A text critiquing system for Swedish-speaking students of French

Fabian Kostadinov
Department of Informatics
University of Zürich
fkostadinov@gmx.ch

Jonas Thulin
Department of Computer Science
Lund University
jonasthulin@hotmail.com

Abstract

Making objective, quantitative linguistic analyses is a time-consuming and demanding task. Therefore, we have developed a language analysis system, which can be used for counting occurences of any given pattern. Rule trees and external dictionaries are supported. The program is written in 100 % Java 2 and Swing. Its strength is that rules can be easily added or modified and the main weakness is that it cannot assess language as well as a human can, since only quantitative criteria, and not semantic ones, are taken into account.

1 Introduction

Ever since people have been learning new languages besides their mother tongue, it ha been a question of interest of how to approach the target most efficiently. A number of different approaches and recommendations has been developped by linguists for support purposes. Interested in how to define reliable measures, which empower any teacher to state the language level of his pupils, some linguists over time developped such measures. They found out that the active and passive ability of speaking a language by a student, is reflected astonishingly well by certain grammatical indicators (besides, of course, the richness and variety of the student's vocabulary). So they generated rule sheets that can be applied to a text written by a student and return an indicator number, stating the student's level. Rules may include for example...

- how many times certain advanced verb tenses are being used throughout the text,
- how often the same substantial grammatical mistakes, as using a pronoun followed by a wrong verb form, are repeated,
- the average sentence length,

and much more.

However, the task of manually applying these rules to a text is time consuming and requires a profound understanding of grammatics, inflections and text analysis in general. As in today's world where one teacher often has to supervise several dozen students at the same time, such analysis cannot be done regularly; it is simply too demanding in terms of time and knowledge.

With the program we have written, we tried to develop a prototype of an analysis tool, that should be able to commit linguistic analysis based on a rule sheet automatically. We believe that there is a real need for such a program since it would allow both the student and the teacher to keep track of the student's changing language level in an easy and uncomplicated manner. The aim of the program is to provide one with an easily understandable single measure on a scale, but also showing the mistakes or certain grammatial constructs in the text.

2 Language learning and linguistic development

Both children learning their native language and adults learning a foreign language acquire the language step by step (Clahsen, 1986), (Schlyter, 2003), starting with nouns and simple phrases, advancing via using simple everyday langauge to (in some cases) using the complex language found in e.g. magazine and journal articles.

However, there is one substantial difference between native- and foreign-language acquisition: in native-language learning, there is no other "deeply-rooted" language with which the new language could interfere, whereas there is in foreign-language learning. This interference makes it difficult for Swedish-speaking FLE[1] learners to grasp e.g. word order, elision, complex relatives (e.g. *auxquels*) incflecting verbs after person and verb forms not present in the Swedish, e.g. *futur simple*, *conditionnel*[2], *gérondif* and *subjonctif*[3] These parts of the French language are (in general) only partially comprehended by casual (non-academic or non-professional) users. Of course, this applies to Swedish-speaking persons, who have not grown up in a French-speaking environment.

Below the levels are roughly described:

2.1 Novice (level 1–2, CEF (*C*ommon *E*uropean *F*ramework) A1–A2)

Novice users having only "survival-level" language skills tend to use mostly noun phrases in their communication, and do not yet inflect verbs fluently. Also, negations are usually of the type `<negator> <noun phrase>`. Novice-level students also need a co-operative and patient interlocutor who does not mind the student having limited fluency and vocabulary as well as needing to ask aid questions. An example of novice-level language production is *non, pas en train, en autobus*.

[1] French as a foreign laguage

[2] In Swedish, the conditional is (usually) constructed with the modal verb *skulle* + infinitive; consider it analogous with the English would+inf.

[3] That mode is certainly not used in the same way in Swedish (or German) as in French.

2.2 Intermediate-level user (level 3–4, CEF B1–B2

Intermediate-level users possess a basic command of the standard langauge, and have adequate ability in coping with day-to-day communication. About 50 % of their utterances consist of full sentences, and verbs are inflected and negated, but not always correctly. Swedes at this level often forget that e.g. *au* and *des* are maps of *à le* and *de les*, respectively, and therefore frequently use the tautological construct *au l'* or forget mapping *de les* to *des*. An intermediate-level user would say something like *Non, je ne suis pas venu en train, mais en autobus.*

2.3 Advanced user (level 5–6, CEF C1–C2)

After several years of French studies or stay in a French-speaking environment one can have reached the advanced level. This level is characterized by full acquisition of typical French-language constructs and productive use of a significant amount idiomatic expressions and fixed phrases (approaching native-speaker quality). Grammatical (syntactic) mistakes are rare and even complex language, like the one being used in this report, is (at least partially) mastered. Level 5+ example: *Si je dis quelque chose auquelle je ne suis pas tout à fait sûr, ce sera la composistion des rapports comme celui-ci. En réfléchissant, je pense que traduire ce rapport en français serait un bon exercise pour le lecteur qui se croit vraiment maîtriser le français. C'est improbable que personne qui lise ceci ne puisse le faire,* with any mistakes corrected.

3 The program

One of the major goals of our program was the separation of the above mentioned linguistic rules from the program logic itself. If we are able to separate language specific linguistic rules for analysis from the rest of the program, we are also able to reuse the same program for different languages. Of course, rule sheets must exist for every language to be checked against by our program. This part has to be done by linguists. The program itself should be able to load a new text, analyze it using the rules and show the results during runtime, and

not during compile time as far as possible. We do not want a poor linguist having to learn the depths of programming languages before being able to translate his/her rules to a format, which the program is able to handle.

We therefore chose the growing standard XML to encode the rules. Further, we need a dictionary to look up words in the foreign language to obtain some grammatical information about every word. In order to analyze a text we had to split it up in words and sentences first. We decided to use Java's (version 1.4) ability to work with regular expressions. Although Perl and Prolog at the moment still tend to be the standard languages, their not providing one with integrated features for creating user interfaces like Java's Swing classes do is a severe disadvantage. It would certainly be possible to use e.g. Prolog for the parser, but since our program is likely to be used mostly on Windows machines target and non-programmers would prefer not having to install Prolog just for our program, we decided to do it entirely in Java.

The program on a high-level user perspective is relatively simple. Fist, the user chooses a file to load into an editor pane (or types or pastes[4] it directly into the editor), presses the "Analyze" button and then receives back another window, where grammatical mistakes are marked up and his/her language level is shown on a scale[5]

Looking deeper into the program, as soon as the analyze button is pressed by the user, the input text is sent to the core `Analyzer` as a string and the `Analyzer`begins its work. If it has not done it before, it loads the XML-encoded rules, splits up the text into tokens and applies the rules one after another to the tokens. Grammatical information about the words is found in a dictionary. In our case the dictionary is contained in a hash table, but better alternatives as using a database or a letter tree could be thought of.

When the analysis is done, the `Analyzer` returns the text to the GUI part of the program, but annotated with tags, for instance HTML[6]. The GUI then presents the result to the user again.

We will now take a deeper look into the program's different pieces.

4 Implementation concepts

4.1 Implementation language

Our program is implemented in 100 % Java and should therefore run on any platform supporting Java 1.4 and Swing GUIs.

4.2 The GUI

The GUI is a multilingual text editor with support for reading and writing Unicode text file as well as OS native clipboard handling, e.g. copy from MS Word and paste into the editor. After the user has typed, pasted or loaded a text, he/she can press the Analyze button to obtain a version of his/her text with XML tags, added by the `Analyzer` according to the rules specified in the rule file.

4.3 The rule file

The information of how a text has to be parsed by the `Analyzer` is stored in an XML-encoded file. We will call it the "rule file" or "XML rule file". (We assume that the reader has a basic knowledge of the current XML standards, understands roughly the syntax and semantics of XML and knows how XML and a DTD are connected one to the other.)

The strict separation between encoding the rules on the one hand and the `Analyzer`'s program logic on the other hand results in a gain of independency and configurability. The rules themselves may be changed easily without any need of complementary changes to the program logic. Since all the tags, their possible attributes and cardinalities are defined in a DTD file, the user has clear guidelines to follow when writing new rules. An invalid XML rule file will result in the parser throwing an error and writing an error message to the screen because a flag is set to check the XML's validity before parsing.

[4] Pasting is OS native and allows the user to analyze e.g. Word documents by opening them in Word, copying the text, and pasting it into the editor.

[5] We have not yet agreed how to report a preliminary language level using the data from the counters, so the table in (Schlyter, 2003).

[6] In our prototype only the raw XML is shown, but adding a style sheet would enhance the user's experience considerably. However, we consider it out of the scope of an 80-hour project, but may add it in a future version, if it is requested by the users and either of us finds the time.

As its name indicates, a rule file contains a set of different rules, each one encoded with valid XML tags. Each time the `Analyzer` is started, the rule file is read in, parsed by a standard XML DOM parser, transformed to a set of Java objects, which are then stored in a hash table. The `Analyzer` finally works with this hash table.

In the rule file, only valid XML tags, which are defined in the corresponding DTD, are allowed. A rule file may contain comment tags (starting with `<!--` and ending with `-->`, however such comments are simply ignored by the parser.

The rule file starts with the standard XML header. We used W3C-compliant version 1.0 standard XML. Currently, no namespaces are in use, however Xerces-J, our parser, should be able to handle namespaces correctly, but such functionality has not been tested out.

An important issue not to be forgotten in the header tag is choosing an appropriate character set. As we are dealing with French, the character set must at least contain all French special characters, such as *é* or *à* etc., and the rule file itself must be stored in the chosen format. Of course one has also to make sure, especially for unusual character encodings, that the used parser engine is able to handle the chosen character set. If an inappropriate character encoding set is chosen, there is a risk that certain words will never be found in the dictionary or the program will not be able to successfully match certain words in the text to the search rules (see below).

Further, the rule file consists of a set of "counter tags", two special rules and finally a set of the common or core rules.

4.3.1 The counters

The tag `counters` embraces a set of empty tags `counter`. These counters can be specified to be the holders of search information, in other words how many times certain grammatical forms are met by the `Analyzer`.

4.3.2 The Sentence Tokenize Rule

The first special rule is called the `sentenceTokenizeRule`. It looks like:

```
<sentenceTokenizeRule>
<regex>[.;:!?]</regex>
</sentenceTokenizeRule>
```

Each Sentence Tokenize Rule contains exactly one "regex-tag" which encloses a regular expression. The text will be split up into sentences using this regular expression as a delimiter. Therefore one has to be aware of blank characters, spaces and so on. (This is valid for all the tags containing regular expressions in the rule file.)

4.3.3 The Word Tokenize Rule

The second special rule is called the `wordTokenizeRule`. It is nearly the same as the `sentenceTokenizeRule` and looks like:

```
<wordTokenizeRule>
<regex>[a-zA-Z0-9]+</regex>
</wordTokenizeRule>
```

This rule is used for tokenizing a sentence into words.

4.3.4 The common rules

Following the two special rules come the common rules (or core rules). There must always exist at least one rule, which at the same time is the root or initial rule.

Each rule consists of two parts: The `search` and the `action`. A rule is applied by first searching in a sentence for an occurrence of the search criteria and then, dependent on whether a construct in the sentence (e.g. a word which) matches it or not, the `action`'s `found` respectively `notfound` part is executed.

The rule tag contains an attribute id, which is its unique identifier in the XML document ("is of type `ID`"). Additionally, each rule tag must specify a `framesize` value, indicating the number of words that the rule should be applied to. The start position for a rule's word frame to be opened is always the current position of the `Analyzer` (the left index) counting as many words to the right as the frame-size's value specifies or to the end of a sentence.

The Search A `search` can contain three different search criteria or any combination of the three.

- `<regex>apprends?</regex>` searches for regular expressions such as `apprend` or `apprends`

- `<lemma>apprendre</lemma>` searches for every word that has a lemma equal to the string *apprendre*, for instance *apprendrait* or *appris*
- `<inflection category="verb">...</inflection>` is able to search for certain grammatical constructs, here for instance for a verb.

The attribute `category` represents a word category and may take one of the following values: `noun`, `verb`, `adjective`, `pronoun`, `int_pronoun` (interrogative pronoun), `determiner`, `adverb`, `preposition`, `conjunction`, `numeral`[7], `interjection`, `abbreviation`, `residual` (any word that does not fit into one of the other categories).

Depending on the category, different combinations of tags may be added to the inflection tag. The program does not check whether a given combination of tags makes sense or not, it simply searches for this very combination (thus searching for a noun with a third person singular tag will of course never return any result). Basically, every word may have a combination of the following grammatical information:

1. `gender`: Possible values are `feminine` or `masculine`.
2. `number`: Possible values are `sg` (singular) or `pl` (plural).
3. `person`: Possible values are `1`, `2` or `3`.
4. `tense`: Possible values are `future`, `present`, `imperfect` or `past`.
5. `mode`: Possible values are `indicative`, `conditional`, `infinitive`, `participle` or `subjunctive`.

All these have a common empty tag format as `<tense value="present"/>`.

By writing these categories we oriented ourselves not only on purely grammatical and linguistic knowledge but tried to find a compromise with the needs of programming purposes too. Further, be aware of the order! The XML parser does only accept tags appearing in this specified order, thus for any rule you do not specify a gender, tense and mode, you have to assure that the number tag appears above the person tag.

[7] Numerals are rarely used in the dictionary. Our program will treat them as Residuals.

Category	gender	number	person	tense	mode
noun	X	X	-	-	-
verb (ordinary)	-	X	X	X	X
verb (participle)	X	X	X	X	X
adjective	X	X	-	-	-
pronoun	X	X	X	-	-
int_pronoun	X	X	X	-	-
determiner	X	X	-	-	-
adverb	-	-	-	-	-
preposition	-	-	-	-	-
conjunction	-	-	-	-	-
numeral	-	-	-	-	-
interjection	-	-	-	-	-
abbreviation	-	-	-	-	-
residual	-	-	-	-	-

Table 1: Possible tag combinations for given word categories

It is also easy to recognize that in the rule file combinations of these tags can be specified that have no connection to real grammatical word forms at all. The program will not show any error message to the user but simply not return any result. Table 4.3.4 indicates the use of word categories and the corresponding grammatical information that may be specified.

Some comments have to be made:

1. In verbs, the grammatical information to be specified varies heavily. Whereas “normal” verb forms do not have any gender, participles may indeed have a gender. Imagine the feminine plural past participle of the French verb *ouvrir* (= to open) *ouvertes*. To have appropriate knowledge about this word, one needs to be able to specify a gender (feminine), a number (plural), a tense (past) and a mode (participle). On the other hand, of course a word like third person singular present of *rire* (= to laugh) does not have any

gender at all.

2. For many European languages, there are no participles with forms other than present or past.

3. Some pronouns do have a gender such as *he, she, his, her* while others, such as *I, you, we, my, yours*, do not. The same applies to interrogative pronouns.

4. Determiners that have a gender and/or a number are for example the French words *sa, son, ses, seize* (sixteen) and others.

Every word has at least a regular expression that may be searched for, namely the word itself, and a lemma. If one would like to look for a verb without further grammatical information, then at least a category can be specified.

Word categories are a subject of debate through the field of linguistics; often they are simply more or less given by the dictionary one uses.

An example:

```
<rule id="myRule" framesize="max"/>
<search>
<!-- <regex>(ouverte)|(ouvertes)
</regex> -->commented out!!
<lemma>ouvrir</lemma>
<inflection category="verb">
<gender value="feminine"/>
<number value="pl"/>
<tense value="past"/>
<mode value="participle"/>
</inflection>
</search>
```

These criteria will look in a frame of 4 words for the first occurrence of a word that has *ouvrir* as its lemma, is a verb and a feminine plural past participle. In the sentence *Il a fermé toutes les portes ouvertes* the word *ouvertes* will be found, but if the frame size is reduced to only 4, even though starting at the beginning of the sentence, applying the rule once to the sentence will result in nothing being found.

The action part specifies mainly three things:

1. Whether any counter should be incremented,

2. Whether tagging of current frame should be done and which tag should be used,

3. Which rule should be the next one to take

The rule uses `nextrule`'s value, which must point to an existing rule's `id`, to detect which rule to take next. This next rule does not need to be specified as child tag inside `found/notfound`. The only reason for which rules can be specified inside the `found/notfound` tags is the higher readability of the XML file. The program will always identify the next rule to take using the reference that `nextrule` points to, but disregard where in the XML file it is specified. If no next rule is set, then the initial or root rule will be the next one to be applied again.

If tagging is set to `yes` but no tag name is provided, the program will tag the output text with the current rule's `id`.

`action` always specifies what has to be done, if the `search` delivers a result, this is called the found and what has to be done if it does not find any result, simply called the `notfound`.

Such a structure makes it possible to search for multi word constructs or to compare words. For instance, one could specify a `search` that is looking for first person singular *je*, then specify an `action found` to go to another rule that then looks for a verb in a specified frame that also has a first person singular ending. On the other hand, if *je* is not found, then simply the word should be skipped and the second rule should never be called. This can be specified by not providing `notfound` with and next rule.

An example:

```
<action>
<found inccounter="fem_pl_pp_ouvrir"
nextrule="rule2"
dotagging="yes"
tagname="fem_pl_pp">
<rule id="rule2">...</rule>the next rule
</found>
<notfound dotagging="no">
</notfound>
</action>
```

These instructions will, if something is found by the `search`, cause the `Analyzer` to increment

the counter `fem_pl_pp_ouvrir`, tag the frame with the tag `<fem_pl_pp>...</fem_pl_pp>` and then go to `rule2`. If nothing is found, the `Analyzer` simply goes to the next rule, which is not specified here, thus making the initial rule be applied again.

Overview

On the next page is an overview of all the possible tags, their attributes and possible child tags.

Tagname	Attributes	(Direct) child tags
Action		found notfound
Counters		counter*
Counter	id::ID # Required	
Found	inccounter::IDREF # IMPLIED nextrule::IDREF # IMPLIED dotagging::(yes | no) "no" tagname::NMTOKEN # IMPLIED	rule?
Gender	value::(feminine|masculine) # REQUIRED	
inflection	category::(noun|verb|adjective| pronoun|int_pronoun|determiner| adverb|preposition|conjunction| numeral|interjection| tense? abbreviation|residual) # REQUIRED	gender? number? person? mode?
Lemma::# PCDATA		
Mode	value::(indicative|conditional |infinitive|participle| subjunctive) #REQUIRED	
Notfound	inccounter::IDREF # IMPLIED nextrule::IDREF # IMPLIED dotagging::(yes|no) "no" tagname::NMTOKEN # IMPLIED	rule?
Number	value::(sg|pl) # REQUIRED	
Person	value::(1|2|3) # REQUIRED	
Regex::# PCDATA		
Rules		counters? sentenceTokenizeRule wordTokenizeRule rule+
Rule	id::ID # REQUIRED framesize::CDATA # REQUIRED (can take the value "max" for a frame as big a possible, or a numer > 0 alternatively).	search action
Search		regex? lemma? inflection?
SentenceTokenizeRule		regex
Tense	value::(future|present| imperfect|past) # REQUIRED	
WordTokenizeRule		regex

Table 2: All possible tags, their attributes and possible child tags

All tags that do not embrace child tags are by definition empty tags. Exceptions are the two tags `regex` and `lemma`, which simply only embrace `PCDATA`.

This table is to be read as: "There is a tag called `gender` that has exactly one attribute called `value`. A value of `value` is required and must be either `feminine` or `masculine`. The tag `gender` contains no further tags, therefore it is an empty tag."

`# REQUIRED` and `# IMPLIED` are used in their original XML-standard meaning. Child tags followed by a question mark `?` indicate that the current tag may have 0 or 1 of this child tag, a plus `+` that it may have 1 or more child tags of this kind, and an asterisk `*` that it may have 0 or more child tags. If no sign is specified, then exactly 1 child tag has to be added.

4.4 The Analyzer

The `Analyzer` carries out the core functionality of the program. Generally said, it holds a hash table of rules, as specified in the rule file, and simply applies one after the other sequentially to the text. It always starts by applying the initial rule and then following the rule's references (as specified by the `nextrule` attribute of the `found/notfound` tags). Also, as specified by the rule file, it increments counters if needed that may later on be read out and used for further purposes.

An `Analyzer` must implement the `IAnalyzer` interface, which only provides one method:
`public String analyze(String input)` `analyze()` takes the text to be parsed as an input `String` and returns a pseudo HTML-annotated `String` if some rules are specified to do some annotation tagging. This return `String` may in a further step be displayed by a web browser or processed further.

The class `Analyzer` implements this interface. Further, it provides a method
`public Map getCounters()` that returns a (Java) `Map` containing the counters. The key to the `Map` is simply the name of the counter (thus a `String`), while its value is of type `Integer`. The `Integer` object can simply be read out by using `intValue()` to receive an `int`.

The basic text analyzing algorithm works as follows:

1. Split up the input string into sentences by using `sentenceTokenizeRule`.
2. The basic algorithm to analyze a text is the following:
3. Split up every sentence into words by using `wordTokenizeRule`.
4. Repeat as long as there are more sentences:
   (a) Repeat as long as there are more words in a sentence:
      i. Go to the next word in the frame.
      ii. Try whether the current word matches the search criteria of the current rule.
      iii. If there is a match, then process the found part of the `action`.
      Otherwise, check whether there are still more words in this frame to be considered. If there are, choose the next one and go to 4a. If there are no more words in this frame, then the `search` was not successful. Process the not found part of the `action` in this case.
      iv. Eventually: Increment counters now and annotate the current frame with annotation tags if specified.
      v. Check for the next rule.
      If no such next rule is specified explicitly at this moment, then choose the initial rule again. The initial rule should be applied now to the first word in the sentence after the current word.
      Otherwise, apply the next rule to the current word.
   (b) Save the return value of applying the rule. This value now indicates the index position of the next word to be analyzed. This is now your current word. Go to 4a
5. Put all the possibly annotated/tagged words together to a single text again and return the final result string.

As we can see, this algorithm simply applies the initial rule sequentially to all words and if it once is successful, then it tries to process the next rule in the engine/rule tree to the word.

The `Analyzer` always works with word frames of a certain size that must be given in the rule tag. The frame's size should be set with care. If a frame is too small, certain grammatical constructs in a sentence, for instance words that are separated through other words but belong together, may not be found. Imagine the sentence *I have of course never seen something like this before.* If the frame size is set to a size of 3, the `Analyzer` is not able to connect logically the participle *seen* to the earlier encountered main verb *have*. On the other hand, if the frame size is too big, certain nonsensical forms may be believed to match the search criteria perfectly, although of course they do not.

4.5 The dictionary

We use a French dictionary[8] to look up inflections and lemmas of words. At program start, the whole dictionary is loaded into the computer's memory and stored in a simple hash table, which then is being used by the `search`. Each word in the text builds a key; the corresponding value is an object that is holder of all possible inflections of the word.

Due to the fact that a dictionary can never contain every possible word of a language and also depending on the tokenizing rules, it might be that certain words in the text are not found in the dictionary. The `Analyzer` will simply ignore this word and continue with the next word, thus never recognizing it as a match for the `search`. Such a behaviour should be tolerable as long as only a small percentage of the text's words are not contained in the dictionary.

5 Summary

Our program is surely not thought as a commercial product, but this has never been our goal. Indeed, with our prototype we have shown that basically it is possible to encode the linguists' rule sheets in a way, that they can more or less easily be changed independently of the rest of the source code without the need of recompilation. We used XML for this purpose.

The program needs a French dictionary to work with. At the moment, the availability of such dictionaries is not exactly the same as for English language. As our time resources were limited, we chose the easiest possible way to extract information from the dictionary. Of course, there is variety of better, but more complicated approaches from using databases to letter trees that improve memory usage[9] very significantly without great losses of performance.

A question still open is about the quality of analysis, which our program can reach. On the one hand, the more sophisticated the rules get, the more precise the results are. On the other hand, there will always be certain problems that cannot be answered fully either by the programmer or the linguist. What is the optimal word frame size for each rule, for example? An important point is that our program does not take into account any semantical information at all about the text to be analyzed, nor is a corpus illustrating *le bon usage* (of today) being used. However, we believe that extending our program to a semantical level too would be a much more demanding task, and it is dubious whether a semantical analysis would honour our huge efforts implementing it with any remarkably higher feedback quality.

Although our idea of splitting up the analyzation of repeatedly taken new `search`es and then `action`s, it is unceratin that such an approach really can cover the complexity of all linguistic rules. It was merely hard to find a least common general structures behind all rules. Of course, this structure now could be improved. One could think of adding several `search` parts to a single rule in the XML file, that then could be linked logically using an AND or an OR operator ("The whole `search` is only successful, if `search`1 AND `search`2 OR `search`3 are successful"). It would also be nice to have a graphical user interface for editing those rules.

[8] The dictionary is freely available for non-commercial purposes at `http://abu.cnam.fr`.

[9] At the moment the program uses up to 85 MB of RAM.

References

[Clahsen1986] Harald Clahsen. 1986. *Die Profilanalyse*, volume 1. Springer-Verlag, Berlin, Germany.

[Schlyter2003] Suzanne Schlyter. 2003. Stades de dveloppement en français l2. Avaliable at `http://www.rom.lu.se/durs/STADES_DE_DEVELOPPEMENT_EN_FRANCAIS_L2.pdf`.

Installation information

A Java prerequisites

Our program makes use of newer Java features to deal with regular expressions extensively. We ourselves used the J2SDK version 1.4.2, Standard Edition, to develop the program. Sun Microsystems introduced regular expression handling in version 1.4, so this is the oldest Java version that we can recommend to the user.

The latest Java 2 Software Development Kit and Java Runtime Environment can be downloaded from `http://java.sun.com/`.

B Insufficient initial heap size

Our program uses lots of memory! Be aware that if you start the program without defining special options for the Java Virtual Machine to increase the maximum heap size, the program might start to save information from the dictionary to the hard disk drive temporarily, not ending to write data to the disk. Under Windows implementations, you should therefore start the Java Virtual Machine as:

```
java -mx90M ...
```

This will set the JVM to use a maximum heap size of 90 megabytes which will be enough for our program to run. In our experiences, the program never used more then 86 MB or memory.

C Working with the Xerces-J XML DOM Parser

To parse the XML file, we used the freely available Xerces-J XML DOM parser. The program works fine with version 2.6.0. It can be found under: `http://xml.apache.org/xerces2-j/index.html` It is easiest to start your program, adding the corresponding paths by using the `-classpath` option. You need to add both the files `xercesImpl.jar` and `xml-apis.jar` to your `CLASSPATH`. Be aware that as soon as you set you these options, it might be that you have also to add the current working directiory (where your program source is located, for instance `se/lu/rom/sources`) and perhaps also the directory, where your binary runtime executables (`java.exe` under Windows), for instance `C:\Program files\jdk1.4.2`, are located.

```
java -classpath C:\...\Xerces-J\xercesImpl.jar;
                E:\...\Xerces-J\xml-apis.jar;
                [C:\...\jdk1.4.2\bin;
                C:\...\se\lu\rom\sources;]
                se.lu.rom.sources.MyMainClass
```

D Usage of the program classes

There is the central interface IAnalyzer and the implementing class Analyzer. Always use these two to get an instance of the Analyzer, as in:

```
/* Get the path of the rules file, e.g. from args[0] */
String rulesFilePath = args[0];

/* Get the dictionary to look up words as a Map, e.g. a Hashtable. Imagine the path
ImportDictController idc =
new ImportDictController(new File(args[1]));
Map dictionary = idc.importDictionary();

/* Instantiate your analyzer */
```

```
IAnalyzer myAnalyzer =
new Analyzer(rulesFilePath, dictionary);

/* Now do the analyzation and catch the result */
String outputText = myAnalyzer.analyze(inputText);
```

Now use the HTML annotated outputText to be displayed in your output window, for instance your browser.