaries. An investigation could be to see whether machine learning and data mining techniques can be applied so as to automatically generate a valid LangLM for a specific language. Clustering algorithms could be investigated for such a task.

7.2 Heavyweight Morphology

The original goal of this thesis was to provide a better representation for both Lightweight Morphology and Heavyweight Morphology. LM was more challenging than anticipated, and the representation for Heavyweight Morphology still remains to be done. The following section presents the initial architecture to provide this better representation.

7.2.1 Prolog and RuleML approach

To represent the rules and make them portable, we choose rule markup languages. The first one we focused on was XRML [26]. This choice appeared not suitable since XRML is essentially designed for e-business and little activity is taking place to further develop XRML.

We considered RuleML [6], an initiative to create a canonical Web language for rules using XML markup. The goal pursued is to have a general-purpose language to encode any kind of rule. RuleML is very similar to Prolog, and representing Heavyweight Morphology rules with Prolog or RuleML is equivalent. The following discussion supposes a Prolog approach. Figure 7.1 presents a possible conversion of the *-fish* rule presented in Figure 2.8 into a Prolog representation.

Figure 7.2 presents an approach to implement the lexical analyzer introduced in Figure 2.6 with a Prolog approach. Aggressive morphology rules are split into 'morphological processing' which will perform string manipulation and 'Prolog rules' that infer facts from the previous facts. This separation is recommended since Prolog has weak string manipulation. The morphological processing step could be written in Java to handle the string manipulation and create Prolog facts. The Prolog engine will then infer facts from the facts created by the morphological processing step and from the set of rules written in Prolog.

Different challenges are involved with this representation. First, the separation

```
/**********************************************
 * FACTS CREATED BY MORPHOLOGICAL PROCESSING STEP *
 **********************************************/

noun(fish).
noun(cat).
plausible_root(fish).
plausible_root(cat).
word(catfish).
prefix(catfish,cat).
suffix(catfish, fish).

/**********************************************
 * INFERENCE RULES FROM AGGRESSIVE MORPHOLOGY     *
 **********************************************/

root(X,Y) :- word(X), suffix(X,fish), prefix(X,Y).
nmsp(X) :- root(X,Y), noun(Y), plausible_root(Y).
nmsp(X) :- root(X,Y), adj(Y), plausible_root(Y).
false_root(X,Y) :- root(X,Y), nmsp(X).
kind_of(Z,X) :- suffix(X,Z), nmsp(X).
real_root(X,Y) :- suffix(X,Y), nmsp(X).
inflectional_paradigm(X, es) :- nmsp(X).
```

Figure 7.1: *-fish* Heavyweight Morphology rule converted to Prolog.

between morphological processing and inference processing is non-existent in the original Heavyweight Morphology rules in Lisp. Some difficulties will arise in providing this separation and in providing the interactions existing in the original Lisp rules. Second, the Lisp rules are launched in a specific order. Prolog engines do not provide a convenient way to control how the inferencing is performed. Careful design of the inference rules is necessary to respect the order of the rules. The order of rule firing can also be controlled by the syntactic facts created by morphological processing. Third, we need to access the Prolog database, to retrieve all inferred facts. Finally, we should be careful about the inferred facts created. Intermediate facts should not be created since there is no way to know the fact is intermediate and it will be considered

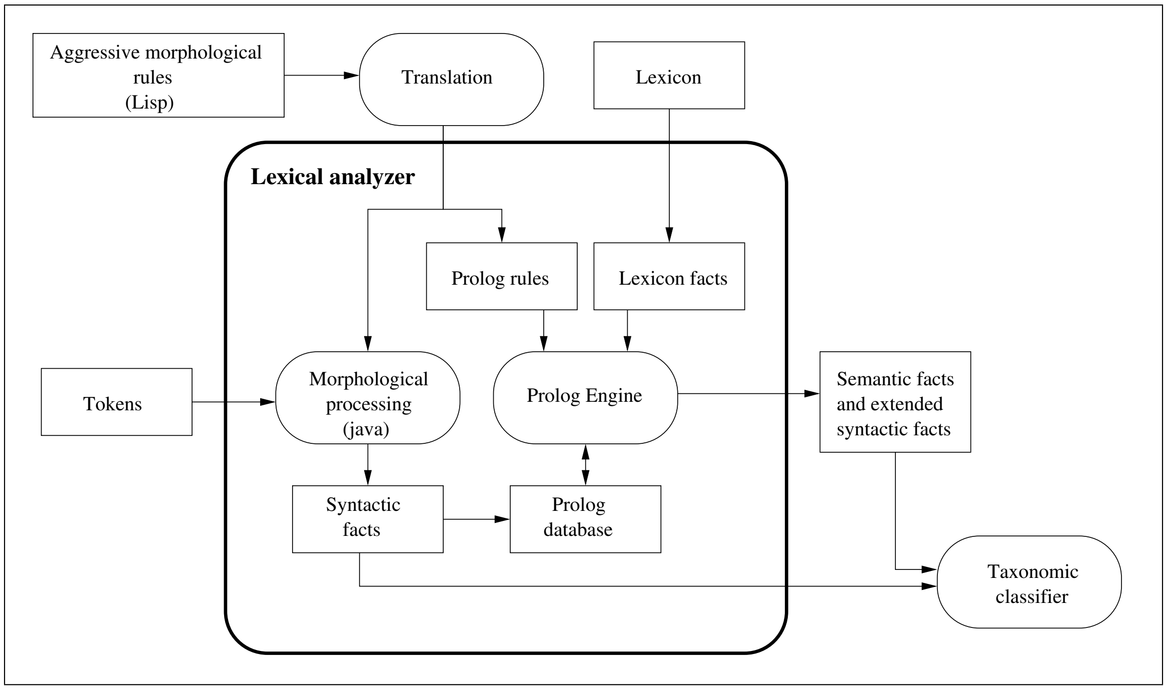


Figure 7.2: Prolog architecture.

as a syntactic or semantic fact.

CHAPTER 8

Conclusions

We introduced a new representation for natural language morphology with Lightweight Morphology and LiteMorph, a language to specify LangLMs. LiteMorph highly abstracts the definition of a LangLM by removing all unneeded features of the underlying computer language used to actually run the LM.

We tested LiteMorph by creating a French specification. Verbal, noun and adjectives inflections are provided. Six kind of derivations for the suffixes -*age*, -*ance*, -*ment*, -*eur*, -*ion*, and -*able* are integrated in the French LangLM, producing nouns and adjectives from verbs (and vice-versa). 526 rules, 41 rule sets and 16,842 exception table words are used for this specification. The pattern-matching rules, along with the exception table, are supposed to handle inflections for verbs, nouns and adjectives. On a set of 200 French words, 5% produced incorrect variants (e.g. *paye* from *pays*), and 7.5% did not produce one or more important morphological variant (e.g. *rapporter* is not produced from *rapport*).

Morphological analysis provided opposite results on improving information retrieval. The differential recall measures showed that 9 out of 10 queries retrieved more relevant document than an exact query. On average, Lightweight Morphology has 3.9 times more queries retrieving less irrelevant documents than stemming. The French version has, on average, 2.5 times more queries retrieving more relevant documents. TREC testing showed however that morphological analysis for English decreases the efficiency of information retrieval. The precision scores for stemming and Lightweight Morphology were equal to or lower than the precision scores for an exact query, the opposite of what we anticipated. The lack of available TREC collections for French prevented us from saying that Lightweight Morphology is not helpful for information retrieval.

Questions still remain. Where do the poor scores of morphological analysis on TREC testing come from? One can wonder if the incompleteness of the TREC relevance judgement is responsible for the results, if the ranking algorithm was not suitable for morphological analysis or if morphological analysis brings noise to a query.

Lightweight Morphology offers an alternative to stemming to expand query terms into morphological variants. It provides more control than a stemmer by offering a more intuitive way to construct rules or conditions, often resulting in better results. Lightweight Morphology can only go so far in morphological analysis since a lexicon has to be used at some point to avoid erroneous analysis. For example, avoiding analyzing *visage* (face) as *vis + age* and producing the inflections of *viser* (to aim) needs the assistance of a lexicon.

Chapter 7 briefly presented a possible representation of Heavyweight Morphology rules using Prolog. The specification and verification of such an approach still needs to be done. The major challenge faced is to separate the morphological processing from the inference rules while preserving the ordering in which the rules are fired.

References

[1] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 2nd edition, 2002.

[2] M. Arrivé. *Réformer l'ortographe ?* Presses Universitaires de France, Paris, 1993.

[3] K. Beesley and L. Karttunen. *Finite State Morphology.* CSLI Publications, 2003.

[4] *Bescherelle : La Conjugaison pour tous.* Hatier, Paris, 1997.

[5] E. Bled and O. Bled. *Bled : orthographe-grammaire.* Hachette, Paris, 2003.

[6] R. Boley, S. Tabet, and G. Wagner. Design rationale of RuleML: A markup language for semantic web rules. In *Proceedings of the First Semantic Web Working Symposium*, pages 191–202, 2001.

[7] Rapport du conseil supérieur de la langue française sur les rectifications de l'orthographe. *Documents officiels, Journal Officiel de la République Française*, 100, December 1990.

[8] M. Delacroix. *Les racines et la signification des mots français.* Eugène Belin, Paris, 4^{e} édition, 1886.

[9] J. Dendien. Dictionnaire de l'Académie Française, 8^{e} édition, version informatisée. `http://atilf.atilf.fr/academie.htm`.

[10] J. Dendien. Dictionnaire de l'Académie Française, 9^{e} édition, version informatisée. `http://atilf.atilf.fr/academie9.htm`.

[11] Académie Française. *Dictionnaire de l'Académie Française.* Fayard, Paris, 9^{e} édition, 1993-2004. Only tomes 1 and 2 available (A-mappemonde).

[12] Académie Française. *Dictionnaire de l'Académie Française.* 8^{e} édition, 1932-1935.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] Ulrich Germann. Aligned Hansards of the 36th parliament of Canada release 2001-1a (proceedings from September 25, 1997). Resources available at `http://www.isi.edu/natural-language/download/hansard/`.

[15] S. Green. Personal communication on Sun Labs Search Engine, October 14, 2004.

[16] M. Grevisse. *Le Bon Usage: Grammaire française avec des remarques sur la langue française d'aujourd'hui*. Duculot, Paris, 11^{e} édition, 1980.

[17] Groovy, project homepage. `http://groovy.codehaus.org/`.

[18] D. Harman. How effective is suffixing? *Journal of the American Society for Information Science*, 41(1):7–15, 1991.

[19] D. Harman. Overview of the Second Text REtrieval Conference (TREC-2). In *The Second Text REtrieval Conference (TREC-2)*, NIST Special Publication 500-215. National Institute of Standards and Technology, 1993.

[20] Ispell. `http://fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html`.

[21] JTB: Java Tree Builder, project homepage. `http://compilers.cs.ucla.edu/jtb`.

[22] JavaCC: Java Compiler Compiler, project homepage. `https://javacc.dev.java.net/`.

[23] D. Jurafsky and J. H. Martin. *Speech and language processing*. Prentice Hall, 2000.

[24] R. Krovetz. Viewing morphology as an inference process. In *Proceedings of the Sixteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Linguistic Analysis, pages 191–202, 1993.

[25] R. Krovetz. Homonymy and polysemy in information retrieval. In *Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics*, pages 72–79. Association for Computational Linguistics, 1997.

[26] J. K. Lee and M. M. Sohn. The eXtensible Rule Markup Language. *Communications of the ACM*, 46(5):59–64, 2003.

[27] J.R Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly, 2nd edition, 1995.

[28] J. B. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11(1-2):22–31, 1968.

[29] G. A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[30] M. Popovič and P. Willett. The effectiveness of stemming for natural-language access to Slovene textual data. *Journal of the American Society for Information Science*, 43(5):384–390, 1992.

[31] M. F. Porter. French stemmer. `http://snowball.tartarus.org/french/stemmer.html`.

[32] M. F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[33] M. Roussillon, B. G. Nickerson, and A. E. Maclachlan. Lightweight natural language morphology representation with Xerox finite state morphology. Technical Report TR04-166, University of New Brunswick, 2004.

[34] English list of stop words used in the SMART information retrieval system. `ftp://ftp.cs.cornell.edu/pub/smart/english.stop`.

[35] TREC collection volume 1 & 2 on Linguistic Data Consortium (references LDC93T3B and LDC93T3C). `http://www.ldc.upenn.edu/`.

[36] TREC retrieval runs evaluation program `trec_eval`. `ftp://ftp.cs.cornell.edu/pub/smart/trec2_eval.shar`.

[37] E. M. Voorhees and D. Harman, editors. *The Twelfth Text REtrieval Conference (TREC 2003)*, NIST Special Publication 500-255. National Institute of Standards and Technology, 2003.

[38] W. A. Woods. Conceptual indexing: A better way to organize knowledge. Technical Report TR-97-61, Sun Microsystems, 1997.

[39] W. A. Woods. Aggressive morphology for robust lexical coverage. Technical Report TR-99-82, Sun Microsystems, 1999.

[40] W. A. Woods. Documentation of Sun Java Enterprise System search engine. Internal Document, March 2004.

[41] W. A. Woods, L. A. Bookman, A. Houston, R. J. Kuhns, P. Martin, and S. Green. Linguistic knowledge can improve information retrieval. In *Proceedings of the Sixth Conference on Applied Natural Language Processing*, pages 262–267, April 2002.

[42] W. A. Woods, S. Green, P. Martin, and A. Houston. Halfway to question answering. In *Proceedings of the Ninth Text REtrieval Conference (TREC 9)*, pages 489–511, 2000.

[43] W. A. Woods and A. Houston. Documentation of Sun Java Enterprise System search engine. Internal Document, March 2004.

APPENDIX A

LiteMorph and modified Java — Grammars

The following sections present EBNF grammar for LiteMorph and a modified version of Java 1.4 used to parse the Java rule set from LiteMorph, based on a Java 1.4 grammar from Sriram Sankar used in JTB [21].

A.1 LiteMorph grammar

1	*LiteMorphLanguage*	::=	*Language* { *Options* }* { *DefLetterVariable* }* [*Exceptions*] [*JavaPart*] { { *RuleSetEnding* | *RuleSetNormal* } }* *RuleSetDefault* { { *RuleSet-Ending* | *RuleSetNormal* } }* **EOF**
2	*Language*	::=	**LANG =** *Id* **;**
3	*Options*	::=	**DEBUG ;** | **TRACE ;**
4	*DefLetterVariable*	::=	*LetterVariable* = *UnOrderedList* **;**
5	*Exceptions*	::=	**EXCEPTIONS {** { *ExceptionsList* }+ **}**
6	*ExceptionsList*	::=	*Word* { **,** *Word* }* **;**
7	*RuleSetNormal*	::=	**RULESET** *Id* **{** { *Rule* }+ **}**
8	*RuleSetEnding*	::=	**RULESET** *Id* **ENDING** *Id* **{** { *Rule* }+ **}**
9	*RuleSetDefault*	::=	**RULESET DEFAULT** *Id* **{** { *Rule* }+ **}**
10	*Rule*	::=	*PatternElements* **->** *ModificationPatterns* **;**

11	*PatternElements*	::=	[*FirstPattern*] { *LeftPatternEnd* }* [*LeftPatternInside*] { *LeftPatternEnd* }* [*LastPattern*]
12	*FirstPattern*	::=	[*BeginWord*] { *LeftAnchoredPattern* }* *BeginDelimiter*
		|	*BeginWord*
13	*BeginDelimiter*	::=	**-** ␣
14	*BeginWord*	::=	#
15	*LeftPatternEnd*	::=	[**.**] *LeftAnchoredPattern*
16	*LeftPatternInside*	::=	< { *LeftAnchoredPattern* }* >
17	*LeftAnchoredPattern*	::=	*OpSet Letters* | **&**
18	*OpSet*	::=	[|] [˜] [* | + | **?**]
19	*Letters*	::=	*UnOrderedList* | *OrUnorderedList* | *LetterVariable*
20	*LastPattern*	::=	*EndDelimiter* { *LeftAnchoredPattern* }* [*EndWord*] | *EndWord*
21	*EndDelimiter*	::=	+ ␣
22	*EndWord*	::=	#
23	*ModificationPatterns*	::=	*RightPattern* { **,** *RightPattern* }*
24	*RightPattern*	::=	*ReapplyPattern* [**&**] [*EndSubstitution* | *InsideSubstitution*]
		|	> *Mode UnOrderedList* > *UnOrderedList* / _ [*UnOrderedList*]
		|	**&** [*EndSubstitution* | *InsideSubstitution*]
		|	* [*EndSubstitution* | *InsideSubstitution*]
		|	{ *EndSubstitution* | *InsideSubstitution* }
25	*ReapplyPattern*	::=	[**TRY**] **(** [*Id*] **)**
26	*EndSubstitution*	::=	*UnOrderedList* _ [**-**] *UnOrderedList* | *UnOrderedList* _ | _ [**-**] *UnOrderedList* | [**-**] *UnOrderedList* | _
27	*InsideSubstitution*	::=	< [*UnOrderedList*] > [*ContextPattern*]
28	*ContextPattern*	::=	/ [*EndSubstitution*]
29	*Mode*	::=	* | < | >
30	*Id*	::=	*Letters1*
		|	*Letters2*
31	*UnOrderedList*	::=	*Letters1*
32	*OrUnorderedList*	::=	*Orunorderedlist*
33	*LetterVariable*	::=	*Lettervariable*
34	*Word*	::=	*Letters1* | *Letters2*
35	*JavaPart*	::=	{ *TableDef* | *ListDef* | *ListConcat* }* *JavaRuleSet*

36	*TableDef*	::=	**TABLE** Id **{** [[Id { , Id }*]] { Word { , Word }* ; }+ **}**
37	*ListDef*	::=	**LIST** Id **{** { Word ; }+ **}**
38	*ListConcat*	::=	**LIST** Id = Id { { + | + ␣ } Id }* ;
39	*JavaRuleSet*	::=	**JAVARULESET** { Modifiedjavacode }* **END-JAVARULESET**
40	*Orunorderedlist*	::=	Letter { | { Letter | Digit } }+
41	*Lettervariable*	::=	**$** Letter { Letter | Digit | **-** | _ }*
42	*Letters1*	::=	Letter { Letter | Digit }*
43	*Letters2*	::=	Letter { Letter | Digit | **-** }*
44	*Letter*	::=	{ **A** | ... | **Z** | **a** | ... | **z** }
45	*Digit*	::=	{ **0** | ... | **9** }
46	*Modifiedjavacode*	::=	

A.2 Modified Java grammar (based on Java 1.4 grammar)

1	*ModifiedJavaCode*	::=	{ ImportDeclaration }* { BlockStatement }* { ConditionBlockStatement }* { RuleBlockStatement }+ **EOF**
2	*ImportDeclaration*	::=	**import** Name [**.** *****] ;
3	*VariableDeclarator*	::=	VariableDeclaratorId [= VariableInitializer]
4	*VariableDeclaratorId*	::=	Identifier { [ ] }*
5	*VariableInitializer*	::=	ArrayInitializer
		|	Expression
6	*ArrayInitializer*	::=	**{** [VariableInitializer { , VariableInitializer }*] [,] **}**
7	*FormalParameter*	::=	[**final**] Type VariableDeclaratorId
8	*Type*	::=	PrimitiveType | Name { [ ] }*
9	*PrimitiveType*	::=	**boolean** | **char** | **byte** | **short** | **int** | **long** | **float** | **double** | **TABLE** | **LIST**
10	*ResultType*	::=	**void** | Type
11	*Name*	::=	Identifier { **.** Identifier }*
12	*NameList*	::=	Name { , Name }*
13	*Expression*	::=	Assignment
		|	ConditionalExpression

14	*Assignment*	::=	*PrimaryExpression AssignmentOperator Expression*
15	*AssignmentOperator*	::=	= | *= | /= | %= | += | -= | <<= | >>= | >>>= | &= | ˆ= | |=
16	*ConditionalExpression*	::=	*ConditionalOrExpression* [**?** *Expression* **:** *ConditionalExpression*]
17	*ConditionalOrExpression*	::=	*ConditionalAndExpression* { || *ConditionalAndExpression* }*
18	*ConditionalAndExpression*	::=	*InclusiveOrExpression* { && *InclusiveOrExpression* }*
19	*InclusiveOrExpression*	::=	*ExclusiveOrExpression* { | *ExclusiveOrExpression* }*
20	*ExclusiveOrExpression*	::=	*AndExpression* { ˆ *AndExpression* }*
21	*AndExpression*	::=	*EqualityExpression* { & *EqualityExpression* }*
22	*EqualityExpression*	::=	*InstanceOfExpression* { { == | != } *InstanceOfExpression* }*
23	*InstanceOfExpression*	::=	*RelationalExpression* [**instanceof** *Type*]
24	*RelationalExpression*	::=	*ShiftExpression* { { < | > | <= | >= } *ShiftExpression* }*
25	*ShiftExpression*	::=	*AdditiveExpression* { { << | >> | >>> } *AdditiveExpression* }*
26	*AdditiveExpression*	::=	*MultiplicativeExpression* { { + | - } *MultiplicativeExpression* }*
27	*MultiplicativeExpression*	::=	*UnaryExpression* { { * | / | % } *UnaryExpression* }*
28	*UnaryExpression*	::=	{ + | - } *UnaryExpression*
		|	*PreIncrementExpression*
		|	*PreDecrementExpression*
		|	*UnaryExpressionNotPlusMinus*
29	*PreIncrementExpression*	::=	++ *PrimaryExpression*
30	*PreDecrementExpression*	::=	- - *PrimaryExpression*
31	*UnaryExpressionNotPlusMinus*	::=	{ ˜ | **!** } *UnaryExpression*
		|	*CastExpression*
		|	*PostfixExpression*
32	*CastLookahead*	::=	(*PrimitiveType*
		|	(*Name* [ ]
		|	(*Name*) { ˜ | **!** | (| *Identifier* | **this** | **super** | **new** | *Literal* }
33	*PostfixExpression*	::=	*PrimaryExpression* [++ | - -]
34	*CastExpression*	::=	(*Type*) *UnaryExpression*

		|	(*Type*) *UnaryExpressionNotPlusMinus*
35	*PrimaryExpression*	::=	*PrimaryPrefix* { *PrimarySuffix* }*
		|	**trace** (*Expression*)
		|	**applyrule** (*Expression* , *Identifier* , *Expression*)
36	*PrimaryPrefix*	::=	*Literal*
		|	*Name*
		|	**this**
		|	(*Expression*)
		|	*AllocationExpression*
37	*PrimarySuffix*	::=	**. this**
		|	**. class**
		|	. *AllocationExpression*
		|	[*Expression*]
		|	. *Identifier*
		|	*Arguments*
38	*Literal*	::=	*Integer_literal*
		|	*Floating_point_literal*
		|	*Character_literal*
		|	*String_literal*
		|	*BooleanLiteral*
		|	*NullLiteral*
39	*BooleanLiteral*	::=	**true** | **false**
40	*NullLiteral*	::=	**null**
41	*Arguments*	::=	([*ArgumentList*])
42	*ArgumentList*	::=	*Expression* { , *Expression* }*
43	*AllocationExpression*	::=	**new** *PrimitiveType ArrayDimensions* [*ArrayInitializer*]
		|	**new** *Name* { *ArrayDimensions* [*ArrayInitializer*] | *Arguments* }
44	*ArrayDimensions*	::=	{ [*Expression*] }+ { [ ] }*
45	*ConditionBlockStatement*	::=	**CONDITION** (*Expression*) *Block*
46	*RuleBlockStatement*	::=	**RULE** (*Identifier*) *Block*
47	*Statement*	::=	*LabeledStatement*
		|	*Block*
		|	*EmptyStatement*
		|	*StatementExpression* ;
		|	*SwitchStatement*
		|	*IfStatement*
		|	*WhileStatement*
		|	*DoStatement*

		|	*ForStatement*
		|	*BreakStatement*
		|	*ContinueStatement*
		|	*ReturnStatement*
		|	*SynchronizedStatement*
		|	*TryStatement*
48	*LabeledStatement*	::=	*Identifier* **:** *Statement*
49	*Block*	::=	**{** { *BlockStatement* }* **}**
50	*BlockStatement*	::=	*LocalVariableDeclaration* **;**
		|	*Statement*
51	*LocalVariableDeclaration*	::=	[**final**] *Type VariableDeclarator* { **,** *VariableDeclarator* }*
52	*EmptyStatement*	::=	**;**
53	*StatementExpression*	::=	*PreIncrementExpression*
		|	*PreDecrementExpression*
		|	*Assignment*
		|	*PostfixExpression*
54	*SwitchStatement*	::=	**switch (** *Expression* **) {** { *SwitchLabel* { *BlockStatement* }* }* **}**
55	*SwitchLabel*	::=	**case** *Expression* **:**
		|	**default :**
56	*IfStatement*	::=	**if (** *Expression* **)** *Statement* [**else** *Statement*]
57	*WhileStatement*	::=	**while (** *Expression* **)** *Statement*
58	*DoStatement*	::=	**do** *Statement* **while (** *Expression* **) ;**
59	*ForStatement*	::=	**for (** [*ForInit*] **;** [*Expression*] **;** [*ForUpdate*] **)** *Statement*
60	*ForInit*	::=	*LocalVariableDeclaration*
		|	*StatementExpressionList*
61	*StatementExpressionList*	::=	*StatementExpression* { **,** *StatementExpression* }*
62	*ForUpdate*	::=	*StatementExpressionList*
63	*BreakStatement*	::=	**break** [*Identifier*] **;**
64	*ContinueStatement*	::=	**continue** [*Identifier*] **;**
65	*ReturnStatement*	::=	**return** [*Expression*] **;**
66	*SynchronizedStatement*	::=	**synchronized (** *Expression* **)** *Block*
67	*TryStatement*	::=	**try** *Block* { **catch (** *FormalParameter* **)** *Block* }* [**finally** *Block*]
68	*Integer_literal*	::=	*Decimal_literal* [**l** | **L**] | *Hex_literal* [**l** | **L**] | *Octal_literal* [**l** | **L**]
69	*Decimal_literal*	::=	{ **1** | ... | **9** } { **0** | ... | **9** }*

70	*Hex_literal*	::=	**0** { **x** | **X** } { **0** | ... | **9** | **a** | ... | **f** | **A** | ... | **F** }+
71	*Octal_literal*	::=	**0** { **0** | ... | **7** }*
72	*Floating_point_literal*	::=	{ **0** | ... | **9** }+ **.** { **0** | ... | **9** }* [*Exponent*] [**f** | **F** | **d** | **D**] | **.** { **0** | ... | **9** }+ [*Exponent*] [**f** | **F** | **d** | **D**] | { **0** | ... | **9** }+ *Exponent* [**f** | **F** | **d** | **D**] | { **0** | ... | **9** }+ [*Exponent*] { **f** | **F** | **d** | **D** }
73	*Exponent*	::=	{ **e** | **E** } [**+** | **-**] { **0** | ... | **9** }+
74	*Character_literal*	::=	**’** { { ˜{ **’** | \ | **\n** | **\r** } } | { \ { **n** | **t** | **b** | **r** | **f** | \\ | **’** | **\”** | **0** | ... | **7** [**0** | ... | **7**] | { **0** | ... | **3** } { **0** | ... | **7** } { **0** | ... | **7** } } } } **’**
75	*String_literal*	::=	**\”** { { ˜{ **\”** | \ | **\n** | **\r** } } | { \ { **n** | **t** | **b** | **r** | **f** | \\ | **’** | **\”** | **0** | ... | **7** [**0** | ... | **7**] | { **0** | ... | **3** } { **0** | ... | **7** } { **0** | ... | **7** } } } }* **\”**
76	*Identifier*	::=	*Letter* { *Letter* | *Digit* }*
77	*Letter*	::=	{ **$** | **A** | ... | **Z** | **_** | **a** | ... | **z** }
78	*Digit*	::=	{ **0** | ... | **9** }

APPENDIX B

LiteMorph — JavaCC source code

The following source code presents a JavaCC implementation of the LiteMorph language, based on the EBNF grammar from Appendix A. The JavaCC source code for the modified Java 1.4 is not included since easily reproducible from Sriram Sankar JavaCC source code used in JTB [21] and from the modified Java 1.4 grammar from Appendix A.

B.1 JavaCC source code for LiteMorph

```
/**
 * A grammar for LiteMorph. Support to include a variant of Java 1.4 included.
 * Author : Mikaël ROUSSILLON
 * Date : 30 august 2004
 *
 * Copyright University of New Brunswick
 */
options {
JAVA_UNICODE_ESCAPE = true;
STATIC = false;               // needed to add some methods
}

PARSER_BEGIN(LiteMorphParser)

package litemorphparser;

public class LiteMorphParser {

   int java_begin_line = -1;
   StringBuffer java_code = new StringBuffer("");
   public String getJavaCode() {
      return java_code.toString();
   }

   public int getJavaLineCode() {
      return java_begin_line;
   }
}

PARSER_END(LiteMorphParser)

SKIP : /* WHITE SPACE */
{
  "␣"
| "\t"
| "\n"
| "\r"
| "\f"
}

SPECIAL_TOKEN : /* COMMENTS */
{
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
| <FORMAL_COMMENT: "/**" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
| <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])* "*"))* "/">
}


TOKEN : /* SEPARATORS AND SOME OPERATORS */
{
   < PLUS: "+␣">
|  < MINUS: "-␣" >
|  < ARROW: "->" >
|  < LPAREN: "(" >
|  < RPAREN: ")" >
|  < MORPH : "!" >
|  < UNDERSCORE: "_" >
|  < COMMA: "," >
|  < HASH: "#" >
|  < LT: "<" >
|  < GT: ">" >
|  < VAR: "$" >
|  < SLASH: "/" >
|  < DQUOTE: "\"" >
|  < DOUBLE: "&" >
|  < STAR: "*" >
|  < COLON: ":" >
|  < LBRACE: "{" >
|  < RBRACE: "}" >
|  < ASSIGN: "=" >
|  < SEMICOLON: ";" >

}

TOKEN : /* RESERVED WORDS AND LITERALS */
{
   < TRY: "TRY" >
|  < LANG: "LANG" >
|  < DEBUG : "DEBUG" >
|  < TRACE : "TRACE" >
|  < RULESET: "RULESET" >
|  < JAVARULESET: "JAVARULESET" > : IN_JAVA_CODE
|  < DEF: "DEFAULT" >
|  < ENDING: "ENDING" >
|  < EXCEPTIONS: "EXCEPTIONS" >
|  < TABLE: "TABLE" >
|  < LIST: "LIST" >
}

<IN_JAVA_CODE> TOKEN :
{
   < ENDJAVARULESET: "ENDJAVARULESET" > : DEFAULT
}


TOKEN : /* PREFIXES */
{
   < OROP: "/" >
|  < NOTOP: "~" >
|  < ANYWHEREOP: "." >
|  < REGEXPOP1: "+" >
|  < REGEXPOP2: "?" >
}

TOKEN : /* ESCAPE PREFIXES, to force the previous prefixes to be real prefix
    operators, without space inbetween. They are not used. */
{
   < ESC_OROP: "/␣" >
|  < ESC_NOTOP: "~␣" >
|  < ESC_ANYWHEREOP: ".␣" >
|  < ESC_REGEXPOP2: "?␣" >

}

TOKEN : /* LETTERS, LITERALS */
{
   < DASH : "-" >
|  < ORUNORDEREDLIST: <LETTER> ("/" ( <LETTER> | <DIGIT> ) )+ >
|  < LETTERVARIABLE: <VAR> <LETTER> ( <LETTER> | <DIGIT> | "-"| "_" )* >
|  < LETTERS1: <LETTER> ( <LETTER> | <DIGIT> )* >
|  < LETTERS2: <LETTER> ( <LETTER> | <DIGIT> | "-" )* >
|  < #LETTER:
       [
        //"\u0024", ="$"
        "\u0041"-"\u005a",
        //"\u005f", ="_"
        "\u0061"-"\u007a",
        "\u00c0"-"\u00d6",
        "\u00d8"-"\u00f6",
        "\u00f8"-"\u00ff",
        "\u0100"-"\u1fff",
        "\u3040"-"\u318f",
        "\u3300"-"\u337f",
        "\u3400"-"\u3d2d",
        "\u4e00"-"\u9fff",
        "\uf900"-"\ufaff"
       ]
  >
|
  < #DIGIT:
       [
        "\u0030"-"\u0039",
        "\u0660"-"\u0669",
        "\u06f0"-"\u06f9",
        "\u0966"-"\u096f",
        "\u09e6"-"\u09ef",
        "\u0a66"-"\u0a6f",
        "\u0ae6"-"\u0aef",
        "\u0b66"-"\u0b6f",
        "\u0be7"-"\u0bef",
        "\u0c66"-"\u0c6f",
        "\u0ce6"-"\u0cef",
        "\u0d66"-"\u0d6f",
        "\u0e50"-"\u0e59",
        "\u0ed0"-"\u0ed9",
        "\u1040"-"\u1049"
       ]
  >
}

<IN_JAVA_CODE> TOKEN :
{
   < MODIFIEDJAVACODE: ~[] >
}


/**************************************
 * THE LITEMORPH GRAMMAR STARTS HERE  *
 **************************************/

void LiteMorphLanguage() : {}
{
   Language()
   ( Options() )*
   ( DefLetterVariable() )*
   [ Exceptions() ]
   [ JavaPart() ]
   ( LOOKAHEAD(3) ( LOOKAHEAD(3) RuleSetEnding() | RuleSetNormal() ) )*
   RuleSetDefault()
   ( LOOKAHEAD(3) ( LOOKAHEAD(3) RuleSetEnding() | RuleSetNormal() ) )*
   <EOF>
}

void Language() : {}
{
   "LANG" "=" Id() ";"
}

void Options() : {}
{
   (
   "DEBUG" ";"
|
   "TRACE" ";"
   )
}

void DefLetterVariable() : {}
{
   LetterVariable() "=" UnOrderedList() ";"
}

void Exceptions() : {}
{
   "EXCEPTIONS" "{" ( ExceptionsList() )+ "}"
}

void ExceptionsList() : {}
{
   Word() ( "," Word() )* ";"
}
```

```
void RuleSetNormal() : {}
{
   "RULESET" Id() "{" ( Rule() )+ "}"
}

void RuleSetEnding() : {}
{
   "RULESET" Id() "ENDING" Id() "{" ( Rule() )+ "}"
}

void RuleSetDefault() : {}
{
   "RULESET" "DEFAULT" Id() "{" ( Rule() )+ "}"
}

void Rule() : {}
{
   PatternElements() "->" ModificationPatterns() ";"
}

void PatternElements() : {}
{
   [ LOOKAHEAD(FirstPattern()) FirstPattern() ]
   ( LOOKAHEAD(LeftPatternEnd()) LeftPatternEnd() )*
   [ LOOKAHEAD(LeftPatternInside()) LeftPatternInside() ]
   ( LOOKAHEAD(LeftPatternEnd()) LeftPatternEnd() )*
   [ LastPattern() ]
}



void FirstPattern() : {}
{
    LOOKAHEAD([BeginWord()] ( LeftAnchoredPattern() )* BeginDelimiter())
    [BeginWord()] ( LeftAnchoredPattern() )* BeginDelimiter()
|
    BeginWord()
}

void BeginDelimiter() : {}
{
   "-␣"
}

void BeginWord() : {}
{
   "#"
}

void LeftPatternEnd() : {}
{
   [ <ANYWHEREOP> ] LeftAnchoredPattern()
}

void LeftPatternInside() : {}
{
   "<" ( LeftAnchoredPattern() )* ">"
}

void LeftAnchoredPattern() : {}
{
   (
   OpSet() Letters()
|
   "&"
   )
}

void OpSet() : {}
{
   [ "/" ] [ "~" ] [ "*" | "+" | "?" ]

}

void Letters() : {}
{
   (
   UnOrderedList()
|
   OrUnorderedList()
|
   LetterVariable()
   )
}


void LastPattern() : {}
{
   (
   EndDelimiter() ( LeftAnchoredPattern() )* [ EndWord() ]
|
   EndWord()
   )
}

void EndDelimiter() : {}
{
   "+␣"
}

void EndWord() : {}
{
   "#"
}

void ModificationPatterns() : {}
{
   RightPattern() ( "," RightPattern() )*
}

void RightPattern() : {}
{
   LOOKAHEAD( ReapplyPattern() [ "&" ] [ EndSubstitution()|InsideSubstitution() ] )
    ReapplyPattern() [ "&" ] [ EndSubstitution() | InsideSubstitution() ]
|
   LOOKAHEAD(4)
   ">" Mode() UnOrderedList() ">" UnOrderedList() "/" "_" [ UnOrderedList() ]
|
   "&" [ EndSubstitution() | InsideSubstitution() ]
|
   "*" [ EndSubstitution() | InsideSubstitution() ]
|
   ( EndSubstitution() | InsideSubstitution() )
}

void ReapplyPattern() : {}
{
   [ "TRY" ] "(" [ Id() ] ")"
}

void EndSubstitution() : {}
{
   (
   LOOKAHEAD(UnOrderedList() "_" [ "-" ] UnOrderedList())
   UnOrderedList() "_" [ "-" ] UnOrderedList()
|
   LOOKAHEAD(UnOrderedList() "_")
   UnOrderedList() "_"
|
   LOOKAHEAD("_" [ "-" ] UnOrderedList())
   "_" [ "-" ] UnOrderedList()
|
   [ "-" ] UnOrderedList()
|
   "_"
   )
}

void InsideSubstitution() : {}
{
   "<" [ UnOrderedList() ] ">" [ ContextPattern() ]
}

void ContextPattern() : {}
{

   "/" [ EndSubstitution() ]
}

void Mode() : {}
{
   (
   "*"
|
   "<"
|
   ">"
   )
}

void Id() : {}
{
   <LETTERS1>
|
   <LETTERS2>
}

void UnOrderedList() : {}
{
   <LETTERS1>
}

void OrUnorderedList() : {}
{
   <ORUNORDEREDLIST>
}

void LetterVariable() : {}
{
   <LETTERVARIABLE>
}
```

```
void Word() : {}
{
   (
   <LETTERS1>
|
   <LETTERS2>
   )
}

/*********************************************
 * THE JAVA PART LANGUAGE GRAMMAR STARTS HERE *
 *********************************************/


/*
 * Program structuring syntax follows.
 */
void JavaPart() : {}
{
   ( TableDef() | LOOKAHEAD(3) ListDef() | ListConcat())* JavaRuleSet()
}

void TableDef() : {}
{
   "TABLE" Id() "{" [ "[" Id() ( "," Id() )* "]" ] ( Word() ( "," Word() )* ";" )+ "}
       "
}

void ListDef() : {}
{
   "LIST" Id() "{" ( Word() ";" )+ "}"
}

void ListConcat() : {}
{
   "LIST" Id() "=" Id() ( ( "+" | "+␣" ) Id() )* ";"
}


void JavaRuleSet() : {}
{
  <JAVARULESET> { java_begin_line = n1.beginLine; }          // JTB complains about
       that but it is normal
  ( <MODIFIEDJAVACODE> { java_code.append(n4.image); } ) *
  <ENDJAVARULESET>
}
```

APPENDIX C

LiteMorph LangLM source code

The following sections present different LiteMorph LangLM source code. The French LangLM was originally written with LiteMorph. The other LangLMs were written in Java (by William Woods, helped by Ellen Hays for English and Ann Houston for Spanish and German) and translated afterwards in LiteMorph. Due to the length of each LangLM, only selected samples are included, trying to present as many features of LiteMorph as possible.

C.1 French LangLM

The overall French LangLM is 1,705 lines of code. It contains 16,842 words in the exception table, spread across 561 entries. The 526 rules composing the French LangLM are divided among 41 rule sets.

The following code presents samples from the complete French LangLM in Litemorph language: 4 entries from the exception table, and 2 rule sets, one being the default rule set (91 rules) and the second being the rule set creating infinitive forms for 1st group verbs (16 rules).

C.1.1 Head of French LangLM with samples of the exception table

```
/**
 * Author : Mikaël ROUSSILLON
 * Copyright (C) 2004 University of New Brunswick
 */

LANG = fr;

DEBUG;
TRACE;

$Consonant = bcçdfghjklmnpqrstvwxyz;
$Vowel = aàáâeèéêëiîïoôuùûüy;
$Letter = aàáâbcçdeèéêëfghiîïjklmnoôpqrstuùûüvwxyz;

EXCEPTIONS {
// A

à;
abattre, abats, abat, abattons, abattez, abattent, abattais, abattait, abattions,
    abattiez, abattaient, abattis, abattit, abattîmes, abattîtes, abattirent,
    abattrai, abattras, abattra, abattrons, abattrez, abattront, abattrais, abattrait
    , abattrions, abattriez, abattraient, abatte, abattes, abattisse, abattisses,
    abattît, abattissions, abattissiez, abattissent, abattant, abattante, abattants,
    abattantes, abattu, abattue, abattus, abattues, abattage, abattages, abatteur,
    abatteurs, abattement, abattements; // 60
absoudre, absous, absout, absolvons, absolvez, absolvent, absolvais, absolvait,
    absolvions, absolviez, absolvaient, absoudrai, absoudras, absoudra, absoudrons,
    absoudrez, absoudront, absoudrais, absoudrait, absoudrions, absoudriez,
    absoudraient, absolve, absolves, absolvant, absous, absout, absoute, absouts,
    absoutes, absolution, absolutions; // 78 + rectif. orth. déc 90
abstenir, abstiens, abstient, abstenons, abstenez, abstiennent, abstenais, abstenait
    , abstenions, absteniez, abstenaient, abstins, abstint, abstînmes, abstîntes,
    abstinrent, abstiendrai, abstiendras, abstiendras, abstiendrons, abstiendrez,
    abstiendront, abstiendrais, abstiendrait, abstiendrions, abstiendriez,
    abstiendraient, abstienne, abstiennes, abstenions, absteniez, abstiennent,
    abstinsse, abstinsses, abstînt, abstinssions, abstinssiez, abstinssent, abstenant
    , abstenante, abstenants, abstenantes, abstenu, abstenue, abstenus, abstenues,
    abstention, abstentions, abstinence, abstinences; //24
...
```

C.1.2 Default rule set for French LangLM

```
RULESET DEFAULT root {

   // test input as INFINITIF

   .$Letter + e r -> (RegErParadigm)_,                    // (verbes 1er groupe,
       chanter)
```

```
(NomAdjSingMasc)er;                                    // (boucher)
//.$Letter + i r e -> (RegIrParadigm)_,                // (verbes 2e groupe maudire)
//(RegErParadigm)ir,                                   // (verber en -irer, admire)
//(NomAdjSingMasc)ire;                                 // (accessoire)
.$Letter o i r # -> (NomAdjSingMasc)_;                 // (isoloir)
|apst + i r -> (RegIrParadigm)_,
(RegIrParadigm)ir,                                     // (éclair, soupir, désir,
    tir)
(NomAdjSingMasc)ir;
.$Letter + i r -> (RegIrParadigm)_,                    // (verbes 2e groupe, finir)
(NomAdjSingMasc)ir;

// test input as -é, -i, -u, -s, -t PARTICIPE PASSÉ (+ ADJECTIF form)
// -i ending is processed at the end

.$Vowel + é ?s -> (RegErParadigm)_,                    // (aimé)
(NomAdjSingMasc)é,                                     // (abandonné)
(NomAdjSingFem)é;                                      // (absurdité)
.$Vowel + é e ?s -> (RegErParadigm)_,                  // (aimée)
(NomAdjSingMasc)ée,                                    // (apogée)
(NomAdjSingFem)ée;                                     // (allée)


.$Vowel + i e ?s -> (RegIrParadigm)_,                  // (finie)
(RegErParadigm)i,                                      // (verbes en -ier)
(NomAdjSingMasc)ie,                                    // (brie)
(NomAdjSingFem)ie;                                     // (jolie)

// test input as -ant PARTICIPE PRÉSENT

.$Letter + i s s a n t ?s -> (RegIrParadigm)_,         // (finissant)
(NomAdjSingMasc)issant;                                // (épaississant)
.$Letter + i s s a n t e ?s -> (RegIrParadigm)_,       // (finissant)
(NomAdjSingFem)issante;                                // (épaississant)
.$Letter + a n t ?s -> (RegErParadigm)_,
(NomAdjSingMasc)ant;                                   // (chantant, adjudant)
.$Letter + a n t e ?s -> (RegErParadigm)_,
(NomAdjSingFem)ante;                                   // (chantant, adjudant)

// test input as -ion to create corresponding verb -- OPTIONAL (comment if not
    needed)

.$Letter s|t i o n # -> (IonNounToVerb)_,
(NomAdjSingMasc)_;                                     // (camion)
.$Letter s|t + i o n s # -> (RegErParadigm)_,          // (mangions)
(IonNounToVerb)ion,
(NomAdjSingMasc)ion;                                   // (camion)
.$Letter a i s o n # -> (IonNounToVerb)_,
(NomAdjSingMasc)_;
.$Letter a i s + o n s # -> (RegErParadigm)_,          // (apaisons)
(IonNounToVerb)on,                                     // (combinaison)
(NomAdjSingMasc)on;

// test input as -eur/euse/rice/resse to create corresponding verb -- OPTIONAL (
    comment if not needed)

.$Letter + e u r ?s # -> (EurNounToVerb)eur,
(NomAdjSingMasc)eur;
.$Letter + e u s e ?s # -> (EurNounToVerb)euse,
(NomAdjSingFem)euse;
.$Letter + r i c e ?s # -> (EurNounToVerb)rice,
(NomAdjSingFem)rice;
.$Letter e|o + r e s s e ?s # -> (EurNounToVerb)resse,
(NomAdjSingFem)resse;
```

```
// test input as -able to create corresponding verb -- OPTIONAL (comment if not
    needed)

.$Letter + a b l e ?s -> (AbleAdjToVerb)able,
(NomAdjSingMasc)able;

// test input as -age to create corresponding verb -- OPTIONAL (comment if not
    needed)

.$Letter + a g e ?s -> (AgeNounToVerb)age,           // (abordage)
(RegErParadigm)ag,                                   // (amménager)
(NomAdjSingMasc)age;                                 // (abordage)

// test input as -ance to create corresponding verb -- OPTIONAL (comment if not
    needed)
.$Letter + a n c e ?s -> (AnceNounToVerb)ance,       // (assurance)
(RegErParadigm)anc,                                  // (balancer)
(NomAdjSingMasc)ance;                                // (assurance)

// test FUTURE -er,-ir, other forms


?r e n v e r + r a i -> (RegErParadigm)er;           // (enverrai)
?r e n v e r + r a ?s -> (RegErParadigm)er;
?r e n v e r + r o n s|t -> (RegErParadigm)er;
?r e n v e r + r e z -> (RegErParadigm)er;

.$Letter + e r a i -> (RegErParadigm)_,
(NomAdjSingMasc)erai;                                // (minerai)
.$Letter + e r a ?s -> (RegErParadigm)_,
(NomAdjSingMasc)era;                                // (chisteras)
.$Letter + e r o n s -> (RegErParadigm)_,
(NomAdjPlurMasc)erons;                               // (bucherons)
.$Letter + e r e z -> (RegErParadigm)_;              // (jerez ???)
.$Letter + e r o n t -> (RegErParadigm)_;

.$Letter + i r a i -> (RegIrParadigm)_;
.$Letter + i r a ?s -> (RegIrParadigm)_;
.$Letter + i r o n s -> (RegIrParadigm)_,
(RegErParadigm)ir,                                   // (admirons)
(NomAdjPlurMasc)irons;                               // (avirons)
.$Letter + i r e z -> (RegIrParadigm)_,
(RegErParadigm)ir;                                   // (admirez)
.$Letter + i r o n t -> (RegIrParadigm)_;


// test  CONDITIONNEL -er,-ir, other forms


?r e n v e r + r a i s|t -> (RegErParadigm)_;        // (enverrais)
?r e n v e r + r i o n s -> (RegErParadigm)_;
?r e n v e r + r i e z -> (RegErParadigm)_;
?r e n v e r + r a i e n t -> (RegErParadigm)_;

.$Letter + e r a i s -> (RegErParadigm)_
,(NomAdjPlurMasc)erais; // (minerais)
.$Letter + e r a i t -> (RegErParadigm)_;
.$Letter + e r i o n s -> (RegErParadigm)_;
.$Letter + e r i e z -> (RegErParadigm)_;
.$Letter + e r a i e n t -> (RegErParadigm)_;

.$Letter + i r a i s|t -> (RegIrParadigm)_,
(RegErParadigm)ir;                                     // (admirais)
.$Letter + i r i o n s -> (RegIrParadigm)_,
(RegErParadigm)ir;
.$Letter + i r i e z -> (RegIrParadigm)_,
(RegErParadigm)ir;
.$Letter + i r a i e n t -> (RegIrParadigm)_,
(RegErParadigm)ir;


// test INDICATIF PRÉSENT et IMPARFAIT, PASSÉ SIMPLE, SUBJONCTIF PRÉSENT et
    IMPARFAIT -er,-ir , other forms


.$Letter + i s s a i s|t -> (RegIrParadigm)_,
(RegErParadigm)iss;                                    // (hisser , verbes en -isser)
.$Letter + a i s -> (RegErParadigm)_,                  // (aimais)
(NomAdjPlurMasc)ais;                                   // (bais)
.$Letter + a i t -> (RegErParadigm)_,
(NomAdjSingMasc)ait;                                   // (abstrait)
.$Letter + o i s -> (NomAdjSingMasc)ois,               // (anchois, bourgeois)
(NomAdjPlurMasc)ois;                                   // (tournois)
.$Letter + i s -> (RegIrParadigm)_,                    // (finis)
(NomAdjPlurMasc)is,                                    // (abris)
(NomAdjSingMasc)is;
.$Letter + i s s e ?s -> (RegIrParadigm)_,             // (finisse)
(RegErParadigm)iss;                                    // (hisser , verbes en -isser)
.$Letter + i t -> (RegIrParadigm)_,                    // (finit)
(NomAdjSingMasc)it,                                    // (déficit)
(NomAdjSingFem)it;                                     // (nuit)
.$Letter + î t -> (RegIrParadigm)_;                    // (finît)
.$Letter + i s s i o n s -> (RegIrParadigm)_,          // (finissions)
(RegErParadigm)iss,                                    // (hisser , verbes en -isser)
(NomAdjPlurFem)issions;                                // (admissions)
.$Letter + i s s o n s -> (RegIrParadigm)_,            // (finissons)
(RegErParadigm)iss,                                    // (hisser , verbes en -isser)
(NomAdjPlurMasc)issons,                                // (buissons)
(NomAdjPlurFem)issons;                                 // (boissons)
.$Letter + i s s ?i e z -> (RegIrParadigm)_,
(RegErParadigm)iss;                                    // (hisser , verbes en -isser)
.$Letter + i s s e n t -> (RegIrParadigm)_,
(RegErParadigm)iss;                                    // (hisser , verbes en -isser)
.$Letter + î m e s -> (RegIrParadigm)_,
(NomAdjPlurMasc)îmes,                                  // (abîmes)
(NomAdjPlurFem)îmes;                                   // (dîmes)
.$Letter + î t e s -> (RegIrParadigm)_,
(NomAdjPlurMasc)îtes,                                  // (gîtes)
(NomAdjPlurFem)îtes; // (faîtes)
.$Letter + i r e n t -> (RegIrParadigm)_,
(RegErParadigm)ir;                                     // (admirent)
.$Letter + a s s e s -> (RegErParadigm)_,              // (aimasses)
(NomAdjPlurFem)asses;                                  // (blondasses -> blonde)
.$Letter + a s s e  -> (RegErParadigm)_,               // (aimasse)
(NomAdjSingFem)asse;                                   // (blondasse -> blonde)
.$Letter + â t -> (RegErParadigm)_,
(NomAdjSingMasc)ât;                                    // (mât)
.$Letter + a s s i o n s -> (RegErParadigm)_,          // (aimasse)
(NomAdjPlurFem)assions;                                // (passions)
.$Letter + i o n s -> (RegErParadigm)_,
(NomAdjSingMasc)ion,                                   // (camions)
(NomAdjSingFem)ion;                                    // (action)
.$Letter + o n s -> (RegErParadigm)_,
(NomAdjPlurMasc)ons,                                   // (abandons)
(NomAdjPlurFem)ons;                                    // (façon)
.$Letter + â m e s -> (RegErParadigm)_,                // (aimâmes)
```

```
    (NomAdjSingMasc)âme;                                     // (âme)
    .$Letter + a s s i e z -> (RegErParadigm)_;              // (aimasse)
    .$Letter + i e z -> (RegErParadigm)_;
    .$Letter + e z -> (RegErParadigm)_,
    (NomAdjSingMasc)ez;
    .$Letter + â t e s -> (RegErParadigm)_,                  // (aimâtes)
    (NomAdjPlurMasc)âtes;                                    // (pâte)
    .$Letter + a s s e n t -> (RegErParadigm)_;              // (aimasse)
    .$Letter + a i e n t -> (RegErParadigm)_,                // (amaient)
    (RegErParadigm)ai; // (verbes en -ayer )
    .$Letter + è r e n t -> (RegErParadigm)_,                // (aimèrent)
    (RegErParadigm)èr;                                       // (adhèrent)
    .$Letter + i s s e m e n t ?s -> (RegIrParadigm)_,       // (assainissement)
    (RegErParadigm)iss,                                      // (crisser, verbes en -isser
        )
    (NomAdjSingMasc)ement;                                   // (agissement)
    .$Letter + e m e n t ?s -> (RegErParadigm)_,             // (dépouillement)
    (NomAdjSingMasc)ement;
    .$Letter + é m e n t ?s -> (RegErParadigm)_,             // (aveuglément)
    (RegErParadigm)é,                                        // (supplément)
    (NomAdjSingMasc)ément;
    .$Letter + e n t -> (RegErParadigm)_,
    (NomAdjSingMasc)ent;                                     // (vent)
    .$Letter + a i -> (RegErParadigm)_,                      // (aimai)
    (NomAdjSingMasc)ai,
    (NomAdjSingFem)ai;
    .$Letter + a s -> (RegErParadigm)_,                      // (aimas)
    (NomAdjSingMasc)as,
    (NomAdjPlurMasc)as,
    (NomAdjPlurFem)as;
    .$Letter + a -> (RegErParadigm)_,                        // (aima)
    (NomAdjSingMasc)a,
    (NomAdjSingFem)a;
    .$Letter + e s -> (RegErParadigm)_,                      // (aimes)
    (NomAdjSingMasc)es,
    (NomAdjPlurMasc)es,
    (NomAdjPlurFem)es;
    .$Letter + e -> (RegErParadigm)_,                        // (aime)
    (NomAdjSingMasc)e,
    (NomAdjSingFem)e;

    //
    .$Letter o i # -> (NomAdjSingMasc)_;                     // (tournoi)
    .$Consonant + i -> (RegIrParadigm)_,                     // (fini)
    (NomAdjSingMasc)i,                                       // (abri)
    (NomAdjSingFem)i;                                        // (fourmi)

    // test input as -in to create corresponding verb -- OPTIONAL (comment if not
        needed)

    .$Consonant i n ?s # -> (RegErParadigm)_,                    // (assasin)
    (NomAdjSingMasc)_;


    .$Letter # -> (NomAdjSingMasc)_,(NomAdjSingFem)_,(NomAdjPlurMasc)_,(NomAdjPlurFem)
        _;
}
```

C.1.3 Sample rule set for French LangLM

```
RULESET ErInfin {
   //generate infinitif forms for -er verbs, from verb stem
   //Assume input has stripped infinitive -er

   .$Letter v + e r -> oyer; // (envoyer, renvoyer from futur simple)
   .$Letter <è> ?$Consonant + c|ç -> <é>/_cer,<e>/_cer; // (rapiécer,dépecer)
   .$Letter <é> ?$Consonant + c|ç -> <é>/_cer; // (rapiécer)
   .$Letter <e> ?$Consonant + c|ç -> <e>/_cer; // (dépecer)
   .$Letter + e t ?t -> etter, eter; // (jetter, acheter, + rectifications
       orthographique 6 déc. 1990)
   .$Letter + è t ?t -> etter, eter, éter; // (jetter, acheter, refléter +
       rectifications orthographique 6 déc. 1990)
   .$Letter + e l ?l -> eller,eler; // (appeler, modeler + rectifications
       orthographique 6 déc. 1990)
   .$Letter + è l ?l -> eller,eler, éler; // (appeler, modeler, corréler +
       rectifications orthographique 6 déc. 1990)
   .$Letter <é|è> g + ?e -> <é>/_er; // (assiéger)
   .$Letter <è> $Consonant ?$Consonant # -> <é>/_er,<e>/_er; // (céder,peser, verbes
       -é(.)er et -e(.)er ...)
   .$Letter <é> $Consonant ?$Consonant # -> <é>/_er; // (céder, verbes -é(.)er)
   .$Letter <e> $Consonant ?$Consonant # -> <e>/_er; // (peser, verbes -e(.)er)
   .$Letter a|o|u + i|y -> yer; // (payer, broyer)
   .$Letter + c|ç -> cer; // (placer)
   .$Letter g + ?e -> er; // (manger)
   .$Letter # -> er; // (aimer, apprécier, créer, default form)
}
```

C.2 Spanish LangLM

The Spanish LangLM is 1,608 lines long and consists of an exception table of 127 entries containing a total of 377 words. Sixty-one rules are split among 59 rule sets. A Java rule set is also used, which is presented in the following sample code. A sample code presenting the call to this Java rule set from the default rule set, made with the *TRY* operator to handle future forms of irregular verbs is also included.

C.2.1 Java rule set for Spanish LangLM

```
/**
 * Authors : W. A. Woods and Ann Houston
 * Copyright (C) 1995-2004 Sun Microsystems, Inc.
 *
 * Translation to LiteMorph : Mikaël ROUSSILLON
 * Copyright (C) 2004 University of New Brunswick
 */

LIST prefixes {
        ad; ab; ante; a; circun; com; contra; con; des; de; dis; entre; en; extra; ex
            ; im; inter; intra; in; menos; minus; ob; o; pre; pro; post; pos; re;
            super; sub; su; sobre; trans; tras;

}

TABLE paradigms {

        [infin, prog, part, pres, pret]

        abrir, abriendo, abierto, abro, abrí;
        andar, andando, andado, ando, anduve;
        almorzar, almorzando, almorzado, almuerzo, almorcé;
        argüir, arguyendo, argüido, arguyo, argüi;
        asir, asiendo, asido, asgo, así;
        avergonzar, avergonzando, avergonzado, avergüenzo, avergoncé;
        bendecir, bendeciendo, bendecido, bendigo, bendije;
        buscar, buscando, buscado, busco, busqué;
        caber, cabiendo, cabido, quepo, cupe;
        caer, cayendo, caído, caigo, caí;
        cocer, cociendo, cocido, cuezo, cocí;
        coger, cogiendo, cogido, cojo, cogí;
        conducir, conduciendo, conducido, conduzco, conduje;
        conocer, conociendo, conocido, conozco, conocí;
        contar, contando, contado, cuento, conté;
        creer, creyendo, creído, creo, creí;
        cubrir, cubriendo, cubierto, cubro, cubrí;
        decir, diciendo, dicho, digo, dije;
        distinguir, distinguiendo, distinguido, distingo, distinguí;
        dormir, durmiendo, dormido, duermo, dormí;
        empezar, empezando, empezado, empiezo, empecé;
        errar, errando, errado, yerro, erré;
        esparcir, esparciendo, esparcido, esparzo, esparcí;
        estar, estando, estado, estoy, estuve;
        freir, friendo, frito, frío, freí;
        haber, habiendo, habido, he, hube;
        hacer, haciendo, hecho, hago, hice;
        huir, huyendo, huido, huyo, huí;
        jugar, jugando, jugado, juego, jugué;
        leer, leyendo, leído, leo, leí;
        llover, lloviendo, llovido, llueve, lloví;  // should probably use 3rd person
             for weather verbs
        maldecir, maldeciendo, maldecido, maldigo, maldije;
        morir, muriendo, muerto, muero, morí;
        nevar, nevando, nevado, nievo, neví;
        oír, oyendo, oído, oigo, oí;
        oler, oliendo, olido, huelo, olí;
        pensar, pensando, pensado, pienso, pensé;
        pedir, pidiendo, pedido, pido, pedí;
        poder, pudiendo, podido, puedo, pude;
        poner, poniendo, puesto, pongo, puse;
        proveer, proveyendo, provisto, proveo, proveí;
        querer, queriendo, querido, quiero, quise;
        reír, riendo, reído, río, reí;
        solver, solviendo, suelto, suelvo, solví;  // a root for resolver, absolver,
            disolver
        romper, rompiendo, roto, rompo, rompí;
        saber, sabiendo, sabido, sé, supe;
        salir, saliendo, salido, salgo, salí;
        seguir, siguiendo, seguido, sigo, seguí;
        sentir, sintiendo, sentido, siento, sentí;
        tener, teniendo, tenido, tengo, tuve;
        traer, trayendo, traído, traigo, traje;
        valer, valiendo, valido, valgo, valí;
        venir, veniendo, venido, vengo, vine;
        ver, viendo, visto, veo, ví;
        vestir, vistiendo, vestido, visto, vestí;
        volver, volviendo, vuelto, vuelvo, volví;

}


JAVARULESET

        String paradigm = (String)paradigms.get(input);
        // infinitive,progressive,participle,present,preterite,stem = (inf - ending)

        String infin, prog, part, pres, pret, stem;
        infin = prog = part = pres = pret = stem = "";  // make sure all are
            initialized
        int length = input.length();
        // if (authorFlag) trace("Paradigm is currently: " + paradigm);

        CONDITION ( paradigm == null ) {
                trace("Did NOT find "+ input +" in Verb Lookup Table");
                String tempPrefix, wordform;
                String[] tempVal = null;
                // collect prefixes - see if one matches input

                StringTokenizer tokens = new StringTokenizer(prefixes, " ");
                while (tokens.hasMoreTokens() && paradigm == null) {
                        tempPrefix = tokens.nextToken();
                        if (input.startsWith(tempPrefix) && (input.length() > 3) && (
                            prefix.equals(""))) {
                                wordform = input.substring(tempPrefix.length()); //
                                    strip prefix from input
                                trace("In computeMorph("+arg+"): "+ input +" is "+
                                    tempPrefix +" + "+ wordform);
                                trace("Testing Prefixes: "+ tempPrefix);
                                if ((wordform.length() > 1) && (paradigms.get(
                                    wordform)!=null)) {
                                        trace("Trying computeMorph("+ arg +") on "+
                                            wordform +" with prefix: "+ tempPrefix
                                             +", at depth "+ (1 + depth));
                                        tempVal = computeMorph(wordform, arg, depth
                                            +1, tempPrefix, suffix);
                                } else {
                                        trace("Did NOT find " + wordform + " (prefix
                                            stripped) in Verb Lookup Table");
                                }
                                if (tempVal != null) {
                                        trace("   returning " + tempVal.length + "
                                            computed values" + " at depth " + (1 +
                                            depth));
                                        return tempVal;
                                }
                        }
                }
                return null; // if prefixes has no token or tempVal == null
        }

        RULE ( irrVerb ) {
                Vector variants = new Vector();
                trace("Inside Spanish IrrVerb");
                LiteMorphRule[] useRules;
                StringTokenizer tokens = new StringTokenizer(paradigm, " ");
                infin = tokens.nextToken();
                prog = tokens.nextToken();
```

```
part = tokens.nextToken();
pres = tokens.nextToken();
pret = tokens.nextToken();

if (infin.length() < 4) {  // (ver,ser,ir)
        stem = infin.substring(0,infin.length()-1);
} else {
        stem = infin.substring(0,infin.length()-2);
}
trace("Found paradigm in Verb Lookup Table: "+ infin +" "+ prog
     +" "+ part +" "+ pres +" "+ pret);
// generates forms whether or not prefix = null, if prefix not null,
    included it.
trace("Using paradigm from Verb Lookup Table: "+ infin +" "+ prog
     +" "+ part +" "+ pres +" "+ pret);
if (infin.endsWith("ar"))  {
        applyrule(prefix+infin, Literal, variants);
        applyrule(prefix+prog, IrregProgPart, variants);
        applyrule(prefix+part, IrregProgPart, variants);

        applyrule(prefix+pres, ArIrregPres, variants);          //
            generate irreg present
        applyrule(prefix+stem, ArImperf, variants);             //
            generate imperfect for -ar
        applyrule(prefix+pret, ArIrregPret,  variants);          //
            generate irreg preterite
        applyrule(prefix+pres, ArIrregPresSubj, variants);      //
            generate present subjunctive
        applyrule(prefix+pret, ArIrregImpSubj, variants);       //
            generate imperfect subjunctive
        applyrule(prefix+stem, ArFutSubj, variants);            //
            generate future subjunctive
        applyrule(prefix+stem, ArIrregImperative, variants);    //
            generate imperative -ar

        applyrule(prefix+infin, AllCond, variants);
        applyrule(prefix+infin, AllFuture, variants);
}
if (infin.endsWith("er"))  {
        applyrule(prefix+infin, Literal, variants);
        applyrule(prefix+prog, IrregProgPart, variants);
        applyrule(prefix+part, IrregProgPart, variants);

        applyrule(prefix+pres, ErIrregPres, variants);          //
            generate irreg -er present
        applyrule(prefix+stem, ErIrImperf, variants);           //
            geneate imperfect for -er
        applyrule(prefix+pret, ErIrregPret, variants);          //
            generate irreg  -er preterite
        applyrule(prefix+pres, ErIrregPresSubj, variants);      //
            generate present subjunctive
        applyrule(prefix+pret, ErIrregImpSubj, variants);       //
            generate imperfect subjunctive
        applyrule(prefix+stem, ErFutSubj, variants);            //
            generate future subjunctive
        applyrule(prefix+pres, ErIrregImperative, variants);    //
            generate imperative -er

        applyrule(prefix+infin, AllCond, variants);
        applyrule(prefix+infin, AllFuture, variants);
}
if (infin.endsWith("ir") || infin.endsWith("ír"))  {              //
    need ír for oír
        applyrule(prefix+infin, Literal, variants);
```

```
                    applyrule(prefix+prog, IrregProgPart, variants);
                    applyrule(prefix+part, IrregProgPart, variants);

                    applyrule(prefix+pres, IrIrregPres, variants);          //
                        generate irreg -ir present
                    applyrule(prefix+stem, ErIrImperf, variants);           //
                        geneate imperfect for -ir
                    applyrule(prefix+pret, IrIrregPret, variants);          //
                        generate irreg  -ir preterite
                    applyrule(prefix+pres, IrIrregPresSubj, variants);      //
                        generate present subjunctive
                    applyrule(prefix+pret, IrIrregImpSubj, variants);       //
                        generate imperfect subjunctive
                    applyrule(prefix+stem, IrFutSubj, variants);            //
                        generate future subjunctive
                    applyrule(prefix+pres, IrIrregImperative, variants);    //
                        generate imperative -ir

                    applyrule(prefix+infin, AllCond, variants);
                    applyrule(prefix+infin, AllFuture, variants);
            }
            String[] result = new String[variants.size()];
            variants.copyInto(result);
            return result;
        }

ENDJAVARULESET
```

C.2.2 Sample rules set for Spanish LangLM

```
DEFAULT RULESET {

[...]
// test FUTURE -ar,-er,-ir forms

        .$Vowel .$Consonant + ?d r é -> TRY(irrVerb)ir,TRY(irrVerb)er;          // (
            sabré, saldré)
        .$Vowel .$Consonant + ?d r á ?s|n -> TRY(irrVerb)ir,TRY(irrVerb)er;     // (
            sabrás, saldrán)
        .$Vowel .$Consonant + ?d r é i s -> TRY(irrVerb)ir,TRY(irrVerb)er;      // (
            sabréis, saldréis)
        .$Vowel .$Consonant + ?d r e m o s -> TRY(irrVerb)ir,TRY(irrVerb)er;    // (
            sabremos, saldremos)

        .$Vowel .$Consonant a|e|i r + é -> TRY(irrVerb)_;     // (romperé, reiré) Note
             - need to handle infinitive reír
        .$Vowel .$Consonant a|e|i r + é -> (RegParadigm)_;  // (hablaré, comeré, viviré
            )

        .$Vowel .$Consonant a|e|i r + á ?s|n -> TRY(irrVerb)_;      // (darás, romperán
            , dormirá)
        .$Vowel .$Consonant a|e|i r + á ?s|n -> (RegParadigm)_;   // (hablará, comerán
            , vivirás)

        .$Vowel .$Consonant a|e|i r + é i s  -> TRY(irrVerb)_;      // (erraréis,
            romeréis, dormiréis)
        .$Vowel .$Consonant a|e|i r + é i s  -> (RegParadigm)_;   // (hablaréis,
            comeréis, viviréis)

        .$Vowel .$Consonant a|e|i r + e m o s -> TRY(irrVerb)_;     // (daremos,
            romperemos, dormiremos)
        .$Vowel .$Consonant a|e|i r + e m o s -> (RegParadigm)_;  // (hablaremos,
            comeremos, viviremos)
[...]

}
```

C.3 English LangLM

The English LangLM is 1,237 lines long, and contains 2,589 words for the 581 entries of the exception tables. Only 123 pattern-matching rules and 17 rule sets are used. The rule sets are all ending rule sets except for the default rule set. The following sample code is one of this ending rule set that process input words that end with *-ment*.

C.3.1 Sample ending rule set for English LangLM

```
/**
 * Authors : W. A. Woods and Ellen Hays
 * Copyright (C) 1995-2004 Sun Microsystems, Inc.
 *
 * Translation to LiteMorph : Mikaël ROUSSILLON
 * Copyright (C) 2004 University of New Brunswick
 */

// For words ending in -ment (ment-rules):

RULESET mentRules ENDING ment {
 s e g m e n t + -> s,ed,ing,ings,er,ers,ly,ness,nesses,less,ful;
            //  (e.g., segment, bisegment, cosegment)

 p i g m e n t + -> s,ed,ing,ings,er,ers,ly,ness,nesses,less,ful;
            //  (e.g., pigment, depigment, repigment)

 .aeiouy d g + m e n t -> *e;
            //  (e.g., judgment, abridgment)

 .aeiouy |bcdfghjklmnpqrstvwxyz + i m e n t -> *y;
            //  (e.g., merriment, embodiment)

 .aeiouy + m e n t -> _,*_;
            //  (e.g., atonement, entrapment)
}
```

APPENDIX D

Comparison of Lightweight Morphology, stemming and wildcard variants

Each of the following sections present sample words selected from a Ispell dictionary (randomly selected words) or from the Hansard collection (freqency selected words) for both French and English and their respective variants produced by Lightweight Morphology (LM), stemming (S) and wildcard (W). By variants we mean the words that were retrieved from the collection (the Hansard described in Chapter 6) by applying one of the three previously mentioned technique.

Each sample contains 15 words for each test case over a pool of 100 words. The variants are classified in either relevant or irrelevant. The relevance criterion is the correctness of the variants regarding to the input word, not the correctness of the context. From *tear*, we consider that *torn* is relevant, but *tearmann* is irrelevant. *Tear* will be classified as relevant either in the context of a teardrop or of something that is torn. The classification is not perfect for some words since 'relevance' is subjective. Changing the classification for such debatable word will however provide little consequences on the differential recall results.

D.1 Sample from English randomly selected words with variants created and relevance/irrelevance judgements

Word: acidly Stem: acidli Wildcard: acid*		
LM	Relevant	acid, acids
	Irrelevant	
S	Relevant	
	Irrelevant	
W	Relevant	acid, acidic, acidification, acidity, acids
	Irrelevant	
Word: awarder Stem: award Wildcard: award*		
LM	Relevant	award, awarded, awarding, awards
	Irrelevant	
S	Relevant	award, awarded, awarding, awards
	Irrelevant	
W	Relevant	award, awarded, awarding, awards
	Irrelevant	
Word: celebrated Stem: celebr Wildcard: celebrat*		
LM	Relevant	celebrate, celebrated, celebrates, celebrating
	Irrelevant	
S	Relevant	celebrate, celebrated, celebrates, celebrating, celebration, celebrations
	Irrelevant	celebrities, celebrity
W	Relevant	celebrate, celebrated, celebrates, celebrating, celebration, celebrations, celebratory
	Irrelevant	
Word: continue Stem: continu Wildcard: continu*		
LM	Relevant	continue, continued, continuer, continues, continuing
	Irrelevant	
S	Relevant	continual, continually, continuance, continuation, continue, continued, continuer, continues, continuing, continuity, continuous, continuously
	Irrelevant	
W	Relevant	continual, continually, continuance, continuation, continue, continued, continuer, continues, continuing, continuity, continuous, continuously
	Irrelevant	continuatinuation, continuum
Word: dreamer Stem: dreamer Wildcard: dream*		
LM	Relevant	dream, dreamed, dreamer, dreamers, dreaming, dreams
	Irrelevant	
S	Relevant	dreamer, dreamers
	Irrelevant	
W	Relevant	dream, dreamed, dreamer, dreamers, dreaming, dreams, dreamt
	Irrelevant	dreamland
Word: fastened Stem: fasten Wildcard: fast*		
LM	Relevant	fasten, fastened
	Irrelevant	
S	Relevant	fasten, fastened
	Irrelevant	
W	Relevant	fast, fasten, fastened, faster, fastest, fasting
	Irrelevant	fastidious
Word: hum Stem: hum Wildcard: hum*		
LM	Relevant	hum, humming
	Irrelevant	
S	Relevant	hum, humming
	Irrelevant	
W	Relevant	hum, humming

	Irrelevant	humaine, human, humane, humanely, humanism, humanist, humanistic, humanitarian, humanitarianism, humanitarians, humanities, humanity, humanization, humanize, humanized, humanizing, humankind, humanly, humans, humanum, humber, humberside, humble, humbled, humbleness, humblest, humbling, humbly, humboldt, humbug, humdinger, hume, humidity, humiliate, humiliated, humiliating, humiliation, humiliations, humility, hummell, hummingbird, humongous, humorist, humorous, humorously, humour, humoured, humouring, humourous, humours, hump, humpback, humped, humphrey, humphreys, humpty, humungous, humus
Word: lameness Stem: lame Wildcard: lame*		
LM	Relevant	lame, lamely, lameness, lamer
	Irrelevant	
S	Relevant	lame, lamely, lameness
	Irrelevant	
W	Relevant	lame, lamely, lameness, lamer
	Irrelevant	lamebrained, lament, lamentable, lamentably, lamented, lamenting, laments
Word: mendacious Stem: mendaci Wildcard: mendac*		
LM	Relevant	mendacious
	Irrelevant	
S	Relevant	mendacious
	Irrelevant	
W	Relevant	mendacious, mendacity
	Irrelevant	
Word: parsing Stem: pars Wildcard: pars*		
LM	Relevant	
	Irrelevant	parsement
S	Relevant	parse
	Irrelevant	
W	Relevant	parse
	Irrelevant	parsement, parsimonious, parsis, parson, parsons
Word: progression Stem: progress Wildcard: progress*		
LM	Relevant	progression
	Irrelevant	
S	Relevant	progress, progressed, progresses, progressing, progression, progressive, progressively, progressiveness, progressives, progressivity
	Irrelevant	
W	Relevant	progress, progressed, progresses, progressing, progression, progressive, progressively, progressiveness, progressives, progressivity
	Irrelevant	progressisve
Word: reflectivity Stem: reflect Wildcard: reflect*		
LM	Relevant	
	Irrelevant	
S	Relevant	reflect, reflected, reflecting, reflection, reflections, reflective, reflectively, reflects
	Irrelevant	
W	Relevant	reflect, reflected, reflecting, reflection, reflections, reflective, reflectively, reflects
	Irrelevant	
Word: shroud Stem: shroud Wildcard: shroud*		
LM	Relevant	shroud, shrouded, shrouds
	Irrelevant	
S	Relevant	shroud, shrouded, shrouds
	Irrelevant	
W	Relevant	shroud, shrouded, shrouds
	Irrelevant	
Word: spoilage Stem: spoilag Wildcard: spoil*		
LM	Relevant	
	Irrelevant	
S	Relevant	
	Irrelevant	
W	Relevant	spoil, spoiled, spoiler, spoiling, spoils
	Irrelevant	spoilsport
Word: turtles Stem: turtl Wildcard: turtle*		
LM	Relevant	turtle, turtles
	Irrelevant	
S	Relevant	turtle, turtles

	Irrelevant	
W	Relevant	turtle, turtles
	Irrelevant	turtleford, turtleneck, turtlenecks

D.2 Sample from English frequency selected words with variants created and relevance/irrelevance judgements

Word: acting Stem: act Wildcard: act*		
LM	Relevant	act, acted, acting, actor, actors, acts
	Irrelevant	
S	Relevant	act, acte, acted, actes, acting, acts
	Irrelevant	
W	Relevant	act, acte, acted, actes, acting, actor, actors, actress, actresses, acts
	Irrelevant	acta, acteal, actew, acti, actifs, action, actionable, actions, activate, activated, activates, activating, activation, activator, active, actively, activi, activism, activist, activists, activites, activities, activity, activités, acton, actuaire, actual, actuality, actualité, actualization, actualize, actualized, actualizing, actually, actuarial, actuarially, actuarials, actuaries, actuarily, actuary, actuated, actuel, actuels, actus
Word: business Stem: busi Wildcard: business*		
LM	Relevant	busier, busiest, busily, business, businesses, busy
	Irrelevant	
S	Relevant	business, businesses, busy
	Irrelevant	
W	Relevant	business, businesses, businessman, businessmen, businesspeople, businessperson, businesswoman, businesswomen
	Irrelevant	businesslike, businesslinc
Word: country Stem: countri Wildcard: countr*		
LM	Relevant	countries, country
	Irrelevant	
S	Relevant	countries, country
	Irrelevant	
W	Relevant	countries, country
	Irrelevant	countradict, countryman, countrymen, countryside, countrywide
Word: deputy Stem: deputi Wildcard: deput*		
LM	Relevant	deputies, deputy
	Irrelevant	
S	Relevant	deputies, deputy
	Irrelevant	
W	Relevant	deputation, deputations, deputies, deputy
	Irrelevant	deputized
Word: finance Stem: financ Wildcard: financ*		
LM	Relevant	finance, financed, financement, finances, financing
	Irrelevant	
S	Relevant	finance, financed, financement, finances, financing
	Irrelevant	
W	Relevant	finance, financed, financement, finances, financial, financially, financier, financiers, financing, financière, financières, financiére
	Irrelevant	
Word: income Stem: incom Wildcard: incom*		
LM	Relevant	income, incomes, incoming
	Irrelevant	
S	Relevant	income, incomes, incoming
	Irrelevant	
W	Relevant	income, incomes, incoming

	Irrelevant	incommensurable, incommensurate, incomparable, incompatibility, incompatible, incompetence, incompetency, incompetent, incompetently, incompetents, incomplete, incompletely, incomprehensibility, incomprehensible, incompétence
Word: justice Stem: justic Wildcard: justice*		
LM	Relevant	justice, justices
	Irrelevant	
S	Relevant	justice, justices
	Irrelevant	
W	Relevant	justice, justices
	Irrelevant	justicegerard, justiceship, justiceships
Word: money Stem: monei Wildcard: money*		
LM	Relevant	money, moneyed, moneys
	Irrelevant	
S	Relevant	money, moneyed, moneys
	Irrelevant	
W	Relevant	money, moneyed, moneys
	Irrelevant	
Word: parliamentary Stem: parliamentari Wildcard: parliament*		
LM	Relevant	parliament, parliamentary, parliaments
	Irrelevant	
S	Relevant	parliamentary
	Irrelevant	
W	Relevant	parliament, parliamentarian, parliamentarianism, parliamentarians, parliamentarily, parliamentarism, parliamentary, parliamentry, parliaments
	Irrelevant	parliamentand, parliamentarianto, parliamented, parliamenti
Word: prime Stem: prime Wildcard: prime*		
LM	Relevant	prime, primed, primer, primes
	Irrelevant	
S	Relevant	prime, primed, primes
	Irrelevant	
W	Relevant	prime, primed, primer, primes
	Irrelevant	primeau, primem
Word: provinces Stem: provinc Wildcard: provinc*		
LM	Relevant	province, provinces
	Irrelevant	
S	Relevant	province, provinces
	Irrelevant	
W	Relevant	province, provinces, provincial, provinciale, provincialism, provincially, provincials
	Irrelevant	provincehood
Word: resources Stem: resourc Wildcard: resource*		
LM	Relevant	resource, resources
	Irrelevant	resourced, resourceful, resourcing
S	Relevant	resource, resources
	Irrelevant	resourced, resourceful, resourcefulness, resourcing
W	Relevant	resource, resources
	Irrelevant	resourced, resourceful, resourcefulness
Word: services Stem: servic Wildcard: servic*		
LM	Relevant	service, serviced, services, servicing
	Irrelevant	servicer
S	Relevant	service, serviceability, serviceable, serviced, services, servicing
	Irrelevant	servicer
W	Relevant	service, serviceability, serviceable, serviced, serviceman, servicemen, services, servicing
	Irrelevant	servicer
Word: system Stem: system Wildcard: system*		
LM	Relevant	system, systems
	Irrelevant	
S	Relevant	system, systems
	Irrelevant	systemic, systemically
W	Relevant	system, systems
	Irrelevant	systematic, systematically, systemhouse, systemic, systemically
Word: world Stem: world Wildcard: world*		
LM	Relevant	world, worldly, worlds
	Irrelevant	

S	Relevant	world, worlds
	Irrelevant	
W	Relevant	world, worldly, worlds, worldwide
	Irrelevant	worldclass, worldiwide

D.3 Sample from French randomly selected words with variants created and relevance/irrelevance judgements

Word: aboutissement Stem: about Wildcard: abouti*		
LM	Relevant	abouti, aboutir, aboutira, aboutiraient, aboutirait, aboutirez, aboutirions, aboutirons, aboutiront, aboutissaient, aboutissais, aboutissait, aboutissant, aboutissants, aboutisse, aboutissement, aboutissent, aboutissions, aboutissons, aboutit
	Irrelevant	
S	Relevant	abouti, aboutir, aboutira, aboutiraient, aboutirait, aboutirez, aboutirions, aboutirons, aboutiront, aboutissaient, aboutissais, aboutissait, aboutissant, aboutissants, aboutisse, aboutissement, aboutissent, aboutissions, aboutissons, aboutit
	Irrelevant	
W	Relevant	abouti, aboutir, aboutira, aboutiraient, aboutirait, aboutirez, aboutirions, aboutirons, aboutiront, aboutissaient, aboutissais, aboutissait, aboutissant, aboutissants, aboutisse, aboutissement, aboutissent, aboutissions, aboutissons, aboutit
	Irrelevant	
Word: amaigrissait Stem: amaigr Wildcard: amaigri*		
LM	Relevant	amaigri, amaigris, amaigrissant, amaigrissement
	Irrelevant	
S	Relevant	amaigri, amaigris, amaigrissant, amaigrissement
	Irrelevant	
W	Relevant	amaigri, amaigris, amaigrissant, amaigrissement
	Irrelevant	
Word: brasserai Stem: brass Wildcard: brass*		
LM	Relevant	brassage, brassaient, brassait, brasse, brassent, brasser, brassera, brassez, brassé, brassée
	Irrelevant	
S	Relevant	brassaient, brassait, brasse, brasser, brassera, brassez, brassé, brassée
	Irrelevant	brass
W	Relevant	brassage, brassaient, brassait, brasse, brassent, brasser, brassera, brasserie, brasseries, brasseur, brasseurs, brassez, brassé, brassée
	Irrelevant	brass, brassard, brassicole
Word: conduite Stem: conduit Wildcard: condui*		
LM	Relevant	conducteur, conducteurs, conduira, conduiraient, conduirait, conduire, conduiront, conduis, conduisaient, conduisais, conduisait, conduisant, conduise, conduisent, conduisez, conduisions, conduisit, conduisons, conduit, conduite, conduites, conduits
	Irrelevant	
S	Relevant	conduit, conduite, conduites, conduits
	Irrelevant	
W	Relevant	conduira, conduiraient, conduirait, conduire, conduiront, conduis, conduisaient, conduisais, conduisait, conduisant, conduise, conduisent, conduisez, conduisions, conduisit, conduisons, conduit, conduite, conduites, conduits
	Irrelevant	
Word: crampe Stem: cramp Wildcard: cramp*		
LM	Relevant	crampes
	Irrelevant	crampons
S	Relevant	crampes
	Irrelevant	
W	Relevant	crampes
	Irrelevant	crampon, cramponnant, cramponne, cramponner, cramponnés, crampons

Word: déportais Stem: déport Wildcard: déport*		
LM	Relevant	déportation, déportations, déporte, déporter, déporterions, déportons, déporté, déportée, déportées, déportés
	Irrelevant	
S	Relevant	déportation, déportations, déporte, déporter, déporterions, déporté, déportée, déportées, déportés
	Irrelevant	
W	Relevant	déportation, déportations, déporte, déporter, déporterions, déportons, déporté, déportée, déportées, déportés
	Irrelevant	
Word: enlacerais Stem: enlac Wildcard: enlac*		
LM	Relevant	enlacer
	Irrelevant	
S	Relevant	enlacer
	Irrelevant	
W	Relevant	enlacer
	Irrelevant	
Word: germe Stem: germ Wildcard: germ*		
LM	Relevant	germe, germer, germera, germes, germs, germé, germées
	Irrelevant	
S	Relevant	germe, germer, germera, germes, germs, germé, germées
	Irrelevant	
W	Relevant	germe, germer, germera, germes, germinal, germinale, germinales, germination, germs, germé, germées
	Irrelevant	germain, germaine, germains, germano, germen
Word: incompétente Stem: incompétent Wildcard: incompét*		
LM	Relevant	incompétent, incompétente, incompétentes, incompétents
	Irrelevant	
S	Relevant	incompétence, incompétences, incompétent, incompétente, incompétentes, incompétents
	Irrelevant	
W	Relevant	incompétence, incompétences, incompétent, incompétente, incompétentes, incompétents
	Irrelevant	
Word: jargonnais Stem: jargon Wildcard: jargon*		
LM	Relevant	jargon
	Irrelevant	
S	Relevant	jargon
	Irrelevant	
W	Relevant	jargon
	Irrelevant	
Word: orange Stem: orang Wildcard: orang*		
LM	Relevant	orange, oranges, orangée
	Irrelevant	
S	Relevant	orange, oranges, orangée
	Irrelevant	orangiste, orangistes
W	Relevant	orange, oranges, orangée
	Irrelevant	orangeville, orangiste, orangistes
Word: plombai Stem: plomb Wildcard: plomb*		
LM	Relevant	plombage, plombée, plombées, plombés
	Irrelevant	
S	Relevant	plomb, plombs, plombée, plombées, plombés
	Irrelevant	
W	Relevant	plomb, plombage, plombs, plombée, plombées, plombés
	Irrelevant	plomberie, plombier, plombiers
Word: rédigeais Stem: rédig Wildcard: rédig*		
LM	Relevant	rédige, rédigea, rédigeaient, rédigeais, rédigeait, rédigeant, rédigent, rédigeons, rédiger, rédigera, rédigerait, rédigeront, rédigez, rédigions, rédigé, rédigée, rédigées, rédigés
	Irrelevant	
S	Relevant	rédige, rédigea, rédigeaient, rédigeais, rédigeait, rédigeant, rédiger, rédigera, rédigerait, rédigeront, rédigez, rédigions, rédigé, rédigée, rédigées, rédigés
	Irrelevant	
W	Relevant	rédige, rédigea, rédigeaient, rédigeais, rédigeait, rédigeant, rédigent, rédigeons, rédiger, rédigera, rédigerait, rédigeront, rédigez, rédigions, rédigé, rédigée, rédigées, rédigés

	Irrelevant	
Word: villégiaturer Stem: villégiatur Wildcard: villégiatur*		
LM	Relevant	villégiature
	Irrelevant	
S	Relevant	villégiature
	Irrelevant	
W	Relevant	villégiature
	Irrelevant	
Word: étalagerais Stem: étalag Wildcard: étalag*		
LM	Relevant	étalage, étalages
	Irrelevant	
S	Relevant	étalage, étalages
	Irrelevant	
W	Relevant	étalage, étalages
	Irrelevant	

D.4 Sample from French frequency selected words with variants created and relevance/irrelevance judgements

Word: accord Stem: accord Wildcard: accord*		
LM	Relevant	accord, accorde, accords
	Irrelevant	
S	Relevant	accord, accorda, accordaient, accordais, accordait, accordance, accordant, accorde, accorder, accordera, accorderai, accorderaient, accorderais, accorderait, accorderez, accorderiez, accorderions, accorderons, accorderont, accordez, accordiez, accordions, accords, accordé, accordée, accordées, accordés
	Irrelevant	
W	Relevant	accord, accorda, accordaient, accordais, accordait, accordance, accordant, accorde, accordent, accorder, accordera, accorderai, accorderaient, accorderais, accorderait, accorderez, accorderiez, accorderions, accorderons, accorderont, accordeur, accordez, accordiez, according, accordions, accordons, accords, accordé, accordée, accordées, accordés
	Irrelevant	accordéoniste
Word: autochtones Stem: autochton Wildcard: autochtone*		
LM	Relevant	autochtone, autochtones
	Irrelevant	
S	Relevant	autochtone, autochtones
	Irrelevant	
W	Relevant	autochtone, autochtones
	Irrelevant	
Word: comité Stem: comit Wildcard: comité*		
LM	Relevant	comité, comités
	Irrelevant	comite
S	Relevant	comité, comités
	Irrelevant	comite
W	Relevant	comité, comités
	Irrelevant	comitédu, comitéle, comitésénatorial, comitéà
Word: dollars Stem: dollar Wildcard: dollar*		
LM	Relevant	dollar, dollars
	Irrelevant	
S	Relevant	dollar, dollars
	Irrelevant	
W	Relevant	dollar, dollarisation, dollars
	Irrelevant	dollarama, dollard, dollards
Word: enfants Stem: enfant Wildcard: enfant*		

LM	Relevant	enfance, enfant, enfants
	Irrelevant	enfer
S	Relevant	enfant, enfants
	Irrelevant	
W	Relevant	enfant, enfantillage, enfantillages, enfantin, enfantine, enfants
	Irrelevant	
Word: honorable Stem: honor Wildcard: honorabl*		
LM	Relevant	honorable, honorables, honorait, honorant, honore, honorent, honorer, honorera, honorerai, honorerait, honoreras, honorerions, honorerons, honoreront, honorez, honorions, honorons, honoré, honorée, honorées, honorés
	Irrelevant	
S	Relevant	honorable, honorablement, honorables, honorait, honorant, honore, honorer, honorera, honorerai, honorerait, honoreras, honorerions, honorerons, honoreront, honorez, honorions, honoré, honorée, honorées, honorés
	Irrelevant	honoris
W	Relevant	honorable, honorablement, honorables
	Irrelevant	honorableandré, honorableb, honorablebarney, honorablebrenda, honorablecharles, honorableconsiglio, honorabledavid, honorabledonald, honorabledouglas, honorableedmund, honorableeric, honorableerik, honorableerminie, honorableeymard, honorablefernand, honorablegérald, honorablehedy, honorableherbert, honorablej, honorablejames, honorablejean, honorablejerahmiel, honorablejoe, honorablejohn, honorableleonard, honorablelowell, honorablem, honorablemabel, honorablemarcel, honorablemarjorie, honorablemarjory, honorablemichael, honorablenorman, honorablenoël, honorableorville, honorablepat, honorablepeter, honorablepierre, honorableralph, honorablerichard, honorablerobert, honorableroméo, honorablesduncan, honorablesheila, honorablesnoël, honorablesénateur, honorableterry, honorabletony, honorableénateur
Word: libéral Stem: libéral Wildcard: libéra*		
LM	Relevant	libéral, libérale, libérales, libéraux
	Irrelevant	
S	Relevant	libéral, libérale, libéralement, libérales, libéralisme, libéralistes, libéraux
	Irrelevant	
W	Relevant	libéral, libérale, libéralement, libérales, libéralisait, libéralisant, libéralisation, libéralisatrice, libéralise, libéralisent, libéraliser, libéralisme, libéralistes, libéralisé, libéralisées, libéralisés, libéraux
	Irrelevant	libérables, libéraient, libérait, libéralmobile, libérant, libérateurs, libération, libérations, libératoire, libératrice
Word: ministère Stem: minister Wildcard: ministère*		
LM	Relevant	minister, ministers, ministère, ministères
	Irrelevant	
S	Relevant	ministers, ministère, ministères
	Irrelevant	
W	Relevant	ministère, ministères
	Irrelevant	ministèresle
Word: parlementaire Stem: parlementair Wildcard: parlement*		
LM	Relevant	parlementaire, parlementairement, parlementaires
	Irrelevant	
S	Relevant	parlementaire, parlementairement, parlementaires
	Irrelevant	
W	Relevant	parlement, parlementaire, parlementairement, parlementaires, parlementarisme, parlementer, parlements, parlementé
	Irrelevant	parlementairede, parlementaireno, parlementairno
Word: problèmes Stem: problem Wildcard: problème*		
LM	Relevant	problème, problèmes
	Irrelevant	
S	Relevant	problème, problèmes
	Irrelevant	
W	Relevant	problème, problèmes
	Irrelevant	problèmens
Word: président Stem: président Wildcard: présid*		
LM	Relevant	présida, présidaient, présidais, présidait, présidant, préside, président, présidente, présidentes, présidents, présider, présidera, présiderai, présideraient, présiderait, présiderez, présideront, présidez, présidiez, présidons, présidé, présidée, présidées, présidés

	Irrelevant	
S	Relevant	présidence, présidences, président, présidente, présidentes, présidents
	Irrelevant	
W	Relevant	présida, présidaient, présidais, présidait, présidant, préside, présidence, présidences, président, présidente, présidentes, présidentialisation, présidentiel, présidentielle, présidentielles, présidentiels, présidents, présider, présidera, présiderai, présideraient, présiderait, présiderez, présideront, présidez, présidiez, présidons, présidé, présidée, présidées, présidés
	Irrelevant	présidentselon
Word: ressources Stem: ressourc Wildcard: ressource*		
LM	Relevant	ressource, ressourcement, ressourcer, ressources
	Irrelevant	
S	Relevant	ressource, ressourcement, ressourcer, ressources
	Irrelevant	
W	Relevant	ressource, ressourcement, ressourcer, ressources
	Irrelevant	
Word: secrétaire Stem: secrétair Wildcard: secrétaire*		
LM	Relevant	secrétaire, secrétaires
	Irrelevant	
S	Relevant	secrétaire, secrétaires
	Irrelevant	
W	Relevant	secrétaire, secrétaires
	Irrelevant	
Word: sénat Stem: sénat Wildcard: sénat*		
LM	Relevant	sénat, sénats
	Irrelevant	
S	Relevant	sénat, sénats
	Irrelevant	
W	Relevant	sénat, sénateur, sénatorial, sénatoriale, sénatoriales, sénatorialle, sénatoriaux, sénatrice, sénatrices, sénats
	Irrelevant	sénatdu, sénaten, sénateuralan, sénateurbrenda, sénateurdi, sénateurdoris, sénateurearl, sénateurfrank, sénateurguy, sénateurjean, sénateurjohn, sénateurjoyal, sénateurmarian, sénateurmuriel, sénateurmurray, sénateurnick, sénateurpeter, sénateurpietro, sénateurprud, sénateurrichard, sénateurroch, sénateurs, sénateursdavid, sénateurst, sénateurwilliam, sénatle, sénatroial
Word: égard Stem: égard Wildcard: égard		
LM	Relevant	égard, égards
	Irrelevant	
S	Relevant	égard, égards
	Irrelevant	
W	Relevant	égard
	Irrelevant	

APPENDIX E

Differential Recall results

The differential recall results are computed using the relevance/irrelevance classification in Appendix D with the Hansard collection (see Chapter 6). As mentioned before, LM, S, W and EQ stands respectively for Lightweight Morphology, Stemming, Wildcard and Exact Query. The following scores are computed when comparing two systems A and B:

$$\begin{aligned} \cap_B^A &= \text{Number of relevant documents returned by both } A \text{ and } B \\ \Delta_B^A &= \text{Number of relevant documents returned by } A \text{ but not } B \\ \Delta_A^B &= \text{Number of relevant documents returned by } B \text{ but not } A \end{aligned}$$

As described in Chapter 6, for each set of words, two differential recall scores are computed. The first one, as we just defined, uses a relevance criterion where we count the number of documents containing relevant terms, therefore the number of relevant document. On the second differential recall score, we use an irrelevance criterion where we count the number of documents containing irrelevant terms and no relevant terms, therefore the number of irrelevant documents.

For the irrelevance criterion, the words that had all their differential recall measures for the different comparison equal to zero were not printed.

E.1 Differential recall results for English randomly selected words

E.1.1 Relevance criterion

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
acidly	37	0	0	0	37	1	37	0	0	0	0	0
alight	0	2	0	0	2	0	1	1	0	1	1	0
allowance	0	204	321	0	204	321	53	151	0	374	151	0
among	4	500	0	0	504	0	4	500	0	0	500	0
amputates	0	6	4	0	6	4	6	0	0	10	0	0
articulates	0	202	7	0	202	7	190	12	0	197	12	0
artist	0	153	16	0	153	19	109	44	0	125	44	0
awarder	0	388	0	0	388	0	388	0	0	388	0	0
belongs	0	388	0	0	388	0	152	236	0	152	236	0
bitingly	23	0	0	11	12	0	23	0	0	0	0	0
boarders	493	5	0	0	498	0	497	1	0	4	1	0
brazen	4	6	0	0	10	0	4	6	0	0	6	0
bumping	8	33	0	0	41	3	39	2	0	31	2	0
canceling	0	3	253	0	3	253	1	2	0	254	2	0
celebrated	0	397	24	0	397	24	257	140	0	281	140	0
chokes	0	54	0	0	54	0	51	3	0	51	3	0
classified	0	71	0	0	71	46	23	48	0	23	48	0
clawed	0	55	0	0	55	0	26	29	0	26	29	0
clever	12	40	0	0	52	0	13	39	0	1	39	0
commit	0	514	0	0	514	10	177	337	0	177	337	0
compiled	0	48	9	0	48	9	12	36	0	21	36	0
continue	0	522	2	0	522	2	4	518	0	6	518	0
coordinations	0	35	56	0	35	56	35	0	0	91	0	0
crumb	0	24	0	0	24	32	23	1	0	23	1	0
cursors	0	0	0	0	0	0	0	0	0	0	0	0
descriptively	0	12	170	0	12	171	12	0	0	182	0	0
dirtiness	3	107	0	0	110	20	110	0	0	107	0	0
donkey	0	6	0	0	6	0	1	5	0	1	5	0
dreamer	280	7	0	0	287	5	284	3	0	4	3	0

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
driveway	0	17	0	0	17	0	4	13	0	4	13	0
education	0	483	7	0	483	7	0	483	0	7	483	0
encircle	0	4	0	0	4	0	2	2	0	2	2	0
engages	0	463	0	0	463	0	423	40	0	423	40	0
estimation	0	49	381	0	49	381	0	49	0	381	49	0
expounds	0	20	0	0	20	0	17	3	0	17	3	0
fastened	0	4	0	0	4	357	3	1	0	3	1	0
fierce	0	60	0	0	60	0	21	39	0	21	39	0
foulness	0	44	0	0	44	0	44	0	0	44	0	0
frisks	0	3	0	0	3	0	3	0	0	3	0	0
hatching	1	11	0	0	12	13	12	0	0	11	0	0
houses	0	529	0	0	529	0	243	286	0	243	286	0
huge	0	434	0	0	434	0	1	433	0	1	433	0
hum	0	2	0	0	2	0	1	1	0	1	1	0
hundred	0	263	181	0	263	183	0	263	0	181	263	0
incurred	0	205	0	0	205	0	53	152	0	53	152	0
indignity	0	21	46	0	21	0	5	16	0	51	16	0
indivisible	0	42	6	0	42	0	0	42	0	6	42	0
journeyed	0	108	0	0	108	7	103	5	0	103	5	0
kitchens	0	191	0	0	191	0	165	26	0	165	26	0
lameness	32	28	0	0	60	0	58	2	0	26	2	0
landlord	0	32	0	0	32	0	13	19	0	13	19	0
leaf	223	110	0	223	110	0	234	99	0	11	99	0
less	18	500	0	14	504	0	18	500	0	0	500	0
looter	29	5	0	0	34	0	32	2	0	3	2	0
mailable	0	0	0	0	0	358	0	0	0	0	0	0
marketed	0	465	0	0	465	0	435	30	0	435	30	0
mendacious	0	1	0	0	1	1	0	1	0	0	1	0
obscured	0	72	6	0	72	6	68	4	0	74	4	0
occupier	0	214	0	0	214	87	212	2	0	212	2	0
openers	2	510	0	0	512	0	512	0	0	510	0	0
outdoors	0	34	0	0	34	0	20	14	0	20	14	0
paddle	0	18	0	0	18	0	5	13	0	5	13	0
palace	0	14	0	0	14	0	2	12	0	2	12	0
parsing	0	0	4	0	0	4	0	0	0	4	0	0
pastime	0	9	0	0	9	0	2	7	0	2	7	0
perplexed	0	35	1	0	35	1	15	20	0	16	20	0
pitier	81	0	0	0	81	1	81	0	0	0	0	0
poetical	0	12	0	0	12	61	12	0	0	12	0	0
presumptions	0	59	6	0	59	23	50	9	0	56	9	0
prime	0	512	0	0	512	0	0	512	0	0	512	0
progression	0	25	469	0	25	469	0	25	0	469	25	0
prowlers	2	0	0	0	2	0	2	0	0	0	0	0
pumpkin	0	2	0	0	2	0	0	2	0	0	2	0
punctuation	0	1	2	0	1	2	0	1	0	2	1	0
quadrupled	0	13	0	0	13	1	5	8	0	5	8	0
raise	0	518	0	0	518	0	54	464	0	54	464	0
rave	0	24	0	0	24	0	16	8	0	16	8	0
reflectivity	0	0	503	0	0	503	0	0	0	503	0	0
reinserted	0	2	0	0	2	0	2	0	0	2	0	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
renumbering	0	28	0	0	28	0	1	27	0	1	27	0
rhythms	0	0	0	0	0	1	0	0	0	0	0	0
roarer	27	0	0	0	27	0	27	0	0	0	0	0
salesmen	12	4	0	12	4	0	12	4	0	0	4	0
showered	0	26	0	0	26	0	22	4	0	22	4	0
shroud	0	16	0	0	16	0	7	9	0	7	9	0
singers	163	39	0	17	185	0	193	9	0	30	9	0
slaying	24	8	0	23	9	0	28	4	0	4	4	0
slicer	27	0	0	0	27	0	27	0	0	0	0	0
sovereignty	0	291	0	0	291	94	0	291	0	0	291	0
special	0	510	1	0	510	3	0	510	0	1	510	0
speculation	0	100	81	0	100	81	6	94	0	87	94	0
spoilage	0	0	0	0	0	27	0	0	0	0	0	0
statistical	0	390	0	0	390	0	316	74	0	316	74	0
subverter	0	24	0	0	24	9	24	0	0	24	0	0
swims	10	62	0	5	67	2	70	2	0	60	2	0
tablet	0	9	0	0	9	514	8	1	0	8	1	0
tear	55	182	1	55	182	1	152	85	0	98	85	0
tonnage	0	10	0	0	10	122	0	10	0	0	10	0
turtles	0	4	0	0	4	0	3	1	0	3	1	0
wrenches	0	47	0	0	47	0	42	5	0	42	5	0

E.1.2 Irrelevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
among	0	0	1	0	0	1	0	0	0	1	0	0
boarders	0	0	0	0	0	21	0	0	0	0	0	0
brazen	0	0	0	0	0	69	0	0	0	0	0	0
celebrated	0	0	5	0	0	0	0	0	0	5	0	0
clawed	0	0	0	0	0	45	0	0	0	0	0	0
commit	0	0	0	0	0	13	0	0	0	0	0	0
continue	0	0	0	0	0	21	0	0	0	0	0	0
cursors	0	0	0	0	0	22	0	0	0	0	0	0
dreamer	0	0	0	0	0	2	0	0	0	0	0	0
fastened	0	0	0	0	0	2	0	0	0	0	0	0
frisks	0	0	0	0	0	1	0	0	0	0	0	0
hatching	0	0	0	0	0	6	0	0	0	0	0	0
houses	0	0	0	0	0	213	0	0	0	0	0	0
huge	0	0	0	0	0	1	0	0	0	0	0	0
hum	0	0	0	0	0	521	0	0	0	0	0	0
hundred	0	0	0	0	0	1	0	0	0	0	0	0
incurred	0	0	12	0	0	28	0	0	0	12	0	0
lameness	0	0	0	0	0	62	0	0	0	0	0	0
leaf	0	0	0	0	0	8	0	0	0	0	0	0
less	0	0	0	0	0	283	0	0	0	0	0	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
mailable	0	0	0	0	0	8	0	0	0	0	0	0
marketed	0	0	0	0	0	198	0	0	0	0	0	0
occupier	0	0	0	0	0	1	0	0	0	0	0	0
openers	0	0	0	0	0	13	0	0	0	0	0	0
paddle	0	0	0	0	0	1	0	0	0	0	0	0
parsing	1	0	0	0	1	11	1	0	0	0	0	0
pitier	0	0	0	0	0	207	0	0	0	0	0	0
prime	0	0	0	0	0	4	0	0	0	0	0	0
progression	0	0	0	0	0	1	0	0	0	0	0	0
pumpkin	0	0	0	0	0	1	0	0	0	0	0	0
raise	0	0	0	0	0	40	0	0	0	0	0	0
rave	0	0	0	0	0	41	0	0	0	0	0	0
singers	0	0	0	0	0	473	0	0	0	0	0	0
slicer	0	0	0	0	0	10	0	0	0	0	0	0
sovereignty	0	0	0	0	0	1	0	0	0	0	0	0
spoilage	0	0	0	0	0	1	0	0	0	0	0	0
statistical	0	0	1	0	0	0	0	0	0	1	0	0
tablet	0	0	0	0	0	489	0	0	0	0	0	0
tear	0	0	0	0	0	1	0	0	0	0	0	0
tonnage	0	0	0	0	0	3	0	0	0	0	0	0
turtles	0	0	0	0	0	3	0	0	0	0	0	0

E.2 Differential recall results for English frequency selected words

E.2.1 Relevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
acting	0	531	0	0	531	0	48	483	0	48	483	0
affairs	0	518	0	0	518	0	1	517	0	1	517	0
agreement	12	516	0	0	528	0	13	515	0	1	515	0
amendment	0	523	0	10	513	0	30	493	0	30	493	0
amendments	0	523	0	10	513	0	43	480	0	43	480	0
bill	0	528	0	0	528	0	0	528	0	0	528	0
budget	0	492	0	0	492	1	9	483	0	9	483	0
business	0	522	0	1	521	1	4	518	0	4	518	0
canadian	0	526	0	0	526	5	1	525	0	1	525	0
children	3	508	0	0	511	1	3	508	0	0	508	0
colleague	0	521	0	0	521	0	20	501	0	20	501	0
committee	0	524	0	0	524	0	1	523	0	1	523	0
commons	0	528	0	0	528	0	2	526	0	2	526	0

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
community	0	517	0	0	517	0	9	508	0	9	508	0
country	0	519	0	0	519	0	0	519	0	0	519	0
court	0	496	0	0	496	0	11	485	0	11	485	0
criminal	0	464	0	0	464	0	10	454	0	10	454	0
deal	2	519	0	0	521	0	4	517	0	2	517	0
debate	0	527	0	0	527	0	4	523	0	4	523	0
decision	0	515	0	0	515	0	5	510	0	5	510	0
department	0	510	0	510	0	0	7	503	0	7	503	0
deputy	0	523	0	0	523	0	0	523	0	0	523	0
development	0	522	0	0	522	0	11	511	0	11	511	0
division	0	422	6	0	422	0	6	416	0	12	416	0
economic	0	503	0	0	503	0	4	499	0	4	499	0
education	0	483	7	0	483	7	0	483	0	7	483	0
farmers	35	376	0	0	411	2	47	364	0	12	364	0
federal	0	517	1	0	517	1	0	517	0	1	517	0
finance	0	498	0	0	498	21	10	488	0	10	488	0
foreign	0	494	0	0	494	0	1	493	0	1	493	0
future	0	517	0	0	517	0	1	516	0	1	516	0
government	0	530	0	0	530	3	0	530	0	0	530	0
health	0	514	0	0	514	2	0	514	0	0	514	0
honourable	0	407	124	0	407	124	4	403	0	128	403	0
human	0	515	1	0	515	2	0	515	0	1	515	0
income	0	478	0	0	478	0	6	472	0	6	472	0
industry	0	496	8	0	496	8	4	492	0	12	492	0
information	0	514	14	0	514	0	0	514	0	14	514	0
interest	0	518	0	0	518	0	8	510	0	8	510	0
international	0	515	0	0	515	0	0	515	0	0	515	0
issue	0	526	0	0	526	0	0	526	0	0	526	0
issues	0	526	0	0	526	0	10	516	0	10	516	0
justice	0	500	0	0	500	0	0	500	0	0	500	0
law	0	510	0	0	510	4	0	510	0	0	510	0
leader	0	526	0	0	526	0	2	524	0	2	524	0
legislation	0	515	3	0	515	3	0	515	0	3	515	0
liberal	0	494	1	0	494	1	4	490	0	5	490	0
members	0	526	0	0	526	0	3	523	0	3	523	0
minister	0	523	0	0	523	0	0	523	0	0	523	0
money	0	504	0	0	504	0	0	504	0	0	504	0
motion	0	527	0	0	527	0	0	527	0	0	527	0
national	0	523	2	0	523	1	0	523	0	2	523	0
opportunity	0	517	0	0	517	0	1	516	0	1	516	0
opposition	0	519	0	0	519	0	0	519	0	0	519	0
order	0	531	0	0	531	0	0	531	0	0	531	0
parliament	1	530	0	0	531	0	1	530	0	0	530	0
parliamentary	35	496	0	0	531	0	35	496	0	0	496	0
party	0	517	1	19	498	0	19	498	0	20	498	0
pay	13	507	0	3	517	0	18	502	0	5	502	0
people	1	523	0	1	523	0	1	523	0	0	523	0
policy	0	513	0	0	513	0	6	507	0	6	507	0
political	0	516	0	0	516	1	4	512	0	4	512	0
position	0	518	0	0	518	0	1	517	0	1	517	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
prime	0	512	0	0	512	0	0	512	0	0	512	0
private	0	498	5	0	498	5	5	493	0	10	493	0
problem	0	521	0	0	521	0	5	516	0	5	516	0
process	0	516	0	0	516	1	2	514	0	2	514	0
program	0	514	0	0	514	0	5	509	0	5	509	0
provide	0	523	0	0	523	0	6	517	0	6	517	0
province	0	516	0	0	516	2	10	506	0	10	506	0
provinces	0	516	0	0	516	2	14	502	0	14	502	0
provincial	0	509	0	0	509	9	0	509	0	0	509	0
public	0	522	0	0	522	0	0	522	0	0	522	0
question	0	522	0	0	522	0	0	522	0	0	522	0
questions	0	522	0	0	522	0	4	518	0	4	518	0
reform	0	484	1	0	484	1	14	470	0	15	470	0
report	0	523	0	0	523	0	0	523	0	0	523	0
resources	0	504	0	0	504	0	2	502	0	2	502	0
respect	0	530	0	0	530	0	5	525	0	5	525	0
rights	0	526	0	0	526	0	14	512	0	14	512	0
secretary	0	455	1	0	455	10	1	454	0	2	454	0
senate	0	480	13	0	480	13	0	480	0	13	480	0
senator	0	407	86	0	407	0	55	352	0	141	352	0
senators	0	407	86	0	407	86	45	362	0	131	362	0
services	0	522	0	0	522	0	11	511	0	11	511	0
situation	0	518	0	0	518	0	2	516	0	2	516	0
social	0	508	0	0	508	1	0	508	0	0	508	0
society	0	499	0	0	499	1	2	497	0	2	497	0
speaker	0	533	0	0	533	0	0	533	0	0	533	0
states	1	525	0	0	526	0	10	516	0	9	516	0
support	0	520	0	0	520	0	1	519	0	1	519	0
system	0	518	0	0	518	0	1	517	0	1	517	0
tax	0	499	0	0	499	12	4	495	0	4	495	0
taxes	0	499	0	0	499	12	70	429	0	70	429	0
trade	0	500	0	0	500	1	9	491	0	9	491	0
united	0	516	0	0	516	2	2	514	0	2	514	0
vote	1	500	0	1	500	0	14	487	0	13	487	0
women	9	485	0	9	485	0	9	485	0	0	485	0
world	0	518	0	0	518	1	0	518	0	0	518	0
years	0	528	0	0	528	0	6	522	0	6	522	0

E.2.2 Irrelevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
acting	0	0	0	0	0	523	0	0	0	0	0	0
amendment	0	0	1	0	0	1	0	0	0	1	0	0
amendments	0	0	1	0	0	1	0	0	0	1	0	0
bill	12	0	0	0	12	479	12	0	0	0	0	0

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
budget	0	0	0	0	0	1	0	0	0	0	0	0
business	0	0	0	0	0	11	0	0	0	0	0	0
canadian	0	0	0	0	0	37	0	0	0	0	0	0
children	0	0	0	0	0	24	0	0	0	0	0	0
committee	0	0	0	0	0	9	0	0	0	0	0	0
commons	0	0	17	0	0	156	0	0	0	17	0	0
community	0	0	448	0	0	0	0	0	0	448	0	0
country	0	0	0	0	0	41	0	0	0	0	0	0
court	0	0	0	0	0	199	0	0	0	0	0	0
deal	0	0	0	0	0	16	0	0	0	0	0	0
decision	0	0	0	0	0	1	0	0	0	0	0	0
department	0	78	0	78	0	0	78	0	0	78	0	0
deputy	0	0	0	0	0	1	0	0	0	0	0	0
development	0	0	0	0	0	49	0	0	0	0	0	0
farmers	0	0	0	0	0	11	0	0	0	0	0	0
government	0	0	0	0	0	6	0	0	0	0	0	0
health	0	0	0	0	0	1	0	0	0	0	0	0
honourable	0	0	0	0	0	7	0	0	0	0	0	0
human	0	0	93	0	0	94	0	0	0	93	0	0
income	0	0	0	0	0	233	0	0	0	0	0	0
industry	0	0	0	0	0	1	0	0	0	0	0	0
information	0	0	39	0	0	0	0	0	0	39	0	0
international	0	0	364	0	0	20	0	0	0	364	0	0
issue	0	0	0	0	0	48	0	0	0	0	0	0
justice	0	0	0	0	0	3	0	0	0	0	0	0
law	0	0	0	0	0	306	0	0	0	0	0	0
leader	0	0	0	0	0	1	0	0	0	0	0	0
liberal	0	0	83	0	0	3	0	0	0	83	0	0
members	1	0	0	1	0	4	1	0	0	0	0	0
motion	1	0	0	0	1	1	1	0	0	0	0	0
national	0	0	0	0	0	1	0	0	0	0	0	0
opportunity	0	0	30	0	0	0	0	0	0	30	0	0
order	0	0	0	0	0	8	0	0	0	0	0	0
parliament	0	0	1	0	0	6	0	0	0	1	0	0
parliamentary	0	0	0	0	0	6	0	0	0	0	0	0
party	0	1	2	1	0	0	1	0	0	3	0	0
pay	0	0	0	0	0	302	0	0	0	0	0	0
people	0	0	326	0	0	326	0	0	0	326	0	0
policy	0	4	0	0	4	417	4	0	0	4	0	0
political	0	0	73	0	0	22	0	0	0	73	0	0
position	0	0	457	0	0	0	0	0	0	457	0	0
prime	0	0	0	0	0	4	0	0	0	0	0	0
private	0	0	6	0	0	6	0	0	0	6	0	0
process	0	0	4	0	0	4	0	0	0	4	0	0
provide	0	0	21	0	0	22	0	0	0	21	0	0
province	0	0	0	0	0	2	0	0	0	0	0	0
provinces	0	0	0	0	0	2	0	0	0	0	0	0
provincial	0	0	0	0	0	2	0	0	0	0	0	0
public	0	0	284	0	0	284	0	0	0	284	0	0
reform	0	0	0	0	0	11	0	0	0	0	0	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
resources	0	25	7	3	22	7	25	0	0	32	0	0
respect	0	0	254	0	0	254	0	0	0	254	0	0
rights	245	12	0	0	257	1	257	0	0	12	0	0
senate	0	0	0	0	0	4	0	0	0	0	0	0
senator	0	0	0	0	0	2	0	0	0	0	0	0
senators	0	0	0	0	0	4	0	0	0	0	0	0
services	0	1	0	0	1	0	1	0	0	1	0	0
speaker	0	0	0	0	0	16	0	0	0	0	0	0
states	0	0	0	0	0	19	0	0	0	0	0	0
system	0	0	97	0	0	240	0	0	0	97	0	0
tax	0	0	0	0	0	38	0	0	0	0	0	0
taxes	0	0	0	0	0	39	0	0	0	0	0	0
trade	0	0	0	0	0	475	0	0	0	0	0	0
united	0	0	0	0	0	3	0	0	0	0	0	0
world	0	0	0	0	0	7	0	0	0	0	0	0
years	0	0	0	0	0	14	0	0	0	0	0	0

E.3 Differential recall results for French randomly selected words

E.3.1 Relevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
aboutissement	0	344	0	0	344	0	257	87	0	257	87	0
abstentionnisme	0	0	0	0	0	191	0	0	0	0	0	0
adonneriez	31	73	0	0	104	0	104	0	0	73	0	0
adressait	4	514	0	0	518	0	455	63	0	451	63	0
aguichée	0	3	0	0	3	0	3	0	0	3	0	0
aliénerait	0	120	0	1	119	0	120	0	0	120	0	0
allume	18	45	0	0	63	3	53	10	0	35	10	0
amaigrissait	0	8	0	0	8	0	8	0	0	8	0	0
amollissant	0	1	0	0	1	0	1	0	0	1	0	0
amoncellerait	1	3	0	1	3	0	4	0	0	3	0	0
approuverai	2	484	0	0	486	0	485	1	0	483	1	0
automatisasse	0	43	0	0	43	11	43	0	0	43	0	0
avertissement	0	251	0	0	251	0	145	106	0	145	106	0
ballaste	0	5	0	0	5	0	5	0	0	5	0	0
brasserai	7	33	0	0	40	14	40	0	0	33	0	0
carburerai	0	167	0	0	167	0	167	0	0	167	0	0
clamer	5	47	0	0	52	6	32	20	0	27	20	0
cloisonnement	0	10	0	0	10	0	8	2	0	8	2	0

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
comestible	0	8	0	0	8	0	6	2	0	6	2	0
comparaîtrais	168	239	0	103	304	0	407	0	0	239	0	0
compromettrai	210	164	0	0	374	0	373	1	0	163	1	0
conduite	26	451	0	2	475	0	84	393	0	58	393	0
congratule	1	13	0	0	14	0	11	3	0	10	3	0
contravention	264	59	0	264	59	0	285	38	0	21	38	0
contribué	0	520	0	0	520	0	110	410	0	110	410	0
convoitais	1	25	0	0	26	6	26	0	0	25	0	0
copropriétaire	0	6	0	0	6	0	2	4	0	2	4	0
couteau	0	54	0	0	54	0	14	40	0	14	40	0
crampe	0	2	0	0	2	0	2	0	0	2	0	0
dilatante	0	2	0	0	2	70	2	0	0	2	0	0
discorderai	0	61	0	0	61	0	61	0	0	61	0	0
décontractais	0	5	0	0	5	0	5	0	0	5	0	0
défrayerai	4	43	0	6	41	0	47	0	0	43	0	0
dégouliné	0	1	0	0	1	0	1	0	0	1	0	0
déjeuner	10	92	0	0	102	0	17	85	0	7	85	0
déportais	0	52	0	0	52	0	52	0	0	52	0	0
désacraliserai	0	1	0	0	1	0	1	0	0	1	0	0
détroit	0	34	0	0	34	0	0	34	0	0	34	0
détromperai	0	4	0	0	4	0	4	0	0	4	0	0
embourberaient	1	25	0	0	26	0	26	0	0	25	0	0
employer	0	501	17	0	501	19	269	232	0	286	232	0
enclencheront	1	44	0	0	45	0	45	0	0	44	0	0
enlacerais	0	1	0	0	1	0	1	0	0	1	0	0
escrimions	0	1	0	0	1	0	1	0	0	1	0	0
excisai	0	4	0	0	4	0	4	0	0	4	0	0
explicitement	2	225	0	0	227	0	117	110	0	115	110	0
exécuteur	371	7	0	0	378	70	372	6	0	1	6	0
frapper	6	409	0	0	415	0	315	100	0	309	100	0
frapperai	6	409	0	0	415	0	415	0	0	409	0	0
germe	0	27	0	0	27	5	22	5	0	22	5	0
grondement	0	6	0	0	6	0	6	0	0	6	0	0
habilla	2	57	0	0	59	0	59	0	0	57	0	0
hardie	0	8	2	0	8	2	4	4	0	6	4	0
homologuer	8	343	0	0	351	0	347	4	0	339	4	0
imperfectif	0	0	55	0	0	55	0	0	0	55	0	0
impromptue	0	6	0	0	6	0	5	1	0	5	1	0
incompétente	0	51	80	0	51	80	40	11	0	120	11	0
incurvât	0	1	0	0	1	0	1	0	0	1	0	0
inscrivant	424	88	0	0	512	0	474	38	0	50	38	0
introspectif	0	4	6	0	4	6	1	3	0	7	3	0
inventeraient	3	203	10	0	206	12	206	0	0	213	0	0
irritabilité	0	2	94	0	2	4	0	2	0	94	2	0
itérée	0	0	1	0	0	0	0	0	0	1	0	0
jargonnais	0	56	0	0	56	0	56	0	0	56	0	0
jutais	0	1	0	0	1	9	1	0	0	1	0	0
lapider	1	5	0	0	6	6	5	1	0	4	1	0
mannequinerai	0	0	5	0	0	5	0	0	0	5	0	0
mesurais	0	523	0	0	523	0	523	0	0	523	0	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
mobilisai	7	139	2	0	146	2	146	0	0	141	0	0
naïve	49	33	0	0	82	14	64	18	0	15	18	0
orange	0	37	0	0	37	0	20	17	0	20	17	0
orbiter	0	12	0	0	12	1	11	1	0	11	1	0
otage	0	92	0	0	92	0	32	60	0	32	60	0
particularisation	0	1	0	0	1	519	0	1	0	0	1	0
pauvresse	358	0	0	0	358	76	358	0	0	0	0	0
pavement	5	43	0	29	19	0	48	0	0	43	0	0
pendulions	0	34	0	0	34	0	34	0	0	34	0	0
plombai	1	3	28	0	4	28	4	0	0	31	0	0
préexisterai	0	2	0	0	2	2	2	0	0	2	0	0
quotidienne	0	391	0	0	391	0	234	157	0	234	157	0
raisiné	0	0	9	0	0	9	0	0	0	9	0	0
recommandais	0	512	0	0	512	0	510	2	0	510	2	0
refoulions	0	27	0	0	27	0	27	0	0	27	0	0
remilitarisation	0	1	0	0	1	0	0	1	0	0	1	0
rédigeais	4	420	0	0	424	0	422	2	0	418	2	0
révère	0	2	0	0	2	10	2	0	0	2	0	0
signal	0	416	97	0	416	98	312	104	0	409	104	0
sonnerai	0	529	0	317	212	0	529	0	0	529	0	0
splendeur	0	19	0	0	19	0	5	14	0	5	14	0
surencombrée	0	1	0	0	1	0	1	0	0	1	0	0
surtaxe	0	117	0	18	99	0	32	85	0	32	85	0
villégiaturer	0	16	0	0	16	0	16	0	0	16	0	0
ébattissent	2	0	0	2	0	0	2	0	0	0	0	0
écopée	8	55	0	0	63	0	63	0	0	55	0	0
écraserais	6	224	0	84	146	0	230	0	0	224	0	0
écrirai	286	200	0	0	486	4	483	3	0	197	3	0
éditorial	0	120	41	0	120	41	30	90	0	71	90	0
éperdez	0	0	0	0	0	22	0	0	0	0	0	0
étalagerais	0	22	0	0	22	0	22	0	0	22	0	0
évapora	2	13	1	0	15	1	15	0	0	14	0	0

E.3.2 Irrelevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
adressait	0	0	0	0	0	1	0	0	0	0	0	0
aliénerait	0	0	1	0	0	0	0	0	0	1	0	0
brasserai	0	0	1	0	0	35	0	0	0	1	0	0
carburerai	0	0	0	0	0	1	0	0	0	0	0	0
clamer	0	0	0	0	0	1	0	0	0	0	0	0
comparaîtrais	0	0	0	0	0	377	0	0	0	0	0	0
contribué	0	0	1	0	0	1	0	0	0	1	0	0
crampe	2	0	0	0	2	8	2	0	0	0	0	0
exécuteur	0	0	0	0	0	2	0	0	0	0	0	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
frapper	0	0	0	0	0	2	0	0	0	0	0	0
frapperai	0	0	0	0	0	2	0	0	0	0	0	0
germe	0	0	0	0	0	75	0	0	0	0	0	0
hardie	0	0	18	0	0	3	0	0	0	18	0	0
inventeraient	0	0	0	0	0	45	0	0	0	0	0	0
jutais	0	0	0	0	0	3	0	0	0	0	0	0
orange	0	0	5	0	0	6	0	0	0	5	0	0
otage	1	0	0	1	0	0	1	0	0	0	0	0
particularisation	0	0	0	0	0	23	0	0	0	0	0	0
pavement	0	0	0	0	0	4	0	0	0	0	0	0
plombai	0	0	0	0	0	26	0	0	0	0	0	0
quotidienne	0	0	0	0	0	1	0	0	0	0	0	0
raisiné	0	0	0	0	0	2	0	0	0	0	0	0
révère	0	0	0	0	0	51	0	0	0	0	0	0
signal	0	0	0	0	0	412	0	0	0	0	0	0
sonnerai	0	0	0	0	0	4	0	0	0	0	0	0
écraserais	0	0	0	0	0	1	0	0	0	0	0	0
écrirai	0	0	0	0	0	13	0	0	0	0	0	0

E.4 Differential recall results for French frequency selected words

E.4.1 Relevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
accord	0	526	4	0	526	4	0	526	0	4	526	0
affaires	0	526	0	0	526	0	0	526	0	0	526	0
années	0	523	0	0	523	0	3	520	0	3	520	0
argent	0	492	0	0	492	0	0	492	0	0	492	0
article	0	518	0	0	518	0	1	517	0	1	517	0
assurance	0	526	0	24	502	0	27	499	0	27	499	0
assurer	0	526	0	8	518	0	10	516	0	10	516	0
autochtones	0	451	0	0	451	0	6	445	0	6	445	0
avis	0	520	7	0	520	7	0	520	0	7	520	0
budget	0	499	0	0	499	0	12	487	0	12	487	0
canadienne	0	528	0	0	528	3	6	522	0	6	522	0
canadiens	0	528	0	0	528	3	6	522	0	6	522	0
chambre	0	530	0	0	530	0	0	530	0	0	530	0
collègue	0	521	0	0	521	0	13	508	0	13	508	0
comité	0	524	0	0	524	0	0	524	0	0	524	0
commission	1	483	0	484	0	0	1	483	0	0	483	0

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
communes	0	528	0	1	527	0	1	527	0	1	527	0
compte	0	522	0	0	522	0	1	521	0	1	521	0
concernant	0	526	0	0	526	0	1	525	0	1	525	0
conseil	0	516	1	5	511	0	5	511	0	6	511	0
demande	0	525	0	0	525	0	5	520	0	5	520	0
dollars	0	514	0	0	514	0	0	514	0	0	514	0
débat	0	527	0	0	527	0	3	524	0	3	524	0
décision	3	517	0	0	520	0	9	511	0	6	511	0
député	0	511	0	0	511	0	45	466	0	45	466	0
députés	0	511	0	0	511	0	11	500	0	11	500	0
développement	0	513	0	3	510	0	7	506	0	7	506	0
emploi	2	504	14	0	506	1	17	489	0	29	489	0
enfants	1	511	0	1	511	0	1	511	0	0	511	0
entreprises	14	503	0	14	503	0	31	486	0	17	486	0
finances	0	515	0	1	514	0	26	489	0	26	489	0
fonds	1	518	0	0	519	0	11	508	0	10	508	0
fédéral	0	518	0	0	518	0	5	513	0	5	513	0
gens	0	520	0	0	520	0	0	520	0	0	520	0
gouvernement	0	528	0	0	528	5	0	528	0	0	528	0
honorable	0	529	0	3	526	0	3	526	0	3	526	0
honorables	0	529	0	3	526	0	250	279	0	250	279	0
impôts	0	471	0	0	471	0	42	429	0	42	429	0
industrie	0	491	0	0	491	6	8	483	0	8	483	0
jeunes	3	497	0	0	500	0	15	485	0	12	485	0
justice	0	500	0	0	500	0	0	500	0	0	500	0
leader	2	519	0	0	521	3	2	519	0	0	519	0
libéral	0	496	0	0	496	1	22	474	0	22	474	0
libéraux	0	496	0	0	496	1	54	442	0	54	442	0
matière	0	531	0	0	531	0	11	520	0	11	520	0
membres	0	520	0	0	520	0	0	520	0	0	520	0
mesure	0	523	0	0	523	0	5	518	0	5	518	0
mesures	0	523	0	0	523	0	4	519	0	4	519	0
ministre	0	523	0	0	523	0	0	523	0	0	523	0
ministère	0	510	0	0	510	0	7	503	0	7	503	0
monde	0	519	0	0	519	0	0	519	0	0	519	0
monsieur	1	436	0	1	436	0	1	436	0	0	436	0
motion	0	527	0	0	527	0	0	527	0	0	527	0
nationale	0	523	0	0	523	0	10	513	0	10	513	0
opposition	1	519	0	1	519	0	1	519	0	0	519	0
parlement	0	526	0	4	522	0	7	519	0	7	519	0
parlementaire	0	519	0	0	519	6	22	497	0	22	497	0
parti	0	527	0	3	524	0	32	495	0	32	495	0
pays	0	520	0	0	520	0	0	520	0	0	520	0
personnes	0	525	0	0	525	0	3	522	0	3	522	0
politique	0	523	0	0	523	0	4	519	0	4	519	0
population	0	509	0	0	509	0	0	509	0	0	509	0
problème	0	521	0	0	521	0	3	518	0	3	518	0
problèmes	0	521	0	0	521	0	11	510	0	11	510	0
processus	0	512	0	0	512	0	0	512	0	0	512	0
programme	0	518	0	0	518	0	5	513	0	5	513	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
programmes	0	518	0	0	518	0	26	492	0	26	492	0
projet	0	529	0	0	529	0	0	529	0	0	529	0
protection	15	501	0	15	501	0	16	500	0	1	500	0
provinces	0	517	0	0	517	0	8	509	0	8	509	0
président	0	533	0	0	533	0	0	533	0	0	533	0
présidente	0	533	0	0	533	0	125	408	0	125	408	0
question	0	526	0	0	526	0	6	520	0	6	520	0
questions	0	526	0	0	526	0	0	526	0	0	526	0
québécois	0	440	0	2	438	62	8	432	0	8	432	0
raison	0	521	0	0	521	0	3	518	0	3	518	0
rapport	0	523	1	0	523	1	0	523	0	1	523	0
ressources	0	505	0	0	505	0	1	504	0	1	504	0
revenu	7	497	0	0	504	0	9	495	0	2	495	0
règlement	0	528	0	86	442	0	3	525	0	3	525	0
réformiste	0	398	91	0	398	0	12	386	0	103	386	0
régime	0	500	0	0	500	0	4	496	0	4	496	0
réponse	0	520	0	0	520	0	2	518	0	2	518	0
santé	0	513	0	0	513	0	0	513	0	0	513	0
secrétaire	0	460	0	0	460	0	2	458	0	2	458	0
secteur	3	509	0	3	509	0	10	502	0	7	502	0
services	2	522	0	2	522	0	9	515	0	7	515	0
situation	0	522	0	0	522	0	0	522	0	0	522	0
société	0	518	0	0	518	0	3	515	0	3	515	0
système	0	515	0	0	515	0	2	513	0	2	513	0
sécurité	0	512	0	0	512	0	0	512	0	0	512	0
sénat	0	472	0	0	472	14	1	471	0	1	471	0
sénateur	4	424	0	4	424	0	79	349	0	75	349	0
sénateurs	4	424	0	4	424	0	41	387	0	37	387	0
temps	0	521	0	0	521	0	0	521	0	0	521	0
travail	5	520	0	0	525	1	5	520	0	0	520	0
voix	0	521	0	0	521	0	0	521	0	0	521	0
vote	2	499	0	2	499	0	26	475	0	24	475	0
égard	0	520	0	0	520	0	0	520	0	0	520	0
étude	0	521	1	0	521	0	1	520	0	2	520	0

E.4.2 Irrelevance criterion

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
accord	0	0	0	0	0	1	0	0	0	0	0	0
années	278	0	0	277	1	0	278	0	0	0	0	0
argent	0	0	0	0	0	141	0	0	0	0	0	0
assurer	0	0	0	0	0	1	0	0	0	0	0	0
avis	0	0	0	0	0	44	0	0	0	0	0	0
canadienne	0	0	0	0	0	80	0	0	0	0	0	0
canadiens	0	0	0	0	0	80	0	0	0	0	0	0

	LM vs. S			LM vs. W			LM vs. EQ			S vs. EQ		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
chambre	0	0	0	0	0	1	0	0	0	0	0	0
collègue	0	0	0	0	0	2	0	0	0	0	0	0
comité	0	1	0	1	0	6	1	0	0	1	0	0
commission	0	0	0	0	0	1	0	0	0	0	0	0
communes	1	10	429	11	0	2	11	0	0	439	0	0
compte	0	0	0	0	0	2	0	0	0	0	0	0
dollars	0	0	0	0	0	88	0	0	0	0	0	0
débat	0	0	0	0	0	1	0	0	0	0	0	0
décision	0	0	0	0	0	79	0	0	0	0	0	0
député	0	0	2	0	0	0	0	0	0	2	0	0
députés	0	0	2	0	0	0	0	0	0	2	0	0
enfants	43	0	0	43	0	0	43	0	0	0	0	0
finances	110	0	0	110	0	0	110	0	0	0	0	0
fonds	0	0	499	0	0	518	0	0	0	499	0	0
fédéral	0	0	0	0	0	1	0	0	0	0	0	0
gens	0	0	190	0	0	0	0	0	0	190	0	0
gouvernement	0	0	0	0	0	3	0	0	0	0	0	0
honorable	0	0	5	0	0	62	0	0	0	5	0	0
honorables	0	0	5	0	0	62	0	0	0	5	0	0
impôts	0	0	0	0	0	4	0	0	0	0	0	0
industrie	0	0	1	0	0	0	0	0	0	1	0	0
jeunes	0	1	0	1	0	1	1	0	0	1	0	0
libéral	0	0	0	0	0	227	0	0	0	0	0	0
libéraux	0	0	0	0	0	227	0	0	0	0	0	0
ministre	0	0	1	0	0	8	0	0	0	1	0	0
ministère	0	0	0	0	0	1	0	0	0	0	0	0
motion	0	0	0	0	0	1	0	0	0	0	0	0
nationale	0	0	0	0	0	3	0	0	0	0	0	0
parlement	0	0	9	0	0	503	0	0	0	9	0	0
parlementaire	0	0	0	0	0	3	0	0	0	0	0	0
parti	0	0	120	0	0	523	0	0	0	120	0	0
pays	0	121	0	121	0	0	121	0	0	121	0	0
personnes	0	0	1	0	0	514	0	0	0	1	0	0
politique	0	0	2	0	0	1	0	0	0	2	0	0
population	0	0	24	0	0	0	0	0	0	24	0	0
problème	0	0	0	0	0	1	0	0	0	0	0	0
problèmes	0	0	0	0	0	1	0	0	0	0	0	0
projet	0	49	99	0	49	148	49	0	0	148	0	0
provinces	0	0	0	0	0	1	0	0	0	0	0	0
président	0	0	0	0	0	1	0	0	0	0	0	0
présidente	0	0	0	0	0	1	0	0	0	0	0	0
question	0	0	0	0	0	1	0	0	0	0	0	0
questions	0	0	0	0	0	1	0	0	0	0	0	0
raison	0	0	0	0	0	2	0	0	0	0	0	0
rapport	0	0	0	0	0	4	0	0	0	0	0	0
réformiste	0	0	0	0	0	2	0	0	0	0	0	0
régime	71	0	0	0	71	1	71	0	0	0	0	0
réponse	0	0	0	0	0	1	0	0	0	0	0	0
santé	0	8	2	8	0	0	8	0	0	10	0	0
services	0	0	0	0	0	1	0	0	0	0	0	0

	LM vs. S			**LM vs. W**			**LM vs. EQ**			**S vs. EQ**		
	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B	Δ_B^A	$\cap_B^A$	Δ_A^B
société	0	0	1	0	0	0	0	0	0	1	0	0
système	0	0	72	0	0	2	0	0	0	72	0	0
sécurité	0	0	1	0	0	0	0	0	0	1	0	0
sénat	0	0	0	0	0	387	0	0	0	0	0	0
sénateur	0	0	0	0	0	21	0	0	0	0	0	0
sénateurs	0	0	0	0	0	21	0	0	0	0	0	0
temps	0	0	7	0	0	0	0	0	0	7	0	0
travail	0	0	0	0	0	3	0	0	0	0	0	0
vote	6	0	0	6	0	3	6	0	0	0	0	0

APPENDIX F

TREC queries and results

Precision and recall measures for different query pre-processing techniques are obtained using TREC document collection and relevance judgements to topics. The collection used is TIPSTER Volume 1 & 2 [35], the queries are topics 51 to 200 [19]. The following sections present sample queries used to retrieve relevant documents and aggregate relevance and precision results along with a presentation of how the measures are computed (see Chapter 6). We use the following abbreviation: LM (Lightweight Morphology), S (Stemming) and EQ (Exact Query).

F.1 Set 1 queries

The first set of queries was obtained by straightforwardly selecting the 'title' of each topic to be a query. We manually preprocessed the queries to remove all articles and meaningless words and included expanded acronyms (e.g. *U.S.* into *United States*).

F.1.1 Sample of queries

```
51 Airbus Subsidies

52 South African Sanctions

53 Leveraged Buyouts

54 Satellite Launch Contracts

55 Insider Trading
```

F.1.2 Results

Table F.1: TREC average interpolated precision-recall $\overline{P_I(r)}$ for topics 51 to 200 on set 1 queries. Results for L, S and EQ.

Recall level r	LM	S	EQ
0.00	0.3047	0.3067	0.3132
0.10	0.1732	0.1802	0.1832
0.20	0.1401	0.1491	0.1493
0.30	0.1088	0.1192	0.1143
0.40	0.0824	0.0920	0.0867
0.50	0.0641	0.0652	0.0624
0.60	0.0435	0.0419	0.0403
0.70	0.0256	0.0237	0.0207
0.80	0.0162	0.0159	0.0131
0.90	0.0064	0.0064	0.0050
1.00	0.0005	0.0005	0.0007

Table F.2: TREC average precision after d documents retrieved $\overline{P(d)}$ for topics 51 to 200 on set 1 queries. Results for L, S and EQ.

Documents d	LM	S	EQ
5	0.1373	0.1387	0.1573
10	0.1573	0.1633	0.1580
15	0.1653	0.1627	0.1573
20	0.1673	0.1667	0.1620
30	0.1656	0.1673	0.1609
100	0.1430	0.1475	0.1448
200	0.1216	0.1270	0.1236
500	0.0933	0.0956	0.0890
1000	0.0682	0.0694	0.0627

F.2 Set 2 queries

The second set of queries by selecting for each topic the following entries: ‘domain’, ‘title’, ‘description’, ‘summary’ and ‘concepts’. We processed the text to remove common words with a stop list [34] and to remove punctuation symbols. Each resulting query is a list of unique and meaningful words.

F.2.1 Sample of queries

```
51 international economics airbus subsidies document discuss
   government assistance industrie mention trade dispute u.s.
   aircraft producer issue relevant cite french german british
   spanish european governments boeing co. mcdonnell douglas corp.
   federal consortium messerschmitt boelkow blohm gmbh aerospace plc
   aerospatiale construcciones aeronauticas s.a. aid loan financing
   controversy tension general agreement tariffs gatt code policy
   review group tprg complaint objection retaliation anti dumping
   duty petition countervailing sanctions
52 international economics south african sanctions document discusses
   africa relevant discuss aspect declared proposed country
   government response apartheid policy pressure individual
   organization pretoria imposed united nations effects opposition
   compliance company identify instituted considered corporate
   disinvestment trade ban academic boycott arms embargo economic
   exodus stock divestiture investment import diamonds u.n.
   curtailment defense contracts cutoff nonmilitary goods reduction
   cultural ties white domination racism antiapartheid black majority
    rule
```

F.2.2 Results

Table F.3: TREC average interpolated precision-recall $\overline{P_I(r)}$ for topics 51 to 200 on set 2 queries. Results for L, S and EQ.

Recall level r	LM	S	EQ
0.00	0.3546	0.3784	0.4209
0.10	0.2001	0.1946	0.2193
0.20	0.1544	0.1460	0.1680
0.30	0.1184	0.1107	0.1297
0.40	0.0870	0.0823	0.0998
0.50	0.0669	0.0586	0.0649
0.60	0.0370	0.0338	0.0440
0.70	0.0210	0.0157	0.0263
0.80	0.0087	0.0093	0.0129
0.90	0.0018	0.0000	0.0019
1.00	0.0000	0.0000	0.0000

Table F.4: TREC average precision after d documents retrieved $\overline{P(d)}$ for topics 51 to 200 on set 2 queries. Results for L, S and EQ.

Documents d	LM	S	EQ
5	0.1947	0.1933	0.2333
10	0.2067	0.2060	0.2407
15	0.2040	0.2031	0.2356
20	0.2073	0.2030	0.2330
30	0.2004	0.1938	0.2271
100	0.1705	0.1655	0.1843
200	0.1455	0.1418	0.1550
500	0.1071	0.1060	0.1111
1000	0.0776	0.0766	0.0805

F.3 TREC recall and precision measures

TREC defines implicit and explicit cutoffs to compute recall and precision. We assume the following notation for a query: *Rel* is a partial ranked list of documents documents retrieved by the IR system, stored in an array and $Rel[i]$ is 1 if the i^{th} document is judged as relevant, 0 if not. M is the number of elements of *Rel*, and can be lower than the number of documents retrieved by the IR system. L is the number of relevant document for the query (as judged by the TREC organizers), and NQ is the total number of queries. The `trec_eval` program computes the following measures for each query:

$P_I(r)$ — **Interpolated recall-precision at level *r*** The precision defined as $P_I(r) = \max_{R(x)\geq r}(P(x))$. x is the number of relevant documents returned by the IR system.

$\overline{P}$ — **Average precision** The precision is calculated after each relevant document is retrieved and averaged over the number of relevant documents retrieved.

$P(d)$ — **Precision after d documents have been retrieved** The precision with a cutoff at d documents.

$P_R(d)$ — **R-precision** The precision with a cutoff at d documents, $d \leq \min(L, M)$.

F.3.1 Algorithm for determining $P_I(r)$

```
**Input:** $Rel$, $r$, $L$, $M$
$P \leftarrow 0$ /* $P$ is the precision calculated with the first $i$ documents */
$R \leftarrow 0$ /* $R$ is the recall calculated with the first $i$ documents */
$Rcount \leftarrow 0$ /* Number of relevant documents found so far */
$maxP \leftarrow 0$ /* Contains current $\max_{R(x) \leq r}(P(x))$ */

**for** $i = 0$ to $M$ **do**
  **if** $Rel[i] = 1$ **then**
    $Rcount \leftarrow Rcount + 1$
  **end if**
  $P \leftarrow \frac{Rcount}{i+1}$
  $R \leftarrow \frac{Rcount}{L}$
  **if** $P > maxP$ **and** $R \leq r$ **then**
    $maxP \leftarrow P$
  **end if**
**end for**
**Return:** $maxP$ /* $P_I(r)$ */
```

$\overline{P_I(r)}$ is calculated as $\frac{\sum_{j=1}^{NQ} P_I^j(r)}{NQ}$, where $P_I^j(r)$ is the interpolated recall-precision at level r calculated for the j^{th} query.

F.3.2 Algorithm for determining $\overline{P}$

```
Input: $Rel$, $M$
$P \leftarrow 0$ /* $P$ is the sum of precision calculated each time a new relevant document is found in $Rel$ */
$Rcount \leftarrow 0$ /* Number of relevant documents found so far */

for $i = 0$ to $M$ do
  if $Rel[i] = 1$ then
    $Rcount \leftarrow Rcount + 1$
    $P \leftarrow P + \frac{Rcount}{i+1}$
  end if
end for
Return: $\frac{P}{Rcount}$ /* $\overline{P}$ */
```

$\overline{\overline{P}}$ is calculated as $\frac{\sum_{j=1}^{NQ} \overline{P^j}}{NQ}$ where $\overline{P^j}$ is the average precision calculated for the j^{th} query.

F.3.3 Algorithm for determining $P(d)$ and $P_R(d)$

Input: Rel, M, d

```
$P \leftarrow 0$ /* $P$ is the precision calculated at a document cutoff $d$ */
$Rcount \leftarrow 0$ /* Number of relevant documents found so far */

for $i = 0$ to $\min(d, M)$ do
    if $Rel[i] = 1$ then
        $Rcount \leftarrow Rcount + 1$
    end if
end for
$P \leftarrow \frac{Rcount}{\min(M, d)}$
Return: $P$ /* $P(d)$ */
```

$\overline{P(d)}$ is calculated as $\frac{\sum_{j=1}^{NQ} P^j(d)}{NQ}$, where $\overline{P^j(d)}$ is the precision after d documents have been retrieved, calculated for the j^{th} query. $P_R(d)$ is a special case of $P(d)$ for $d = min(L, M)$.

VITA

Candidate's full name:	Mikaël Pierre Arthur Roussillon
Place and date of birth:	Paris XX^{e}, France August 16, 1981
Permanent address:	21, rue Chanzy 45000 Orléans France
Universities:	2001 - 2005 École nationale supérieure des Mines Saint-Étienne, France 2003 - 2005 University of New Brunswick Fredericton, Canada
Publications:	M. Roussillon, B. G. Nickerson, and A. E. Maclachlan. Lightweight natural language morphology representation with Xerox finite state morphology. Technical Report TR04-166, University of New Brunswick, 2004. M. Roussillon, B. G. Nickerson, S. Green and W. A. Woods. Lightweight Morphology: A Methodology for Improving Text Search, In *Proceedings of the Third Annual Canadian Symposium on Text Analysis (CaSTA)*, McMaster University, Hamilton, Ontario, Canada, November 19-21, 2004, pp. 99–105.