selnolig-CHECK

Steffen Hildebrandt
Felix Lehmann

Linguistics 409:
Introduction to Computational Linguistics

Final Project

University of Massachusetts Amherst

December 2012

This document mostly reflects the state of affairs in December 2012, when we turned selnolig-check in. While we have improved several aspects of the project and implemented some of the ideas outlined in chapter 6, the general concept described here still applies.

This document was typeset using LuaLaTeX. The text is set in Linux Libertine and Linux Biolinum, code is set in DejaVu Sans Mono.

The title page design is based on an answer by egreg[1] to the question *Filling white space in title page with random numbers or a text*[2] by FormlessCloud[3] on TeX - LaTeX Stack Exchange.

The code documented here is available on GitHub[5] and is published under a Simplified BSD License.

[1] http://tex.stackexchange.com/users/4427/egreg

[2] http://tex.stackexchange.com/q/63383/4012

[3] http://tex.stackexchange.com/users/14524/formlesscloud

[4] http://creativecommons.org/licenses/by-nc-sa/3.0/

[5] https://github.com/SHildebrandt/selnolig-check

Contents

1

LaTeX and Ligatures – almost there!

1.1 LaTeX

LaTeX is a high-quality typesetting system. It is mainly used in academia, originally coming from the fields of Computer Science and Mathematics, but by now it has spread to about every field there is (including Linguistics, of course). As opposed to WYSIWYG word-processing programs like Microsoft Word, LaTeX documents need to be compiled before the actual output can be viewed. Nowadays, the most common output format is `.pdf`, compiled from a `.tex` file, frequently using the pdfLaTeX compiler. A noteworthy recent development in the world of LaTeX is LuaLaTeX, which – for the end user – is about the same as LaTeX, but it provides package authors with the additional power of the programming language Lua. It also is capable of hooking into the compilation process of LaTeX in places that hadn't been accessible before. LuaLaTeX is part of the two major distributions MiKTeX and TeX Live. Its current stable version is 0.6.

While technically it is TeX that does most of the things discussed here and not LaTeX, this differentiation is irrelevant for our purposes. Hence, we will always be speaking of LaTeX and LuaLaTeX, even if we may actually be referring to TeX or LuaTeX, respectively.

1.2 Ligatures

LaTeX incorporates many features of high-quality typesetting. The one that we are looking at for this project is *ligatures*. Ligatures are "combinations" of two letters, or more technically speaking *glyphs*, into a single glyph. So instead of setting e.g. an ⟨f⟩ *and* an ⟨i⟩, resulting in the two-glyph sequence ⟨fi⟩, LaTeX sets a single ⟨fi⟩-ligature. Such ligatures are typically used to avoid literal collisions of parts of the letters, or unfavorable spacing. Figure 1.1 shows the five standard LaTeX ligatures, which are part of many, if not most typefaces. Numerous typefaces contain additional ligatures as well, e.g. ⟨Qu⟩, ⟨Th⟩, ⟨fb⟩, ⟨fk⟩, ⟨tt⟩, ⟨ct⟩, or ⟨st⟩.

ff – ﬀ   fi – ﬁ   fl – ﬂ

ffi – ﬃ   ffl – ﬄ

Figure 1.1: The five "standard" LaTeX ligatures.

In most cases, these ligatures should make text more pleasant to read and improve the reading flow, e.g. for the words in (1) and (2).

(1) stuff, riff, fill, refine, flush, stifle, office, afflict

(2) Stoff, schaffen, finden, flüssig, Pflanze, offiziell, knifflig

However, when a ligature is used across a *morpheme boundary*[1], it impedes the reading flow – at least this is what a typographical rule-of-thumb says, and it seems sensible, as can be seen in the words in (3) and (4), which retain the allegedly undesirable ligatures.

(3) shelfful, selfish, leafless, dwarflike, briefly, rooftop, safflower

(4) Schaffell, Gugelhupfform, Kaufindex, Schilfinsel, Hofladen, straflos, Wahlkampffieber, auffinden, auffliegen

Unfortunately, LaTeX cannot deal with the ligatures in the words in (3) and (4) correctly. It recklessly sets a ligature whenever it can, not worrying about morpheme boundaries at all. Especially for German texts, this is a considerable drawback, since its wealth of overt, concatenating morphology leads to numerous cases of undesired ligatures.

1.3 selnolig

Mico Loretan wrote the package selnolig, which keeps LaTeX from inserting ligatures across morpheme boundaries, using the compilation hooks provided by LuaLaTeX and a list of recognition patterns.[2] Each of these patterns consists of a search string and a replacement string with a | character indicating the morpheme boundary, i.e. where *no* ligature should be set. At the end of the compilation, just before the text is actually put "on the paper", LuaLaTeX scans the entire text for all of the search strings and replaces them with the replacement strings.

[1]For our purposes, the basic definition is entirely sufficient: A morpheme is the smallest meaning-bearing unit in a word. Note that we are exclusively concerned with morphemes, not with syllables.

[2]A beta version of selnolig is available online: https://github.com/micoloretan/selnolig

Here are some of the 269 patterns selnolig currently[3] has for German:

```
\nolig{lflos}{lf|los} % hilflos
\nolig{ffäh}{f|fäh} % auffährt lauffähig hoffähig kampffähig
\nolig{mpffisch}{mpf|fisch} % Kampffisch
```

They follow standard LaTeX syntax: `\nolig` is the command name; it has two `{arguments}`, each in a pair of braces; the `%` symbol indicates a comment, which Mico always added to provide example words for the patterns.

As can be seen in the example above, the search patterns aren't entire words, but word fragments. This reduces the number of rules greatly – and indeed, it seems linguistically plausible that *lflos* is not a string of letters one would find within a single German morpheme.This approach also tackles the problem of the abundantly creative concatenating morphology of German: `\nolig{mpffisch}{mpf|fisch}` catches not only *Kampffisch*, but also *Kampffische*, *Kampffisches*, *Kampffischereiunternehmen*, *Lieblingskampffisches*, *Zuchtkampf fischereiexperte*, etc. At the same time, this approach introduces uncertainty that a list of entire words would most likely only have to a minute degree: For instance, the pattern `\nolig{ffäh}{f|fäh}` correctly deligaturizes *auffährt* etc., but it also erroneously removes the ligature in *stoffähnlich* and *schiffähnlich* – the pattern is too broad. On the other hand, there most likely are words with a potential ligature across a morpheme boundary for which selnolig does *not* have a pattern.

selnolig currently examines the following set of ligatures for German documents: ⟨ff⟩, ⟨fi⟩, ⟨fl⟩, ⟨fff⟩[4], ⟨ffi⟩, ⟨ffl⟩, ⟨fb⟩, ⟨fh⟩, ⟨fj⟩, ⟨fk⟩, ⟨ft⟩, and ⟨th⟩[4].

The goal of this project, named selnolig-check, is to find such flaws in the German patterns of selnolig and provide detailed information as a basis for their improvement.

[3]Our project is based on a slightly altered (cf. section 3.2.3) version 0.141 of selnolig from October 25, 2012. This document itself is typeset with LuaLaTeX version 0.7 and selnolig version 0.160.

[4]This ligature is not available in the font used for this document.

2 Our Approach

2.1 Project Layout

As implied in the previous chapter, we're currently, i.e. for this project, only looking at selnolig's German patterns. Our general approach is straightforward: It is essentially a large-scale automation of manual word-for-word testing. Manual testing would mean creating a LaTeX document that loads the selnolig package and has testing words in its body, compiling it, and comparing the results to our own morphological analyses. This gets tedious pretty fast, since coming up with testing words is not that easy, especially not with ones that haven't already been considered. Hence, we chose an automated approach: We take a big list of words with ligaturizable glyph combinations in them (regardless of morpheme boundaries), retrieve the relevant morpheme boundaries from a morphological analyzer, simulate selnolig on these words as well, and compare and categorize the results. This means that we are in fact not testing *the package* selnolig, but only the ancillary file that contains the German patterns. None of the Lua code or the other programs of selnolig play any role in our testing here.[1]

2.2 Resources

Our approach requires two elements that we would not have been able to create from scratch within the context of this class: the word list and the morphological analyzer. In order to get reliable results, especially for cases like *schiffähnlich*, where selnolig's patterns are too inclusive, we need a fairly extensive word list, covering more than just the most common words; since such a list of the dimension we envisioned most likely doesn't exist, it seemed feasible to extract it from a corpus ourselves. Automated morphological analysis, on the other hand, especially in a morphologically rich language like German, is an enormously

[1]We have been testing the *functionality* of selnolig outside of this project, of course, and given a lot of feedback to Mico.

complex undertaking beyond our reach. Fortunately, we were able to acquire two excellent resources to fill these two places.

2.2.1 SDeWaC

What were we looking for in a corpus? Many components of and packages for LaTeX (e.g. the hyphenation patterns, spacing, etc.) assume that their primary usage will be primarily for standard, current, written-style German. We are adopting this assumption for our treatment of selnolig. Hence, our corpus should also contain data from on this register and variety of German. Also, as mentioned above, it should contain a great number of different lexemes and different inflected forms of them. While LaTeX documents certainly often contain very specialized terminology from various areas of expertise, a first approach to general usage fields seems appropriate.

We settled for the SDeWaC corpus[2], provided by the **W**eb-**a**s-**C**orpus **k**ool **y**nitiative (WaCky); the entire acronym stands for *Stuttgart "Deutsch" Web as Corpus*. It is, as the name implies, a web-based corpus, but has been cleansed of noise quite thoroughly. It gathered data from the `.de` domain range, and thus covers a variety of different topics and levels of specialization. Overall, it contains 44,084,442 sentences, 846,159,403 word form tokens and 1,094,902 types.[3]

Here are the first three lines, i.e. sentences of the corpus – this is what our array of programs will have to process:

(1)
```
<year="0"/> <source="1403"/> <error="0.00869565217391304"/> Sie dürfen eine Kopie der Software
auf dem Dateiserver Ihres Computers installieren , um die Software auf andere Computer Ihres
internen Netzwerks bis zur zulässigen Anzahl herunterzuladen und auf ihnen installieren zu
können und Sie dürfen eine Kopie der Software auf dem Dateiserver eines Computers innerhalb
Ihres Netzwerks nur zu dem Zweck installieren , um die Software mittels Befehlen , Daten oder
Anweisungen ( z. B. Skripten ) von anderen Computern aus in demselben Netzwerk zu verwenden ,
unter der Voraussetzung , dass die gesamte Anzahl der Benutzer ( nicht die Anzahl der
gleichzeitigen Benutzer ) , denen der Zugriff auf die Software des Dateiservers oder die
Verwendung der Software gestattet ist , die zulässige Anzahl nicht überschreitet .
```

(2)
```
<year="0"/> <source="3708"/> <error="0.00892857142857143"/> Die Henker der Geschichte , die
Abkömmlinge Kains , machten sich über die Kinder Adams her und spalteten die in Einheit und
Eintracht lebenden Menschen in Herren und Knechte , Herrscher und Beherrschte , Satte und
Hungrige , Reiche und Arme , Meister und Diener , Tyrannen und Unterdrückte , Kolonialherren
und Kolonisierte , Ausbeuter und Ausgebeutete , Betrüger und Betrogene , Starke und Schwache ,
Verführer und Verführte , Besitzer und Besitzlose , Adelige und Bürgerliche , Geistliche und
Weltliche , Auserwählte und Gemeine , Freie und Unfreie , Arbeitgeber und Arbeitnehmer ,
Glückliche und Unglückliche , Weiße und Schwarze , Westliche und Östliche , Zivilisierte und
Unzivilisierte , Araber und Nicht-Araber .
```

---

[2] M. Baroni, S. Bernardini, A. Ferraresi and E. Zanchetta, 2009, The WaCky Wide Web: A Collection of Very Large Linguistically Processed Web-Crawled Corpora. *Language Resources and Evaluation*, 43 (3): 209–226.

[3] `http://wacky.sslmit.unibo.it/lib/exe/fetch.php?media=papers:sdewac-description.pdf`, accessed Dec. 2, 2012.

(3) `<year="0"/> <source="5609"/> <error="0.00892857142857143"/> Wenn ich dir jetzt vorschlüge , dich in die Zwangsjacke Ewigen Glücks zu stecken , sagen wir , ich würde dich in meinen Eksator einsperren , damit du dort die nächsten einundzwanzig Milliarden Jahre in höchster und reinster Glückseligkeit verbringen könntest , statt dich in dunkler Nacht auf Friedhöfen herumzudrücken , die Gebeine deines Professors in ihrer ewigen Ruhe zu stören und Informationen aus Gräbern zu stehlen , wenn du zudem die Suppe nicht mehr auszulöffeln hättest , die du dir eingebrockt hast , wenn du frei wärst von allen künftigen Aufgaben , Sorgen , Problemen und Mühen , die unser tägliches Dasein belasten und bedrücken - wärst du dann mit meinem Vorschlag einverstanden ?`

There remain some things in the corpus we'll have to deal with, e.g. the introductory tags for each sentence, misspellings, words so rare they could be considered irrelevant for our purposes, foreign words, strange glyphs[4], words that were obviously hyphenated at the end of a line, using a variety of different Unicode glyphs as hyphens[5], and a similar variety of quotation marks.

2.2.2 SMOR

What were we looking for in a morphological analyzer? From a pragmatic point of view, there is by far not as much choice as for corpora. Nonetheless, we needed a program that would reliably indicate all morpheme boundaries in a word; the task of filtering out the relevant morpheme boundaries is one we would easily be able to carry out ourselves. Since we're looking at the graphic form of words, we only really care for morphemes with a graphic representation (which would usually be called a phonological representation, i.e. we're not interested in null morphemes). Among these morphemes, our first approach is just treating all morpheme boundaries equally, regardless of whether they are free or bound, grammatical or lexical (cf. section 6.4).

SMOR[6] is "a morphological analyser for German inflection and word formation implemented in finite state technology", which "can account for productive word formation"[7], created by the *Institut für maschinelle Sprachverarbeitung* at Universität Stuttgart. SMOR's "normal" output doesn't quite have the form we need, but we can tweak it according to our needs (cf. section 3.2.2). Nonetheless, here are a few example analyses from the "standard" SMOR[8]:

---

[4] There even are a few instances of the Unicode ligature glyphs, which seem to have resulted in a splitting of the word during the cleaning up of the corpus: `Pfl ichtveranstaltungen` (note the ligature followed by a space, which remain visible in the plain text code here).

[5] U+00AD 'hyphen-minus', U+002D 'minus sign', U+2010 'hyphen', U+2211 'non-breaking hyphen', U+2212 'figure dash'.

[6] Helmut Schmid, Arne Fitschen and Ulrich Heid: SMOR: A German Computational Morphology Covering Derivation, Composition, and Inflection, *Proceedings of the IVth International Conference on Language Resources and Evaluation (LREC 2004)*, p. 1263-1266, Lisbon, Portugal.

[7] `http://www.ims.uni-stuttgart.de/projekte/gramotron/PAPERS/LREC04/smor.pdf`, accessed Dec. 4, 2012.

[8] For the meaning of SMOR's various tags, see `http://www.ims.uni-stuttgart.de/projekte/dspin/ch01s03.html#Tags`.

```
analyze> Suppenkasper
Suppe<NN>Kasper<+NN><Masc><Acc><Pl>
Suppe<NN>Kasper<+NN><Masc><Acc><Sg>
Suppe<NN>Kasper<+NN><Masc><Dat><Sg>
Suppe<NN>Kasper<+NN><Masc><Gen><Pl>
Suppe<NN>Kasper<+NN><Masc><Nom><Pl>
Suppe<NN>Kasper<+NN><Masc><Nom><Sg>

analyze> Eintracht
Eintracht<+NN><Fem><Acc><Sg>
Eintracht<+NN><Fem><Dat><Sg>
Eintracht<+NN><Fem><Gen><Sg>
Eintracht<+NN><Fem><Nom><Sg>
```

```
analyze> Treibhausgases
Treibhaus<NN>Gas<+NN><Neut><Gen><Sg>
treiben<V>Haus<NN>Gas<+NN><Neut><Gen><Sg>

analyze> tanzen
tanzen<+V><3><Pl><Pres><Subj>
tanzen<+V><3><Pl><Pres><Ind>
tanzen<+V><1><Pl><Pres><Subj>
tanzen<+V><1><Pl><Pres><Ind>
tanzen<+V><Inf>

analyze> hintergangen
hinter<VPART>gehen<+V><PPast>
hinter<VPART>gehen<V><PPast><SUFF><+ADJ><Pos><Adv>
hinter<VPART>gehen<V><PPast><SUFF><+ADJ><Pos><Pred>
```

Unfortunately, SMOR also presents some analyses that do not coincide with the analyses we would provide for these words. Here are some cases that are actually relevant for our very project since SMOR doesn't provide a morpheme boundary between two ligaturizable glyphs, where we would see one:[9]

```
analyze> beauftragen
beauftragen<+V><3><Pl><Pres><Subj>
beauftragen<+V><3><Pl><Pres><Ind>
beauftragen<+V><1><Pl><Pres><Subj>
beauftragen<+V><1><Pl><Pres><Ind>
beauftragen<+V><Inf>
missing: beauf|tragen

analyze> aufhaltsam
aufhaltsam<+ADJ><Pos><Adv>
aufhaltsam<+ADJ><Pos><Pred>
missing: auf|haltsam
```

```
analyze> Auftakt
Auftakt<+NN><Masc><Acc><Sg>
Auftakt<+NN><Masc><Dat><Sg>
Auftakt<+NN><Masc><Nom><Sg>
missing: Auf|takt

analyze> elfhundert
elfhundert<+CARD><Pro><NoGend><Acc><Pl><Wk>
elfhundert<+CARD><Pro><NoGend><Dat><Pl><Wk>
elfhundert<+CARD><Pro><NoGend><Gen><Pl><Wk>
elfhundert<+CARD><Pro><NoGend><Nom><Pl><Wk>
missing: elf|hundert
```

Nonetheless, the vast majority of SMOR's analyses agree with the analyses we would make.

[9] We specifically deal with these and some other analyses that we consider not apt for our purposes, cf. section 3.2.2. We have not yet sent our feedback to the creators of SMOR, but we will discuss these analyses and other observations with them.

3 The Programs

In this chapter, we will describe how we built up our testing apparatus. We split the testing procedure into two processes: First, we transform the corpus into a much smaller testing dictionary containing all relevant words, and then we run the testing dictionary through SMOR and selnolig and compare their results.

As an overview, here are two example words and their entire way through our programs. We start with two more sentences from the corpus – truncated –, which is where our data processing begins. The words we're following are underlined here while there are other words listed.

```
<year="0"/> <source="6612"/> <error="0.0571428571428571"/> Fröhliche Auferstehung feierte auch
der " Gefrierfleischorden " , […] .
<year="0"/> <source="1149"/> <error="0"/> Wenn […] , dass nach dem Gottesdienst alle Mitarbeit
für einige Stunden zum Schaffang gehen sollen , während […] .
```

`corpus to words` resulting file `words.raw`:

```
Fröhliche
[…]
Gefrierfleischorden
[…]
Wenn
[…]
Schaffang
[…]
```

`words to ligs` resulting file `ligs.good.normal`:

```
Gefrierfleischorden
Schaffang
```

`ligs to ligdict` resulting file `ligdict`:

```
Gefrierfleischorden
Schaffang
```

`ligdict to smor` resulting file `smor`:

```
> Gefrierfleischorden
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Nom>:<><Sg>:<>
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Nom>:<><Pl>:<>
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Gen>:<><Pl>:<>
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Dat>:<><Sg>:<>
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Dat>:<><Pl>:<>
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Acc>:<><Sg>:<>
g:Gefriere:<>n:<><V>:<>F:fleisch<NN>:<>O:orden<+NN>:<><Masc>:<><Acc>:<><Pl>:<>
> Schaffang
Schaf<NN>:<>F:fang<+NN>:<><Masc>:<><Nom>:<><Sg>:<>
Schaf<NN>:<>F:fang<+NN>:<><Masc>:<><Dat>:<><Sg>:<>
Schaf<NN>:<>F:fang<+NN>:<><Masc>:<><Acc>:<><Sg>:<>
s:Schaff:<>e:<>n:<><V>:<><OLDORTH>:<>F:fang<+NN>:<><Masc>:<><Nom>:<><Sg>:<>
s:Schaff:<>e:<>n:<><V>:<><OLDORTH>:<>F:fang<+NN>:<><Masc>:<><Dat>:<><Sg>:<>
s:Schaff:<>e:<>n:<><V>:<><OLDORTH>:<>F:fang<+NN>:<><Masc>:<><Acc>:<><Sg>:<>
```

`smor to morphemes` resulting file `morphemes.bad.oldorth`:

```
Schaffang -> Schaf<V><OLDORTH>fang<+NN><Masc><Nom><Sg>
Schaffang -> Schaf<V><OLDORTH>fang<+NN><Masc><Dat><Sg>
Schaffang -> Schaf<V><OLDORTH>fang<+NN><Masc><Acc><Sg>
```

resulting file `morphemes.good`:

```
Gefrierfleischorden -> Gefrierfleischorden
Schaffang -> Schaf|fang
```

`morphemes to analyses` resulting file `analyses.bad`:

```
Schaffang --- Schaf|fang --- Schaffang ---
```

resulting file `analyses.good`:

```
Gefrierfleischorden
```

`analyses to errors` resulting file `errors.type1.ff`:

```
Schaffang --- Schaf|fang --- Schaffang ---
```

(Only words from `analyses.bad` end up in an `errors.*` file, of course.)

As can be seen in the files `morphemes.bad.oldorth` or `analyses.good`, we produce files with which our programs don't do anything afterwards. There are two reasons for this, which are connected to each other in a way: Firstly, it is interesting to look at these lists, in the case of the analyses according to the old German orthographic rules often even amusing[1].

[1]One main difference here is that *Schiff* and *Fahrt* would have been concatenated under omission of one ⟨f⟩, resulting in the double-f *Schiffahrt*, instead of the triple-f *Schifffahrt* that New Orthography prescribes. Hence,

Secondly, we often could improve our programs and make them more inclusive by looking and words we excluded.

3.1 Building the Testing Dictionary

This process extracts the single words from the corpus, filters out some non-well-formed words, only keeps words with a potential ligature (cf. section 3.1.2), and creates the testing dictionary, removing duplicates to accelerate the second process. This step is intended to be executed only once on one corpus since its result will be completely independent of SMOR and selnolig, which are covered in the second process.

3.1.1 `corpus_to_words`

This program is probably the simplest and most straightforward part of the project. The main task is to get rid of all tags and to convert the sentences in the corpus into a bare list of words.

Decisions

We only keep words containing at least one letter. In this case, we define a word as a sequence of letters surrounded by whitespace.

Layout

The main method opens the corpus file and reads it line by line. In each line, we replace all tags by the empty string and split the line at whitespace. Afterwards, each of the resulting words is written to the output file *iff* it contains at least one letter.

3.1.2 `words_to_ligs`

This is one of the most involved parts of the project. We needed to decide which words to take into consideration, i.e. how to deal with words containing punctuation (especially hyphens and periods) or unusual capitalization. The program splits the output into several files, each containing words of a special category. These categories are grouped into `good` (words that we don't expect to cause problems), `review` (words we want to look at "manually" before passing them on to the next step), and `bad` (words that are most likely useless).[2] This way,

according to Old Orthography, the genitive *Schiffes* also is analyzed as {schiff}+{fes} 'ship-fez', an oriental hat worn aboard a vessel; and *Kaffee* is also analyzed as {kaff}+{fee} 'hicksville-fairy', a fantasy creature living in a small town that is referred to with a derogatory term.

[2]There is also the group `interesting` containing two subcategories: words in all caps and single-letter abbreviations. These are just there for our personal curiosity, to see what kinds of words are in the corpus.

it is much easier for us to judge the quality the results of this step and to look for further necessary adjustments and improvements.

Decisions

First of all, we filter out all words that do not contain any potential ligatures. Words containing a ligature across a punctuation sign are put into a special category. The naming for each output file follows this convention:

```
ligs.<category-group>.[<subgroup>.].<category>
```

In particular we defined the following categories:

`good.normal` Words in which we don’t find anything special.

`good.innen` Example: *WissenschaftlerInnen.* The suffix {er} indicates only a male agent. Adding the suffix {in} makes the male agent female. Using a capital ⟨I⟩ within the word is a graphically politically correct gender-neutral way of referring to acting entities. ({nen} makes it plural.) SMOR is able to deal with such cases, so we can use them.

`good.laws` Examples: *KraftStG*, *AltPflAPrV*[3]. Abbreviated laws and ordinances are the only case in which we allow a word to be in camel case. We identify them by testing whether they are longer than two letters, start with upper case, and end with “G” or “V”, which works quite well. Nonetheless, SMOR will not recognize most of them, so we might at some point manually compile a list of ligature-relevant laws to include in our testing[4].

Hyphenated words are by far the most complicated cases, especially since we had to take into account some limitations of SMOR: Hyphens at the end of a word prevent it from being analyzed, whereas hyphens at the beginning of a word will allow it to be analyzed regardless of whether it starts with upper case or lower case. In the end, we decided to split hyphenated words into the following three `good.hyphen` files:

`good.hyphen.startsWithHyphen` Words containing only one hyphen at their beginning.

`good.hyphen.beginnings` Example: *Wirtschafts-Journalisten.* Parts of a hyphenated word split by hyphens, dropping the last part and the hyphen. (Also applied “recursively” to words of the form A-B-C.)

---

[3]This abbreviation stands for *Altenpflege-Ausbildungs- und Prüfungsverordnung* and is actually not analyzed by SMOR.

[4]Most likely, all ligatures in laws will be kept, analogously to the abbreviations *Aufl.* or *gefl.*, which keep their ligatures even though they occur across what would be a morpheme boundary in the full word. Cf. the documentation of the selnolig package.

`good.hyphen.end` *Wirtschafts-Journalisten*. The last part of words with at least one hyphen, including the hyphen starting the final string.

The remaining cases end up in the following three files, which we don't use later:

`review.punctuation` Words containing punctuation other than periods or only hyphens.

`review.ligAcrossPunctuation` Words containing a ligature across some punctuation sign.

`bad.camel` Words in camel case *not* identified as laws.

Layout

The complex decisions for this program are inevitably mirrored in the complexity of the code. Essentially, we defined a function for each conditional (i.e. `all_caps`, `is_law`, …) we use, and then let the main function `filter-lig-words` coordinate the evaluation process. The rest of this code works similarly to previous and following programs: We loop over the lines of the input file and put the result into the respective output file, depending on our analysis.

3.1.3 `ligs to ligdict`

This program merges the `ligs.good.*` files, eliminating duplicates, and finally writes the output, converting the encoding into latin1, because SMOR is not able to read UTF-8.

Decisions

We didn't have to make any decisions here.

Layout

All words are read into a `set` to implicitly remove duplicates, and then written to the latin1-encoded file `ligdict`.

3.2 Testing selnolig's Patterns

In this second process, we take the testing dictionary `ligdict` and use SMOR to split each word into its morphemes. Then we apply the selnolig rules to each word and check whether ligatures are suppressed *iff* SMOR indicates a morpheme boundary between the ligaturizable glyphs. Finally, the errors, i.e. words that SMOR and selnolig do not agree on, are categorized by type, ligature, and some other characteristics.

3.2.1 `ligdict to smor`

This program executes SMOR on `ligdict`.

Decisions

Instead of `fst-mor`, the "standard" finite state transducer of SMOR demonstrated in section 2.2.2, we use `fst-infl2`, which has some crucial advantages for our purposes: Firstly, it is capable of reading a file and writing the output the another file (whereas `fst-mor` is working interactively reading from and writing to the command line). Secondly, it provides the option `-b`, causing the analyzer to keep the morphemes in their inflected form rather than reducing them to their basal forms (e.g. infinitive or nominative singular). This option is essential for us since there are cases where it would be impossible to derive the correct morpheme boundaries from the `fst-mor` output.

Layout

This is the particular call we use:

```
./fst-infl2 -b -q ./lib/smor.ca ./ligdict ./smor
```

We worked with Cygwin on Windows 7 (cf. section 4.3).

3.2.2 `smor to morphemes`

This program processes and partly modifies the output of SMOR, in preparation for the comparison with selnolig's results. It results into four files, which are explicated below. There are three steps to execute on each line:

1. Clean the SMOR output. Since `fst-infl2` with the option `-b` marks all morphemes with information we are not interested in, this first step consists of deleting or substituting these special marks by appropriate tags.[5]

2. Mark morpheme boundaries, but only keep if it is between two ligaturizable glyphs.

3. Fix some known "bugs" of SMOR (cf. section 2.2.2), i.e. insert morpheme boundaries in places where we know that SMOR doesn't insert them. This includes for example the word fragment *beauftrag*, where SMOR doesn't analyze a boundary between ⟨f⟩ and ⟨t⟩.

[5]Helmut Schmid, one of the creators of SMOR, helped us in finding the optimal configuration of SMOR and provided us with a Perl script to process the substitutions, which we translated into Python.

Decisions

SMOR's analyses[6] implicitly serve as an important "sanity check" for our entire project: It only analyzes words whose base morphemes are in its extensive lexicon, ruling out the remaining irregularities in our corpus mentioned in section 2.2.1. Thus, misspelled words, words in a language other than German, and generic gibberish will be part of our testing of selnolig, but instead are put in a dedicated file (`morphemes.bad`).

Unfortunately, SMOR often provides multiple analyses with differing morpheme boundaries for one word; while it might be possible to programmatically analyze if these analyses actually are different "words" or just different sizes of morphological "chunks", we decided not to delve into this differentiation at this point, but instead only use words with an unambiguous anlysis of morpheme boundaries between two ligaturizable glyphs (also cf. section 6.2). Words with clashing analyses are put in a dedicated file (`morphemes.differentPossibilities`).

As demonstrated in the beginning of chapter 3, SMOR provides a special tag for analyses that are only possible according to the rules of Old Orthography. Since this would yield numerous unlikely, but differing analyses, unnecessarily sorting words into `morphemes.differentPossibilities`, we decided to ignore all Old Orthography-analyses for now (cf. section 6.3) and put them in `morphemes.bad.oldorth`.

Everything else, i.e. words with unambiguous analyses of morpheme boundaries between two ligaturizable glyphs, ends up in the file that will be used in the ensuing programs: `morphemes.good`.

Layout

This is the only step where we couldn't process all the lines independently from each other, which is why the structure of the code gets slightly more complex. We pre-process each line with the three steps outlined above and put it in a set containing all analyses of the current word. Then we analyze this set of analyses of the current word. If it contains exactly one entry (because there is only one analysis or several analyses equivalent for our testing), the word is written to `morphemes.good`, otherwise it is written to `morphemes.differentPossibilities`.

3.2.3 `morphemes to analyses`

This is the decisive program of our project. It compares the morpheme boundaries between two ligaturizable glyphs indicated by SMOR (and adjusted in the last step) to those according to the selnolig patterns. If both conform, the respective word is put into `analyses.good`[7],

[6] The usage of the word *analysis* in the following is different from the meaning of *analysis* in the next section. Here, we refer to the results of SMOR as *analyses*, whereas in section 3.2.3 an *analysis* is the result of comparing SMOR and selnolig on a specific word.

[7] All the words in the background of this document's title page are taken from `analyses.good`.

otherwise it is put into `analyses.bad` together with the conflicting analyses and the selnolig patterns applied.

Decisions

We altered selnolig's patterns file in a few places: Some patterns were commented out due to a bug in selnolig on the Lua level, which is irrelevant for our testing; furthermore, the ⟨th⟩-patterns were commented out because Mico considered them too incomplete to be applied to a text – but that's even more of a reason for us to test them, of course.

Layout

We simulate selnolig by walking through all patterns and applying them if they match. This corresponds to the approach of selnolig and should produce the same results. The code for reading the patterns file is swapped out to a separate module called `morphemes_to_analyses__read_selnolig_patterns`. This module defines the function `read_rules`, which returns the patterns as a dictionary. Since we replaced all morpheme boundaries with the same symbol, the comparison between SMOR and selnolig can simply be done by a string comparison.

3.2.4 `analyses to errors`

The previous program tells us in which analyses SMOR and selnolig disagree, but they are still just one big list, not sorted in any way helpful for improving selnolig's patterns. There are two kinds of shortcomings in selnolig that we are trying to detect: *type 1 errors* – selnolig erroneously does not break up a ligature; and *type 2 errors* – selnolig erroneously breaks up a ligature. (For this program, we assume that SMOR is always correct, which turns out not to be the case, but that's easily recognizable for our end user (i.e. Mico).) In this documentation's terminology, we differentiate between *error*, which is what our project aims to find, and *bug*, which is something like a "known bug" (in either SMOR or selnolig). In other words, a selnolig *bug* is to be understood as an imperfection in its patterns, whereas a selnolig *error* is a word that selnolig doesn't deal with correctly (as a result of a bug in its patterns). A *ligature* here is a combination of letters that will result in a ligature if selnolig doesn't do anything about it (and if the font used provides that ligature), i.e. the term *ligature* by itself, as used for this program, doesn't say anything about whether there *should* be a ligature or not.

Decisions

The recognition patterns of selnolig feature a few bugs that result in numerous errors, most notably the pattern `\nolig{flich}{f|lich}`, which not only correctly deligaturizes words like *brieflich*, but also removes the ligature in *Pflicht* and all words containing {pflicht} – which turn out to be extremely frequent (cf. table 5.4). Hence, we decided to form dedicated subcategories for a few high-frequency errors in selnolig's patterns, in order to make the other errors – results of "unknown bugs" – more useful. The general category-layout is straightforward: There are supercategories for type 1 and type 2 errors, each containing subcategories for the known bugs and for each ligature. If there are several points of interest within one word – i.e. SMOR, selnolig, or a combination of the two detected morpheme boundaries at several positions in the word – the word is put into several categories. In other words, one line in `analyses.bad` can lead to multiple lines in the `errors.*` files. The information which ligature is currently being examined is encoded in the symbol that is eventually written to the `errors.*` file (cf. the header of each of these files).

Once the sorting into (sub)categories is done, we tried to sort the errors within each list in a way that would be maximally intuitive for a human reader of the `errors.*` files. The sorting criteria (`make_type1_key()` and `make_type2_key()`) turned out surprisingly complex, but pleasingly effective. The clue of the sorting is going from right to left, starting at the ligature under inspection. A more detailed explanation is given in the comments to the code.

Layout

This program splits each line into four parts: the word itself, SMOR's result, selnolig's result, and the selnolig patterns applied, if any. The two results are then made to be of equal length: At every point where *either* of the two has a morpheme boundary, both strings receive a Greek letter, which encodes information about the morpheme boundary:

α Originally, there was *no* ligature detected at this point in this string, but in its counterpart.

β Originally, there *was* a ligature detected at this point in this string, but not in its counterpart.

γ Originally, there *was* a ligature detected at this point in this string, but also in its counterpart.

δ, ε, ζ, η, θ, ι These are assigned to the various known bugs, *instead* of α or β.

With the strings being of equal length, we can just loop over the characters of either string and compare them easily, and thus are instantaneously able to put them into a category. The reason why we used Greek letters is simple: We started out with numbers, but then changed an earlier program to allow for words with numbers in them, which made numbers unusable for the error-category-sorting. The Greek alphabet has a similar ordered nature, so it made for a good replacement, although it looks unfamiliar in the code.

There probably are programmatically "cleaner" ways of performing this error-analysis – e.g. introducing classes instead of Greek symbols, which would most likely imply representing the whole word as a class – but that would have resulted in even more complicated code. Since our solution works and completes a run within seconds, we left it at this.

4 Global Issues

4.1 Reading Large Files

One of the first problems we were confronted with was opening the corpus file, which is about 8 GB in size. Since Windows' built-in *Notepad* as well as *Notepad++* first index a file when opening it, i.e. they read the entire file before showing it, working with either of these editors would have been remarkably unnerving. Our search for an appropriate editor finally lead us to *EmEditor*[1], which allows the user to load only parts of a text file.

Additionally, this tool helped us with some encoding issues by providing detailed information about characters. In particular, we once encountered an issue with the different line endings in UNIX and Windows: While UNIX specifies a line ending just by a *line feed* (`LF`), Windows uses *carriage return* and *line feed* (`CR+LF`). *EmEditor* helped us to figure out that SMOR reads and writes UNIX-like line endings, so that we were able to adjust our code accordingly.

4.2 Unicode

By far the greatest and virtually omnipresent problem was the encoding of strings. At the beginning, we didn't pay close attention to this issue, assuming that we could play it by ear. In the end, however, we realized that we would not get it to work unless we dealt with encodings properly and consistently. While SDeWaC and the selnolig patterns are encoded in UTF-8, `fst-infl2` can only handle latin1, which forced us to switch the encoding from UTF-8 to latin1 in `ligs_to_ligdics`, and back to UTF-8 in `smor_to_morphemes`. Although this might sound quite easy and straightforward, we went through a lot of trouble with this issue: Firstly, the operations on Unicode often are different from the ones used on "normal" *bytestrings*; secondly, the operations to convert strings from one encoding into another are

[1] `http://www.emeditor.com`

not very intuitive; thirdly, the error messages thrown in these cases usually did not really help.

4.3 Compiling SMOR

Another problem was getting SMOR to work. Since the SMOR files did not include executable binary files, but only C code with a corresponding make file, we had to compile it on our own. While this is usually not an issue on UNIX systems, it is quite a challenging task on Windows, which is what we are using. In the beginning, we used Ubuntu running on a Virtual PC, which worked out well in general, but was really cumbersome to use. So eventually, we switched to Cygwin[2], which runs programs on Windows that were designed for UNIX. This method turned out to be much more user-friendly, so we stuck with it.

[2]http://www.cygwin.com

5 Statistical Analysis

A brief look at some figures we gathered while and after running our programs offers interesting insights. We can tell how well the different component cooperated (e.g. the part of the words that SMOR was able to analyze), and eventually even make some statements about the quality of selnolig's patterns.

5.1 Testing Volume

Table 5.1 shows the numbers of analyses of SMOR that we ended up using (`morphemes.good`) or not using (rest). 71 % seems like a good turnout, considering that we're not using entries with ambiguous analyses (`morphemes.differentPossibilities`). Excluding `morphemes.oldorth` doesn't do a lot of harm, but but spares us a lot of problems. It might be worthwhile to take a closer look at `morphemes.bad` to find out if there are any prominent aspects excluding many entries.

The most interesting number here might be the number of entries in `morphemes.good`: 461,939 – this is the number of distinct tokens we use to test selnolig. It seems safe to say that at this point, the automated testing of selnolig has proven to be much more efficient than the manual testing.

Table 5.1: The files `morphemes.*`, output of `smor_to_morphemes`.

file `morphemes.…`	# of entries	percentage
`good`	461,939	71.4 %
`differentPossibilities`	28,049	4.3 %
`oldorth`	11,146	1.7 %
`bad`	145,717	22.5 %

Table 5.2: The files analyses.*, output of morphemes_to_analyses.

file analyses.…	# of entries	percentage
good	426,552	92.34 %
bad	35,407	7.66 %

Table 5.3: Type 1 error distribution.

file errors.type1.…	# of entries	percentage
ff	1,192	6.81 %
fi	875	5.00 %
fl	1,079	6.17 %
ft	958	5.48 %
fb	0	0.00 %
fh	0	0.00 %
fk	0	0.00 %
fj	0	0.00 %
th	10,678	61.03 %
ig	1,050	6.00 %
innen	271	1.55 %
t-Endung	1,287	7.36 %
isch	106	0.61 %

Table 5.4: Type 2 error distribution.

file errors.type2.…	# of entries	percentage
ff	82	0.45 %
fi	1	0.01 %
fl	1,232	6.82 %
ft	2,816	15.58 %
fb	29	0.16 %
fh	35	0.19 %
fk	66	0.37 %
fj	59	0.33 %
th	2,757	15.26 %
hälfte	835	4.62 %
pflicht	10,159	56.22 %

(The 4+2 files below the middle rules are "known bugs" categories.)

Table 5.2 analyzes the analyses on which SMOR and selnolig agreed or disagreed, out of all the words in morphemes.good. A 92.34 % success rate of selnolig indicates that its patterns already were very good (consindering that there are some extremely rare words in the corpus, and also considering that we included the rudimentary ⟨th⟩-patterns, which led to many errors); the remaining 7.66 %, however, also indicate that it was well worthwhile to make the effort in testing we made as a basis for improvement.

5.2 Error Distribution

Looking at the figures in table 5.3 and table 5.4, selnolig's patterns seem to have been constructed remarkably carefully: The majority of the errors (excluding the extremely common and "known bug" {pflicht} case) were type 1 errors, which means selnolig's patterns typically aren't overly broad, in particular regarding the "standard" ⟨ff⟩, ⟨fi⟩, and ⟨fl⟩ ligatures.

It is not surprising that there are no type 1 errors for the ⟨fb⟩, ⟨fh⟩, ⟨fj⟩, and ⟨fk⟩ ligatures

Table 5.5: Runtimes of our programs and the file sizes of the resulting files.

program	runtime	resulting file(s)	size
`---`	`---`	`corpus.raw`	7,772,392 KB
`corpus_to_words`	5,405 s	`words.raw`	5,292,374 KB
`words_to_ligs`	11,798 s	`ligs.*`	283,979 KB
`ligs_to_ligdict`	65 s	`ligdict`	10,515 KB
`ligdict_to_smor`	138 s	`smor`	489,885 KB
`smor_to_morphemes`	695 s	`morphemes.*`	31,002 KB
`morphemes_to_analyses`	22 s	`analyses.*`	12,337 KB
`analyses_to_errors`	4 s	`errors.*`	2,462 KB
total	18,127 s	total	13,894,946 KB
	≈ 5 h		≈ 14 GB

because selnolig's approach is to suppress all of them. The type 2 errors for these ligatures are words of overtly non-German origin like *Nordfjord* or *Kafka*, proper names, or faulty analyses by SMOR.

The lonely type 2 ⟨fi⟩ error is *Spielefindex* (which SMOR analyzed as {spiel}+{e}+{find}+{ex}[1]), which we presume to be a typo, so selnolig did nothing wrong here.

The ⟨th⟩-ligatures are relatively frequent in both error types, which is – as mentioned before – not surprising.

5.3 Runtimes and File Sizes

The figures in table 5.5 are not particularly surprising; we start out with a huge amount of data, and accordingly it takes a lot of time to process it. As file sizes decrease, runtimes decrease. However, steps resulting in a dramatically smaller file tend to take a little longer, since they're typically using many loops and conditionals (`words_to_ligs` and `smor_to_morphemes`). A noteworthy exception to steadily decreasing file size is `ligdict_to_smor`: The resulting `smor.*` files contain several lengthy lines for each word in `ligdict`. Taking this increase in file size into consideration, its runtime is comparatively short, likely because SMOR works with finite state automata.

[1]We're not quite sure what this *could* mean, perhaps something along the lines of 'my former girlfriend with whom I regularly find games'. Go figure.

6 Future Ideas

6.1 Implement `\keeplig`

Before we ran our tests, we already suspected that selnolig would need something like exceptions to the `\nolig` patterns. Our most common example is the aforementioned pattern `\nolig{flich}{f|lich}`, which applies to many words, since {lich} is a common derivational morpheme. Unfortunately, however, there is the resulting {pflicht} error. This asks for an exception to this pattern, making it "no ligature in *flich*, unless it's *pflicht*". To deal with this issue, the real current version of selnolig (as opposed to the one we're using in our project) has a `\keeplig` macro. In order to truly check selnolig's state of affairs, we will need to teach our programs to deal with `\keeplig`.

The situation actually turned out to be even more complicated, as our results did not only confirm our assumption that a `\keeplig` is necessary, but also showed that it might not even be enough: We found a number of words, e.g. *Sumpflicht*, where the `\keeplig` pattern would erroneously override the `\nolig` pattern. We have developed two approaches of dealing with this problem, one with native LaTeX structures and one with a completely redesigned XML-patterns-file, both of which we are discussing with Mico.

6.2 Deal with Different SMOR Analyses

As outlined in section 3.2.2, we currently ignore all cases in which SMOR doesn't give us unambiguous analyses. However, using the detailed information SMOR provides through various tags, it should be possible to include some more cases, in which we can rule out one analysis, e.g. a case where a string has two analyses, as one noun or as two concatenated nouns. Our suspicion would be that the more detailed analysis will usually be the one we would want to use.

In this step, we could also improve of our "bug" lists for SMOR in order to have fewer false positives in our `error.*` files.

6.3 Produce Differentiated Output for Different Varieties of Written German

As outlined in section 3.2.2 as well, we currently ignore analyses according to Old Orthography. Furthermore, words containing ⟨ss⟩ instead of ⟨ß⟩ are not analyzed by SMOR, since they're considered to be misspelled. In Switzerland, however, ⟨ß⟩ has been abandoned altogether in favor of ⟨ss⟩ – and SMOR seems to be capable of recognizing such cases, if used with the right configuration.

Bundling this information, we might be able to create differentiated sets of patterns for German (Old Orthography), German (New Orthography), and Swiss German, next to a shared set, which will most likely be the largest one. We still have to find out if Austrian orthographic conventions differ from the German ones, which might result in yet another set of patterns.

Differentiation between these written varieties is common in localized LaTeX packages and it should be possible to implement them in selnolig relatively easily.

6.4 Differentiate Bound Grammatical Morphemes

With our current approach to suppressing ligatures, we would suppress the ⟨ft⟩-ligature in *ruft*, because {ruf} is the stem of the verb and {t} is the inflectional morpheme. Since it doesn't seem beneficial to the reading flow to suppress this ligature, we hypothesized that it might make sense *not* to deligaturize inflectional morphemes in general, leaving the deligaturization of derivational morphemes, which form the other substantial group of morphemes within the bound grammatical morphemes, untouched.

It should be possible to implement this differentiation into our programs, once again based on the detailed tags provided by SMOR.

6.5 More Corpora

The more words we take into consideration, the better our testing of selnolig will be, and the better selnolig's patterns will become. Incorporating more corpora should be relatively straightforward, once we have found more suitable corpora and acquired the respective necessary licenses. Interesting extensions would be more general-usage corpora, as well as more specialized corpora aiming at a widening of the content scope of selnolig.

A minor improvement on the corpus level would be a "blacklist" of words to be excluded from our testing dictionary. Those would be words that get through SMOR, but that we still don't want to be part of our testing dictionary, simply because they do not seem to be "real" words that we want to have any influence on selnolig's patterns. An example of such a word is *Spielefindex*, mentioned in section 5.2.

6.6 Automatically Generate and Improve selnolig's Patterns

The epitome of elegance for our project would probably be a fully automated re-use of the errors in the patterns that we detected, i.e. an algorithm that automatically corrects and optimizes selnolig's patterns. While we have some initial ideas for such an algorithm and are generally well-accustomed with the issue of morpheme-boundary-based deligaturization, this would be a huge and complex undertaking, which we may or may not end up carrying out. Most certainly we will look into other, more easily accessible areas of improvement first, and we would also have to make sure that such an algorithm would even be worth the effort. If it turns out to be relatively simple to fix all errors by hand, we will abandon this idea. A crucial prerequisite to this task would be that *all* the errors that the improvements are based on are real errors and not results of strange words or faulty SMOR analyses. This would most likely mean manual double-checking of the errors.

6.7 Test selnolig's English Patterns

For this task, we would probably be able to carry over many parts of our programs, but we would have to adapt others. Obviously, we would need an English corpus and an English morphological analyzer, which would require adaptions to their specific formats. Furthermore, selnolig's treatment of English ligatures is somewhat more complex than that of the German ones, involving a basic and a broader set of patterns, and also looking at many ligatures typically only available in Italics, which lead to interesting cases of conflicting ligatures, i.e. glyphs that could form a ligature with their left neighbor or their right neighbor.

7 Conclusion

Based on these figures and the quality of our results, we can say that selnolig-check has been successful. While our approach to the entire task is pretty straightforward, we encountered many challenges on the way that still made the entire project complex in its details. We had to make essential decisions on the way, e.g. decide which tokens in the corpus we still consider *a word* and which we don't. Most of the code we wrote is only necessary to transform the data our two external resources – SDeWaC and SMOR – provide us with into data we can evaluate. Had we had a single resource perfectly suited for our purposes, i.e. a corpus containing inflected forms of words, including their morpheme boundaries, our task would have been much easier.

However, the “raw” nature of the data we're processing gives us a lot of flexibility and even inspiration, and thus the possibility to pursue further ideas in directions we would not have thought of otherwise (e.g. the differentiation of inflectional and derivational morphemes, or dedicated treatment of Old Orthography, New Orthography, and Swiss German).

Once we have updated selnolig-check to be able to test the selnolig-patterns that are truly current, it might be possible to rule out *every* known error in the patterns, thanks to `\keeplig` . The goal of our testing of selnolig and our contributions to its interface and innards is to make it a package whose loading will be a no-brainer for LuaLaTeX users, just like the popular microtype[1] package is for pdfLaTeX. If selnolig once reached this point of being part of the daily workflow of hundreds or even thousands of people, and if we had contributed to that, we would be very proud.

[1]http://www.ctan.org/pkg/microtype