the CST Lemmatiser

version 3.0 (10 December 2008)

Bart Jongejan & Dorte Haltrup

Center for Sprogteknologi, University of Copenhagen

December 2008

Center for Sprogteknologi
University of Copenhagen
Njalsgade 80
2300 Copenhagen S.
Denmark

http://www.cst.dk

Contents

1. Introduction

The CST lemmatiser has been developed under the STO project in 2002. The motivation for developing a lemmatiser was the need to 'handle' new text in order to gather and select domain specific words for the STO database. Since the STO database is corpus based, the selection of new words (lemmas) is based upon frequency and to make a proper frequency calculation, lemmatisation is essential.

Our goal and intention has been to make as accurate and flexible lemmatisation as possible. The need for accuracy means that average stemming or truncation is too gross – we needed rules for regularity as well as exceptions. For this purpose we used the around 50.000 lemmas, with their respective inflected forms, from the general vocabulary that already existed in the STO database. The rules of the CST lemmatiser have been created, or rather learned, from the function between lemma and inflected form of these 50.000 lemmas. The demand for flexibility has arisen from a wish to make the CST lemmatiser usable in different applications and is shown in the extensive list of input and output formats.

The result is a trainable lemmatiser with a variety of functions. It is language independent in the way that it can be trained for different languages, or at least for languages with inflectional suffixes, but not with inflectional prefixes like German[1]. What is needed is a list of lemmas, their infected forms and, if possible, their POS-tag.

This report contains a description of the function of the CST lemmatiser.

[1] A new training algorithm has been implemented to create rules that can address inflection in any place, not just in suffixes or prefixes. Although these rules can be applied by the CST lemmatiser, the training of these rules is done in another program.

2. The CST lemmatiser program

In this chapter we will describe in some detail what the CST lemmatiser does, and why. We will describe what is needed, what can be expected from the output and how the process, the algorithm, functions.

2.1. Input

The input to the CST lemmatiser can either be a text or a list of words and in addition it can be with or without POS-tags. The best lemmatisation is, of course, done with POS-tags because the tags are used to disambiguate homographs in the dictionary, and to control what flex rules to use for new words. If POS-tags are present then the default separator between word form and POS-tag is "/" (slash). If you want another input format, you have to specify it on the command line (see chapter 4).

To get a proper lemmatisation it is essential that the text is tokenised so that punctuation marks are separated from words.

2.2. Processing

The functionality of the CST lemmatiser can be divided into two fields: lemmatisation and training.

The main function of the lemmatisation is of cause to find the proper lemma for each word in the input. But there are a variety of subfunctions: it can produce frequency lists, lists of conflicts between POS-tag in the input and in the dictionary and it can work with or without a dictionary. In the training process the main function is to create new flex rules.

It is the main functions that are described below.

2.2.1. Lemmatisation

In our approach to lemmatisation there are three main tasks:

i) Finding lemmas for known words

ii) Disambiguating if the known words are a homographs

iii) Guessing lemmas for unknown words

Figs. 1-3 show how the algorithm solves these three tasks.

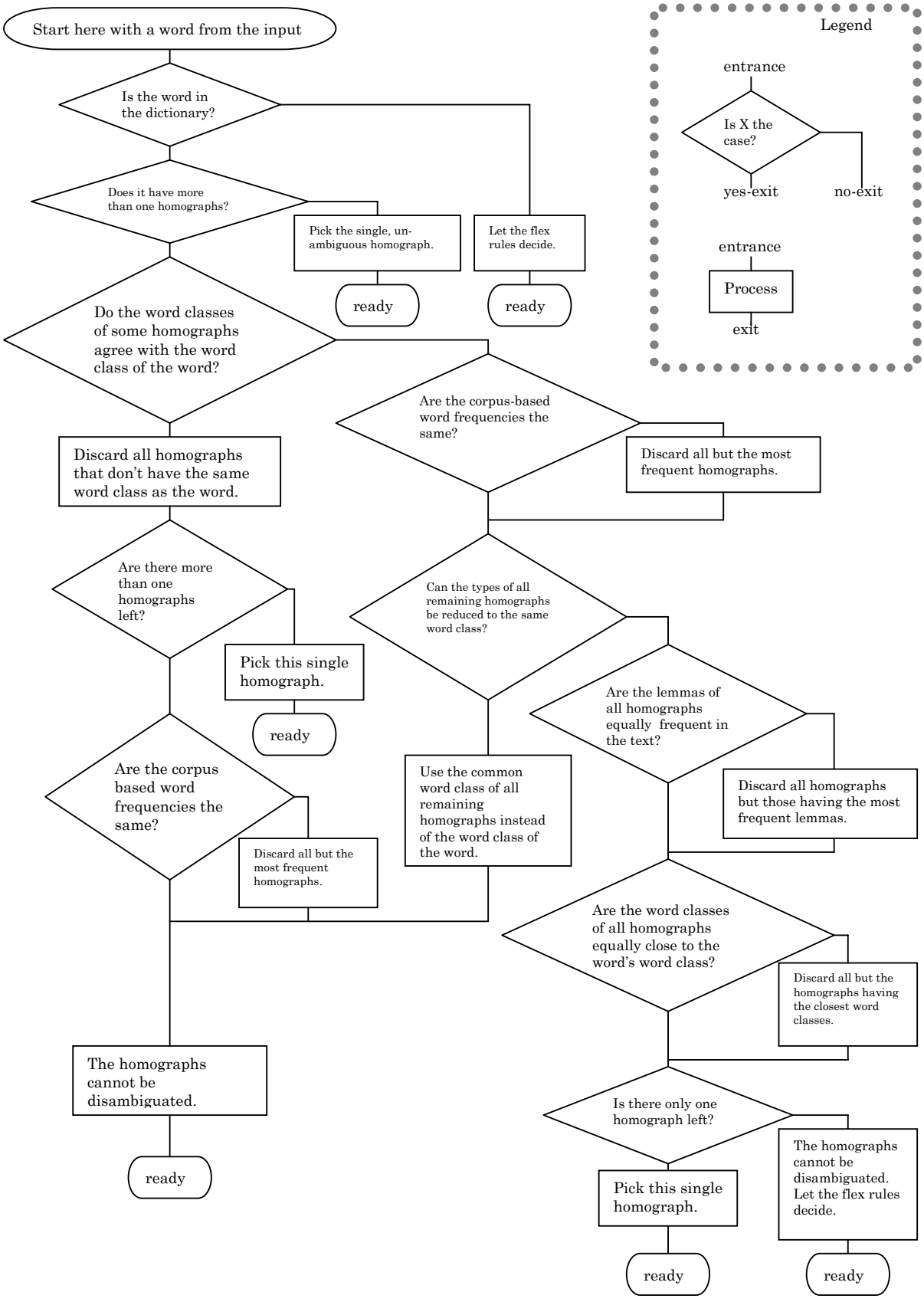

Figure 1: Disambiguation of POS-tagged words

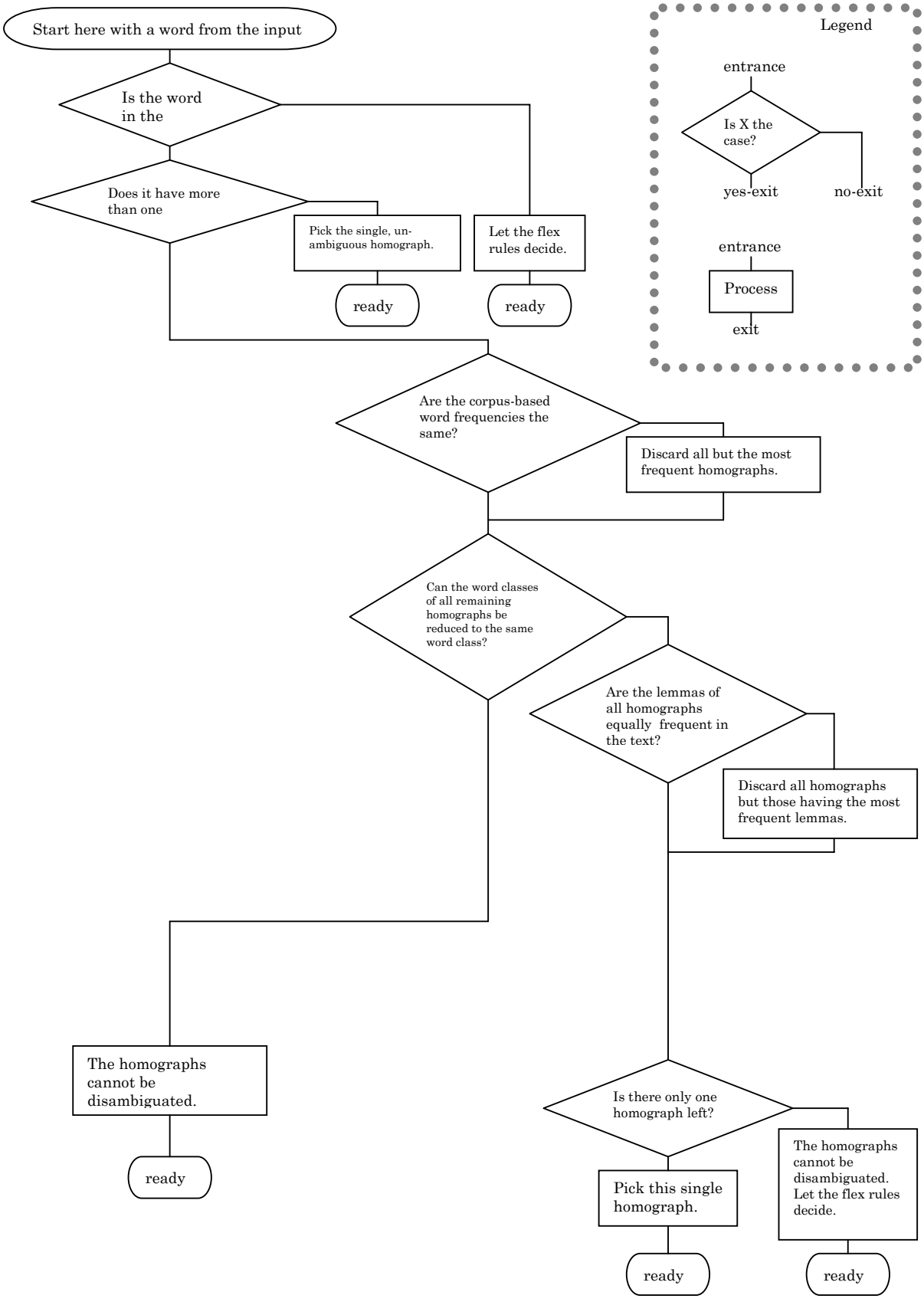

Figure 2: Disambiguation of words that are not POS tagged

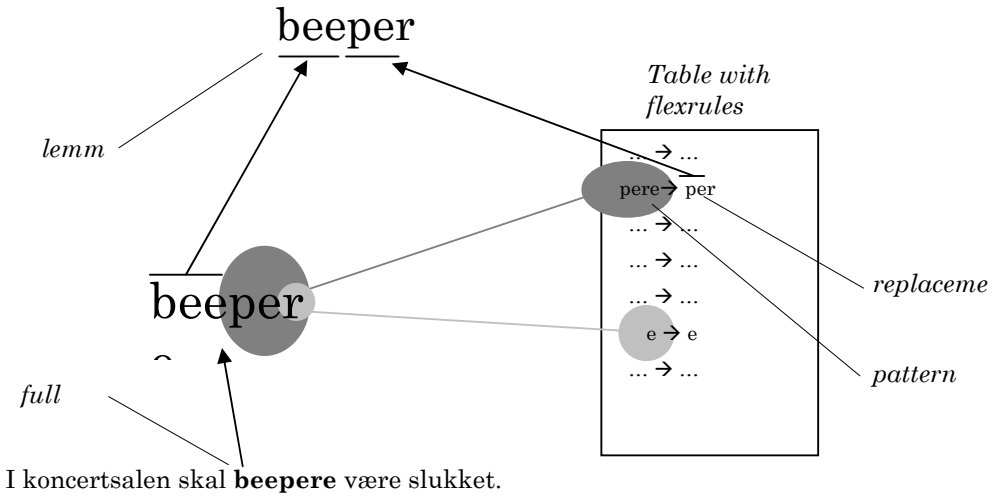

*Figure 3: Lemmatisation using flex-rules. The longest matching pattern **(pere)** prevails over all shorter matching patterns **(e)**.*

Straightforward is the case where a word from the input stream, according to the dictionary, has exactly one homograph that matches both the word form and the POS-tag of the word in question. Also quite straightforward is the case where the dictionary lookup has no result at all: in that case the flex rules can easily produce an unambiguous result.

More complicated is the case where a word has more than one homograph, each homograph having a different lemma. If the two (or more) alternative lemmas belong to different word classes, the POS-tag from the input is used to disambiguate. If, on the other hand, the alternative lemmas belong to the same word class, the most probable word form is chosen instead. In the Danish dictionary frequency information derived from the tagged Parole corpus is used to estimate a word's probability. If frequency isn't enough (eg. the word forms don't exist in the Parole corpus and hence frequency information is lacking), then a probability estimate is made on the input text and the most frequent lemma is chosen. As a final solution the flex rules are used, ignoring the conflicting lemmas from the dictionary altogether.

Another type of ambiguity arises when the POS-tag from the input doesn't correspond to the word class (-es) in the dictionary. Our choice here is to let the dictionary overrule the input tags. The reason is that the dictionary is made manually and is therefore more reliable than the inputtags generated by an automatic tagger. If the dictionary presents alternative lemmas, the same procedure as for homographs is used, with the additional heuristic that if none of the aforementioned selection criteria leads to complete disambiguation, the input POS-tag is compared with the word class of each candidate homograph. Although none of the word classes matches exactly with the POS-tag, some word classes may be more related than others. The closest match points at the winning homograph.

Another approach to lemmatisation is not to discriminate between known and unknown words, ambiguous and unambiguous words. Then flexrules will generate all lemmas. In our application this approach is actually a side effect – our lemmatiser can work by the rules alone and thereby function as an advanced stemmer. This kind of stemming is a bit quicker and a bit more inaccurate than lemmatisation.

We made a test using the CST lemmatiser with and without the dictionary and with and without POS-tags. The test material was 250.000 words from the Parole corpus. Here are the results:

	Correct lemmas	Time
Input *with* POS-tags Lemmatisation *with* dictionary = **real lemmatiser**	97,8 %	App. 1 min.
Input *without* POS-tags Lemmatisation *with* dictionary = **discount lemmatiser**	94,5 %	App. 25 sec
Input *with* POS-tags Lemmatisation *without* dictionary = **good stemmer**	97,4 %	App. 48 sec.
Input *without* POS-tags Lemmatisation *without* dictionary = **stemmer**	88,4 %	App. 30 sec

Among the 250.000 words 24% were unknown to the dictionary. As the schema shows, the CST lemmatiser is doing very well. It seems surprising that the presence of POS-tags makes a bigger difference to the result than the dictionary. The reason is first of all that the flex rule are created from the dictionary, secondly that the rules work according to our intentions, and finally that information about POS is a big help in guiding through the flex rules. In the best result with the *real lemmatiser* half of the errors made are caused by adjectives, a word class still underrepresented in our training material.

2.2.2. Training

As mentioned in the introduction the CST lemmatiser can be trained to different languages. The training material is a dictionary containing lemmas, their respective full forms and the POS-tags of the full forms. From this dictionary the algorithm creates a list of flex rules for each POS-class. Training the Danish version on 450.000 word forms, corresponding to 60.000 different lemmas, created 44.000 different flex rules.

Below is a list of word forms, their corresponding lemmas and the simplest rules, which could map the word forms to their lemmas:

	Word form	**Lemma**	**Rules**
1)	billederne	billede	-rne[+0]
2)	håndteringerne	håndtering	-erne[+0]
3)	forskerne	forsker	-ne[+0]
4)	politikkerne	politik	-kerne[+0]
5)	aftrækkerne	aftrækker	-ne[+0]
6)	bagsmækkerne	bagsmæk	-kerne[+0]
7)	fluesmækkerne	fluesmækker	-ne[+0]

These rules work for each word-/lemma pair in isolation, but in general they are insufficient because there is no information about which rule to use for a given word form. The first rule e.g. could be applied to all 7 words but with a wrong result. We need more information about the words to which the rules are applicable, and to get this information we need to look deeper into the words.

In the training process there is a meta principle guiding the process saying: only use the longest matching rule. In the case above '-rne[+0]' is applied to 2), but fails to produce the correct lemma. Then a new rule is created which must be longer than the former and we get '-erne[+0]'. This process continues throughout the training material in a way that the longest existing rule that match the word form, is applied, and if it fails to produce the correct lemma, a new and longer rule is created. The process can be illustrated by the following rules:

	Word form	**Lemma**	**Rules**
1)	billederne	billede	-rne[+0]
2)	håndteringerne	håndtering	-erne[+0]
3)	forskerne	forsker	-kerne[+ker]
4)	politikkerne	politik	-kkerne[+k]
5)	aftrækkerne	aftrækker	-ækkerne[+ækker]
6)	bagsmækkerne	bagsmæk	-mækkerne[+mæk]
7)	fluesmækkerne	fluesmækker	-smækkerne[+smækker]

The creation of flex rules continues by repeated iterations of the training material until all word forms are correctly lemmatised. Finally rules that have become redundant during the process are removed. The final rules for the abovementioned words are:

	Word form	**Lemma**	**Rules**
1)	billederne	billede	-llederne[+llede]
2)	håndteringerne	håndtering	-eringerne[+ering]
3)	forskerne	forsker	-orskerne[+orsker]
4)	politikkerne	politik	-kkerne[+k]
5)	aftrækkerne	aftrækker	-aftrækkerne[+aftrækker]
6)	bagsmækkerne	bagsmæk	-mækkerne[+mæk]
7)	fluesmækkerne	fluesmækker	-uesmækkerne[+uesmækker]

After the training it is possible to remove unwanted rules from the flex rule file. In Danish, we found that irregular inflection of proper names created wrong lemmas for most names. To avoid this we removed all rules for proper named except those for genitive.

2.3. Output

The default output format for the basic CST lemmatiser is word form/tag/lemma:

Example:

Input: Fordommene/N slører/V_PRES hans/PRON_POSS vurdering/N af/PRÆP sagen/N

Output:
Fordommene/N/ fordom
slører/V_PRES/ sløre
hans/PRON_POSS/ hans
vurdering/N/vurdering
af/PRÆP/af/
sagen/N/sag

In general the CST lemmatiser can be instructed to either focus on listing full forms or lemmas. The first, in turn, can focus on tokens, meaning all words in a text, or on types, meaning all different word

forms in a text. Either way, the lemmatiser can provide information on both in its output. Thus, it is possible to produce:

(i) Output corresponding to the input text, enriched with information on each word's lemma, shown as the default output above. This token oriented output is especially useful if the context of each word needs to be preserved.

(ii) Output as a sorted list of the words that occur in the text, together with information on each word's lemma. This type oriented output, which sorts and merges the input so that each full form occurring one or more times in the input only occurs once in the output, is useful if the main interest is in creating a full-form-to-lemma-"dictionary".

(iii) Output as a sorted list of lemmas that are represented one or more times by full forms in the text. This lemma-oriented output is useful if one wants to know which lemmas are occurring in the text at all. Each lemma in the list can be enriched with a full listing of the full forms leading to the lemma after lemmatisation, thus giving a good impression of which inflected forms of a lemma occur in the text.

All forms of output can be enriched with frequency information, telling how often a full form occurs in the text and how often a lemma (disregarding any inflections) is represented in the text.

As mentioned the CST lemmatiser has two resources that it can use to determine the lemma of a given word: a built-in dictionary and a list of flex rules. The lemmatiser can show the results of both resources and the user can define which result or results should be selected for output.

In section 2.2.1 we described how the CST lemmatiser applies some heuristics to solve ambiguous cases. Some of these heuristics can be activated by setting options on the command line: the flex rules can be forced to only output unambiguous solutions whereas the dictionary mechanism can be instructed to prune solutions away using word frequencies from a "standard" corpus, in the Danish case the Parole corpus, or, as a last refuge, by looking at the lemma's frequency in the input text.

3. Controlling the output format

3.1. Introduction

If we don't take measures to suppress ambiguous dictionary look-ups and ambiguous flex rule applications, a word may have any number of lemmas. Also, if the program uses a dictionary, it is possible that a word isn't found in the dictionary at all. In general, it is desirable to be able to steer the output so that certain phenomena can be scrutinized more easily. For example, it might be nice to always have just a single solution to a lemmatisation, whatever the amount of ambiguity that the program encounters. This could be done by presenting the solution obtained by dictionary look-up if there is exactly one such solution and to present the flex rules' solution in all other cases. As another example, we might be interested to only see those words in the output that gave rise to ambiguity, or to see only those lemmas that are represented by five inflected forms in the text. These examples show that the program must allow the user to define *which items* are shown in the output, under *which conditions* they must be shown and *how* they must be shown.

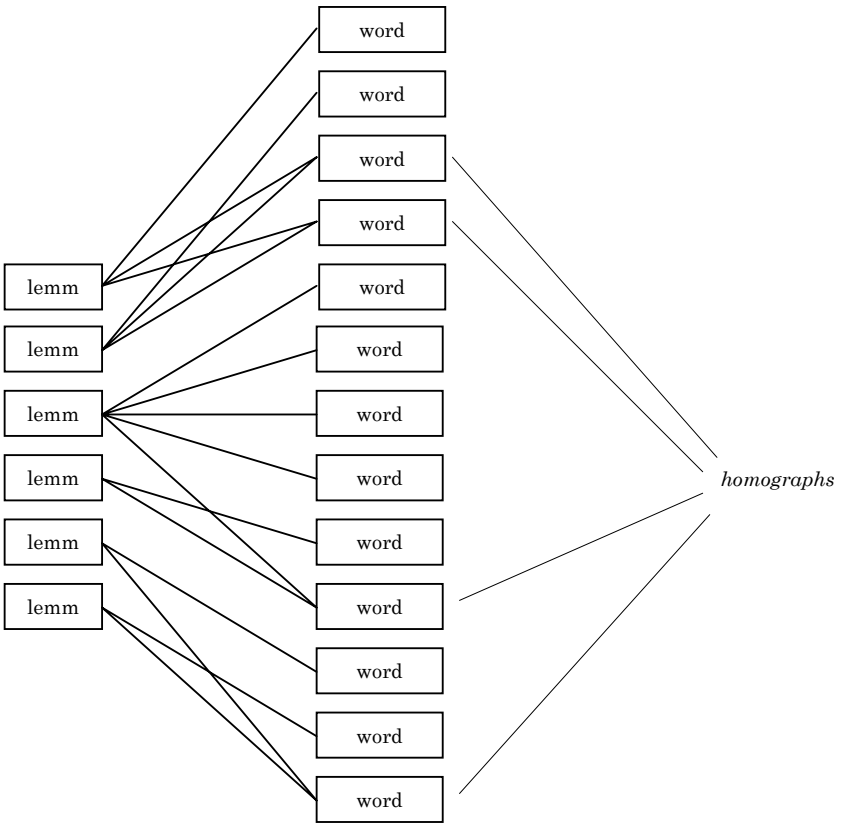

Figure 4. Internally, each word form occurring in the input has one or more lemmas, originating from the dictionary or from application of the flex rules. And conversely, lemmas have typically more than one word form.

Basically, the output consists of lemmas or contains lemmas, among other items. These other items are first and foremost the original word form and the POS-tag of the word form, but also statistics and lay-out elements can be written into the output. If the output is to contain both lemmas and word forms, we have the complication that a lemma can have more than one word form and that a word form can have more than one lemma (see fig. 4).

As we cannot both list all word forms for each lemma and all lemmas for each word form in a meaningful way, we have to choose.

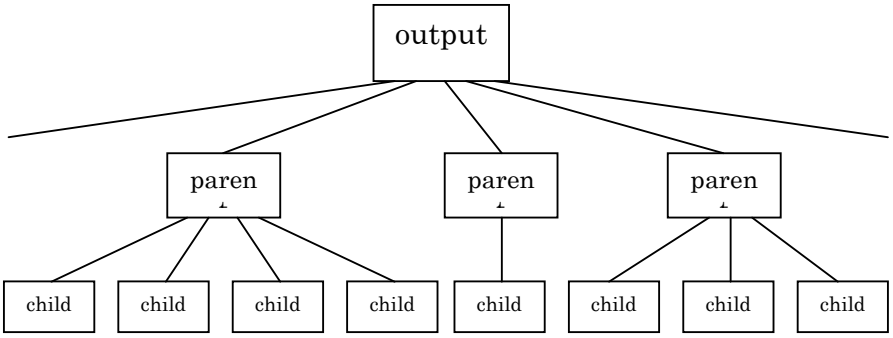

Figure 5. General format of the output.

Thus, either the output lists the lemmas that occur in the text and, for each lemma, lists the word forms (=children) that belong to the lemma (=parent), or, alternatively, the output lists the word forms as they occur in the text and lists the possible lemmas (=children) for each word form (=parent).

The CST lemmatiser can use several strategies for reducing the number of lemmas per word form. In fully disambiguated output each word form has exactly one lemma. On the other hand, a lemma in fully disambiguated output can still have more than one word form. Therefore, fully disambiguated output will look like in figs 6a and 6b.

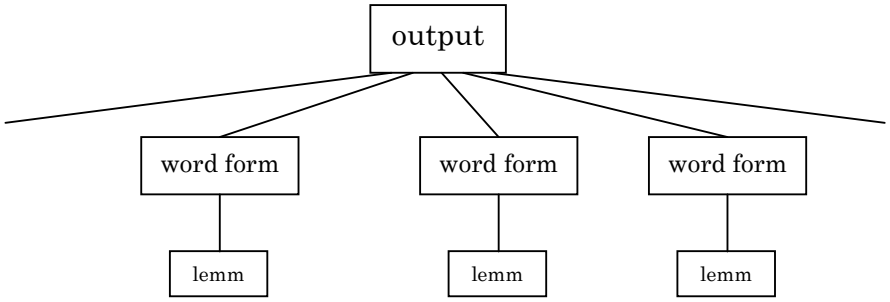

Figure 6a Disambiguated output. One lemma per word form.

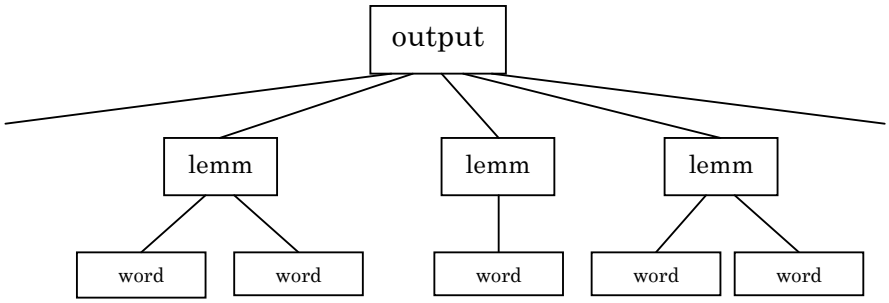

Figure 6b Disambiguated output. Possibly more than one word form per lemma.

Disambiguation can basically take place in three ways:

1) If dictionary look-up has more than one results, then some results may be discarded for some reason.

2) The set of flex rules can be pruned for all ambiguous rules.

3) If dictionary look-up has more than one single result, then we can use the (disambiguated) flex-rules instead.

Whereas 1) and 2) are governed by command line options dedicated to several heuristics, 3) is a kind of disambiguation that is effectuated by carefully formulating how the output must be shaped. Thus, it is possible to program the output in the following way:

```
If there is just one candidate lemma from the dictionary
Then output the lemma from the dictionary.
Else output the lemma as computed by the flex rules.
```

If all ambiguous rules are removed beforehand, then this program would produce fully disambiguated output.

Notice that disambiguation attempts (those that are commanded by command line options) take place before such output rules are applied. So the lemmatiser will first try to disambiguate dictionary look-ups on the basis of, say, corpus frequencies and then it may decide to use the flex rules after all. The programmability of the output makes it possible to format the output in many other ways, for example

```
If there are more than one candidate lemmas from the dictionary
Then output these lemmas
Else do not produce output at all.
```

Or

```
If there are more than one candidate lemmas from the dictionary
Then output these lemmas and the flex rule lemmas
Else only output the dictionary lemma
```

Or

```
If there are no candidate lemmas from the dictionary
Then output the flex rule lemmas
Else don't output anything.
```

As these examples have made clear, there is a need to be able to count the number of children data (lemmas candidating for a word, words candidating for a lemma) and to be able to decide what to output, based on these numbers. This is achieved by defining format strings on the command line; one or two format strings for the way a child must be formatted and one or two format strings for the way a parent must be formatted. It is only in the parent format string that conditions can be tested and decisions can be made as to what to output, based on the results of these tests. If the output must list all word forms for each lemma, then we use the –B or -b command line option to describe the parent format (the lemma) and the –W command line option to describe the child format (the word forms belonging to the lemma). If the output must list all word forms and the lemmas for each word form (whether or not disambiguated), then we use the –c command line option to describe the parent format string (the word form) and the –B or –b command line options to describe the child format strings (the lemmas). The syntax of these format strings is described in the next section.

3.2. Syntax of –c, -b and –B parent format strings

Note: Angled brackets and vertical bars (*OR*) are part of the syntax notation. Expressions between angled brackets denote non-terminal nodes. Italicised text is comment. Longer comments are put in footnotes.

format ::= <quote><expression><quote>[2]
quote ::= " *double quote (Windows)*
' *single quote (Unix, Linux)*
expression ::= <expression element><expression>
expression element ::= <countable[3] expression element>
| <uncountable[4] expression element >
countable expression element ::=
<invisible countable expression element>
| <countable block>
| <countable field>
invisible countable expression element[5] ::=
[<countable expression element>]<hide tag>
uncountable expression element ::=
<uncountable block>
| <uncountable field spec>

---

[2] The quotes can be left out if there are no blanks in the format string.

[3] "Countable", in this context, is a compound concept. A countable expression has the property of a numeric result, but also the property that it can succeed or fail. For example, the expression $b<5 succeeds if there are k=1,2,3 or 4 instantiations of the field b. Its numeric result is the value of k. If, however, k = 0 or k > 4, then the expression fails and the number k is rendered irrelevant. A countable expression also has the property that it appears in the output, unless it is made invisible with the hide tag '?'.

[4] Uncountable expressions elements, in contrast to countable expression elements, have no numeric result or success/failure property.

[5] Invisible countable expression elements do not appear in the output. They can be used to decide the visibility of other expression elements.

	| <literal expression>
literal expression ::=	*a string of printable characters*
uncountable block[6] ::=	[<countable expression>][7]
countable expression ::=	<countable expression element> <expression>
	| <expression element ><countable expression>[8]
countable block ::=	[<countable expression>]<test>
countable field ::=	<countable field spec>
	| <countable field spec><test>
countable field spec ::=	$[9] <countable field name>
uncountable field spec ::=	$ <uncountable field name>
countable field name ::=	b[10]
	| B[11]
	| W[12]
uncountable field name ::=	f[13]
	| s[14]
	| t[15]
	| w[16]

---

[6] Uncountable blocks are used to tie the visibility of uncountable expression elements to the success of fallible expression elements

[7] The square brackets do not appear in the output. To make sure that square brackets appear in the output, use the escape character '\' in front of the bracket: \[\]

[8] Countable expressions can have more than one countable expression elements. For the countable expression to succeed, all its countable expression elements must succeed.

[9] The dollar sign does not appear in the output. To write a dollar sign, use \$.

[10] $b is the field that expands to all lemmas that result from looking up the full form in the dictionary. It can only occur in the –c format and needs the specification of a –b format.

[11] $B is the field that expands to all lemmas that result from applying the flex rules. It can only occur in the –c format and needs the specification of a –B format.

[12] $W is the field that expands to all full forms that were lemmatised to one and the same lemma, either by dictionary look-up (if occurring in the –b format) or flex rule application (if occurring in the –B format). It needs the specification of a –W format.

[13] $f is the field that returns the number of times a full form occurs in the input text. If used in the –b and –B formats, it is the number of times the lemma (lemma) occurs in the text in any of its inflected forms (full forms).

[14] $s evaluates to either a new line character or a blank, depending on whether the current word is the last word before a line break or not. This field can only meaningfully be used in the –c format with sorting turned of (-q-). $s always evaluates to blank if the output is sorted (-q or –q#) or when specified in the –W format.

[15] $t expands to the word class of the word ($w). If used in the –c or –W format, $t is the word class of the full form. If used in the –b and –B formats, $t is the word class of the lemma, which is the same as the word class of the full form, unless there is an entry in the type conversion table (-z command line option) that transforms the full form word class to another type.

test ::= <op>non-negative whole number
| <simple test>
simple test ::= +[17]
| *[18]
| non-negative whole number[19]
op ::= <[20]
| >[21]
| ~[22]

hide tag ::= ?

3.3. Syntax of –b, -B and –W child format strings

format ::= <quote>< uncountable expression><quote>
quote ::= " *double quote (Windows)*
' *single quote (Unix, Linux)*
uncountable expression ::= <uncountable expression element><expression>
uncountable expression element ::=
<uncountable field spec>
| literal expression
uncountable field spec ::= $ <uncountable field name>
uncountable field name ::= f
| t
| w

3.4. Examples of format strings

For example, if we want to output only those lemmas that are inflected in exactly five ways in the text, define the following –b (or –B) format:

-b '[$w [{$W}]5\n]'

As we are using the $W field, we also need to specify the –W format:

-W '$w'

Explanation:

---

[16] $w is the word form. If used in the –c or –W format, $w expands to the full word. If used in the –b and –B formats, $w expands to the lemma.

[17] + tests for the presence of one or more instantiations.

[18] * always succeeds, it tests for the presence of any number of instantiations.

[19] A number k without an operator indicates that the expression succeeds if there are exactly k instantiations and that it fails otherwise.

[20] <k succeeds if there are less than k instantiations and fails if there are k or more instantiations.

[21] >k succeeds if there are more than k instantiations and fails if there are k or less instantiations.

[22] ~k fails if there are exactly k instantiations and succeeds otherwise.

[…..] We only want to generate output under certain conditions. These conditions are somewhere between the outer square brackets.

[….\n] If there is generated output, its final character must be a new line character. Thus, we are generating one line per output item.

$w A $w in a –b or –B format represents the lemma or lemma. Thus, the first thing on a line of output will be a lemma. In the –W format, $w stands for a full form of a word.

$w […] Between the 'w' and the opening square bracket is a blank. This blank is copied literally to the output.

[…..]5 Here we have the expression that the outer square brackets can test. If there are exactly five occurrences of something between the inner square brackets, then this expression succeeds. Also, all five instances of the 'something' are copied to the output line.

{…} Like the blank that we saw before, the characters { and } are copied to the output, but only if the condition that surrounds them is met. Also, only one { and one } are copied, even though there is something that is output five times.

$W This is the 'something' that can occur a variable number of times. It is an item that tells that at this place something must be output, but it does not specify how. The 'how' is specified in another command line argument, the –W format. As seen above, the –W format specifies that just the full form is output. If there are more than one instances of $W, then the instances are separated by a vertical bar '|'. The separator can be re-specified with yet another command line argument, -s.

Let us have a look at another format. We want output that, for each word in the text, shows exactly one lemma, followed by a blank. Although we are not going to have full forms in the output, what we want is still token-or type-oriented, as opposed to lemma oriented. So we use the –c format to have a token- or type- oriented handling and the –b and –B formats to have access to the lemma information of each token or type. The –c format will look as follows:

```
-c ' [$b0$B][$b1][[$b>1]?$B] '

-b '$w'

-B '$w'
```

Explanation:

'...] ' Between the closing bracket and the quote is a blank. Thus, each piece of output is concluded by a blank.

[…][…][…] There are three parts that, depending on the circumstances, can end up in the output. Of course, these parts correspond to the circumstance that we have no successful dictionary look-up, that we have an ambiguous dictionary look-up that cannot be remedied by disambiguation heuristics and that we have a dictionary look-up resulting in exactly one homograph. At face value, the three blocks could provide output independent of whether the other blocks do or not, but by looking at the conditions that are tested, we can conclude that one and only one block 'fires'.

[$b0$B] This block tells that the flex rule solution $B must be output if the dictionary look-up failed to return a homograph. The brackets surrounding $b0$B makes $B dependent on the success of $b0.

$b1 This is a short-hand notation for [$b]1. It means that the field $b is copied to the output if there is exactly one instance of it. $b encapsulates the data that are obtained by dictionary look-up. Thus, if dictionary look-up gave exactly one homograph, then $b1 succeeds. The brackets surrounding $b1 can, in fact, be left out.

[[$b>1]?$B] This block tells to use the flex rules' solution instead of the homographs obtained from the dictionary. This is a last resort way to evade ambiguity.

[…]? This expression suppresses the output from the bracketed expression.

The same result can be obtained using a slightly simpler –c format, testing the number of dictionary solutions two instead of three times:

```
-c '$b1[[$b~1]?$B] '
```

Much more complex formats can be specified. For example, the following format creates output if a lemma is represented by three or four inflected forms:

```
[$w [[{$W}]<5]>2\n]
```

That is, the inner test [{$W}]<5 percolates the number of instances of $W to the surrounding brackets […]>2. If the inner test succeeds, the outer test is applied. The outer test fails if the inner test fails or if there are two or less instances of $W. The same effect could be obtained by e.g.

```
[$w {$W}[$W?]>2[$W?]<5\n]
```

The last example shows that a […] block only creates output if all conditions are met. We could use this to find all words that are lemmatised ambiguously by dictionary look-up as well as flex rule application (turn flex rule disambiguation off, -U-).

```
-c '[$w dictionary( [$b]>1) flex rules([$B]>1)\n]'
```

4. Command line options

4.1. Introduction

The CST lemmatiser is, in fact, three programs in one. Besides its main purpose, lemmatising a text, it has two subordinate functions, namely the creation of resources that are used by the lemmatising process. The resources created by the program are

- A machine-readable dictionary for quick look-up of words
- A set of flex rules for lemmatising words not found in the dictionary.

In the paragraphs describing these functionalities, the following notation is used:

-x option letter x

-x<arg> option x requires an argument. The argument may be separated from the option letter by blanks.

[opt] "opt" is an optional option (sic)

<descr> a variable, to be replaced by a value in a concrete command line.

Important note: Unix and Windows use different methods for keeping command line arguments together that contain blanks: In Windows, such arguments must be surrounded by double quotes. In Unix, arguments must be enclosed in single quotes. The examples below use single quotes (Unix format).

4.2. Create binary dictionary

4.2.1. Command line

```
cstlemma -D -c<format> [-N<frequency file> -n<format>] [-i<lemmafile>] [-o<binarydictionary>]
            [-y[-]] [-k[-]]
```

4.2.2. Options

-D This option letter instructs the program to create a binary dictionary from a list of lemmas. Each line in the lemma file must contain a full form and a lemma.

-i<lemmafile>

Name of the dictionary (lemma) input file.

-c<format> Column format of dictionary file (tab separated). You must use the letters 'F', 'B' and 'T' once. 'F' and 'B' indicate the columns of full forms and lemmas respectively. 'T' indicates the column containing the word class of the full form.

Example:

-cFBT

which means: 1st column = full form, 2nd column = lemma, 3rd column = type

klon	klon	N
klon	klone	V_IMP

klone	klone	V_INF
klonede	klone	V_PARTC_PAST
klonede	klone	V_PAST
klonedes	klone	V_PAST
klonen	klon	N
klonen	klone	V_GERUND
klonende	klone	V_PARTC_PRES
klonens	klon	N_GEN
kloner	klon	N
kloner	klone	V_PRES
klonerne	klon	N
klonernes	klon	N_GEN
kloners	klon	N_GEN
klones	klone	V_INF
klones	klone	V_PRES
klonet	klone	V_PARTC_PAST

-e<n> Encoding to adopt for case conversion. n=1: use ISO8859-1 encoding (Western European, default). Other encodings: 2 (Central European), 7 (Greek) and 9 (Turkish).

-eU Unicode (UTF8) input. *No case conversion is attempted during creation of the dictionary.*

-e Do not apply case conversion.

-N<frequency file>

The name of a file that contains frequency information of full forms, as extracted from a corpus. This file may also contain lemma frequencies. You can specify one or more frequency files. For each frequency file you must specify a column format with a –n option specification. The first –N specification is paired with the first –n specification, the second with the second, etc.

-n<format> Column format of frequency file (tab separated). The column indicated by 'N' contains a full form's frequency. The column indicated by 'F' indicates the full form. The column indicated by 'T' contains the word class of the full form. Optionally, you can indicate a column containing lemmas with the letter 'B'. You can use '?' to cancel out columns containing irrelevant data. You can have both frequency files containing lemma-information and frequency files not containing this information If no lemma-information is present, all lemmas compatible with a given full form are incremented with the frequency data.

Example:

-nN?FT

which means: 1st column N(frequency), 2nd column irrelevant, 3rd column F(ull form), 4th column T(ype).

1533	4.790625	reception	N
1532	4.7875	oprindelige	ADJ
1531	4.784375	tanker	N
1531	4.784375	tabt	V_PARTC_PAST
1531	4.784375	længst	ADV
1531	4.784375	etniske	ADJ
1529	4.778125	serberne	N
1526	4.76875	voldsomt	ADJ

-o<binarydictionary>

Name of the output file. Notice that the output file is a binary file. It is, in general, not portable between operating systems and hardware platforms.

-y test output
-y- production output (default)

-k collapse homographs (remove ",n" endings)(default)
-k- do not collapse homographs (keep ",n" endings)

4.3. Create or add flex patterns

The cstlemma program supports two types of flex patterns when lemmatizing, but the program can only train one of these types. The instructions for generating this type of flex rules (called suffix rules hereafter) are in this section. The other, newer type of flex rules (called affix rules hereafter) are trained with a product that is not part of cstlemma and are still experimental (2008). The suffix rules only support 8 bit encodings like Latin-1, Latin-2, Latin-7 and Latin-9. The affix rules also support Unicode (UTF-8).

4.3.1. Command line

```
cstlemma -F -c<format> [-y[-]] [-C[-|n]] [-R[-]] [-i<lemmafile>] [-f<old flexpatterns>] [-o<new flexpatterns>] [-e<encoding>]
```

4.3.2. Options

-F This option letter instructs the program to build a list of flex rules from a list of lemmas. As minimum, each line in the lemma file must contain a full form and a lemma. If the word class of the full form is provided, then a better list of flex rules can be made, but it can only be applied to POS-tagged texts, preferably using the same tag set as the lemma list.

-c<format> column format, e.g. -cBFT, which means: 1st column B(aseform), 2nd column F(ullform), 3rd column T(ype)

For lemmatising untagged text, suppress word class information by specifying '?' in place of 'T'

-i<lemmafile>

Name of the input dictionary (lemma) file. This can be the same file as used for the creation of the binary dictionary.

-f<old flexpatterns>

By specifying this option you can refine an existing flex rule file (the argument of the –f option).[23]

-o<new flexpatterns>

Name of the output file. The flex rule file is a text file and can be post-edited manually, for example for removing an unwanted group of rules. The flex rule file can safely be ported between operating systems and hardware platforms, which is nice, because generating the flex rules is a time consuming process.

[23] This option hasn't been used for a long time and its functioning is not guaranteed.

-y Test output
-y- Release output (default)

-C<n> n=0,1, … Include only rules that have support from at least *n* words in the input file. (This vastly reduces the size of the output file. Use it if there are signs of overfitting.)
-C- Include all generated rules in the output flex rule file (default).

-R For each rule in the output, add information about the number of words tha support the rule. (This number is not used in any way during lemmatisation.)
-R- Don't add these numbers.
-e<n> Encoding to adopt for case conversion. n=1: use ISO8859-1 encoding (Western European, default). Other encodings: 2 (Central European), 7 (Greek) and 9 (Turkish).
-eU Unicode (UTF8) input. *Do not use this option, the suffix-style of rules are not Unicode-ready.*
-e Do not apply case conversion.

4.4. Lemmatise

4.4.1. Command line

```
cstlemma [-L] -c<format> [-d<binarydictionary>] -f<flexpatterns> [-b<format>] [-B<format>]
         [-W<format>] [-s[<sep>]] [-u[-]] [-U[-]] [-v[-]] [-x<Word class translation table>] [-e<n>]
         [-v<tag friends file>] [-z<type conversion table>] [-i<input>] [-o<output>] [-m<number>]
```

4.4.2. Options

-L This option instructs the program to lemmatise (as opposed to creating a machine-readable dictionary or a set of flex rules). Optional (Lemmatising is the default behaviour of the program).

-i<input> Specification of input text. Only needed if input isn't standard input. The CST lemmatiser can handle tagged as well as untagged text, depending on the setting of the –t option

-I<format> Specification of deviating input format. Without this option, the CST lemmatiser assumes, depending on the setting of the –t option, that the input is a flat text or a text consisting of word-tag pairs where the word is followed by its tag, separated by a slash. With this option, more complex input text can be read. For example, -I'$d\t$w\t$t\n' reads a text that represents one word per line, a line containing the word's lemma, its full form and its tag, in this order and each element separated from its neighbours by a tab. The meaning of the format string is as follows:

$w	captures the full form of the word, i.e. the element to be lemmatised
$t	captures the POS-tag (the word class as assigned during a previous process, eg. a tagger) of the word
$d	captures en element that we don't use (dummy); in the example above this is the lemma as provided in the input.
\t	matches a tab
\n	matches a new line character

Other elements that can occur are

a literal	matches a string of one or more characters. All characters in the literal expression must occur in the input in the same order and with the same case, otherwise the match fails. If there are more than one occurrences of the literal in the input, then only the first occurrence is matched.

\s	matches any number of white space characters (including new lines and tabs)
\S	matches any number of non-white space characters

-o<output> Specification of the output file. If the input is standard input then the default output is standard output. Otherwise the default output is a file with a name that is constructed from the input file's name by merely appending an extension '.lemma'. The output file's format can be defined with the -b, -B, -c and -W options.

-d<binarydictionary>

The name of the machine-readable dictionary as produced with the -D option set. If no dictionary is specified, then only the flex patterns are used. Without dictionary, wrong tags in the input cannot be corrected. Note that the dictionary is in a format that, in general, is not portable between different operating systems and hardware platforms.

-f<flexpatterns>

Specification of the file containing flex patterns. (see –F option). It is advisable to use different files for tagged and untagged text. Best results for untagged input are obtained if the rules are made without word class information.

-b<format string>

Default:"$w" Output format for data pertaining to the lemma, according to the *dictionary*:

$f	Lemma token frequency in the input text.
$t	word class
$w	lemma
\$	dollar character
\[	Opening square bracket [
\]	Closing square bracket]
\t	tab
\n	newline (especially useful in combination with the –W option.
$W	(List of) full form(s) in the text belonging to this lemma. A lemma can have any number of full forms. This number can be tested in conditions. The $W field can only be used if the –W option is specified, which is incompatible with the –c option. For a fuller description of how to program the output, see chapter 3.

-B<format string>

Default:"$w". Output format for data pertaining to the lemma, as predicted by *flex pattern rules*. See -b

-W<format string>

Default: not present. Use the –W option if you want to have an overview over which lemmas occur in the text and in which appearances. The –W format decides how, for each lemma, the full forms belonging to this lemma should be written. You must use this option in combination with –b, -B or both. If both –b and –B are specified, the two lists are simply concatenated in the output, the list of lemmas according to the dictionary (-b) first. The –b or –B format must contain a field $W for the –W option to take effect. The –W option is similar to the –c option, but $b and $B are not allowed in –W format strings.

$w	a full form of the lemma, as occurring in the input text.	
$t	word class(s) according to dictionary	
$f	full form type frequency	
$i	info:	
	-	full form is not in the dictionary
	+	full form is in the dictionary, but having another type
	(blank)	full form is in the dictionary
\t	tab	

Example: -W"$w/$t"

-c<format string>

default: '$w\t$b1[[$b?]~1$B]\t$t\n' Output format

$w	full form	
$b	lemma(s) according to dictionary. (You also need to specify -b<format>) (If the full form is found in the dictionary and tag=word class, then only one lemma is output. Otherwise all lemmas may output, but see also the –U option)	
$B	lemma(s) according to flex pattern rule. You also need to specify -B<format>.	
$s	word separator: new line character when the current word is the last word before a line break, blank otherwise	
$t	word class(s) according to dictionary	
$f	full form type frequency	
$i	info:	
	-	full form not in dictionary
	+	full form in dictionary, but other word class
	(blank)	full form in dictionary
\t	tab	

-q Sort output alphabetically.
-q- Do not sort output (default).
-q# Sort output by frequency – most frequent lemmas or words first.

-s<sep> Multiple lemmas (-b -B) are <sep>-separated. Example: -s" | ".
-s Multiple lemmas (-b -B) are "|"-separated (default).

-t Input text is tagged (default).
-t- Input text is not tagged.

-U Enforce unique flex rules (default).
-U- Allow ambiguous flex rules.

-u Disambiguate dictionary look-up (default). By using this option, the program is instructed to use corpus based frequencies to disambiguate between two otherwise acceptable lemmas. The corpus based frequencies are stored in the dictionary. Also, if dictionary lookup results in more than one homographs that only differ in a sequence number appended to the lemma, then the program is instructed to merge all homographs into one by dropping the sequence number.

-u- Allow ambiguous dictionary look-up

-Hn Disambiguation based on lemma frequencies in the input text:

n = 0: use lemma frequencies in the input text for disambiguation (default)

n = 1: use lemma frequencies in the input text for disambiguation, show candidates for pruning between << and >>

n = 2: do not use lemma frequencies for disambiguation.

-e<n> Encoding to adopt for case conversion. n=1: use ISO8859-1 encoding (Western European, default). Other encodings: 2 (Central European), 7 (Greek) and 9 (Turkish).
-eU Unicode (UTF8) input. *Only the affix-type of rules fully support Unicode.*
-e- Do not apply case conversion.

-v<tag friends file>:

Use this option to coerce the nearest fit between input tag and the dictionary's word classs if the dictionary has more than one homographs of the input word and none of these has a word class that exactly agrees with the input tag. Format:

{<dict type> {<space> <tag>}* <newline>}*

The more to the left the tag is, the better the agreement with the dictionary's word class.

Example:

ADJ	V_PARTC_PAST ADV V_PAST V_INF
ADJ_GEN	N_GEN V_PRES V_INF
ADV	PRÆP UKONJ

-x<Word class translation table>:

Use this option to handle tagged texts with tags that do not occur in the dictionary. Format:

{<dict word class> {<space> <tag>}* <newline>}*

-z<type conversion table>:

Use this option to change the meaning of $t in -b and -B formats. Without conversion table, $t is the word class of the full form. With conversion table, $t is the word class of the lemma, as defined by the table. Format:

{<lemma type> <space> <full form word class> <newline>}*

Example:

V_GERUND	V
V_IMP	V
V_INF	V
V_MED_INF	V_MED
V_MED_PARTC_PAST	V_MED
V_MED_PAST	V_MED
V_MED_PRES	V_MED
V_PARTC_PAST	V
V_PARTC_PRES	V
V_PAST	V
V_PRES	V

-m<number> Limit the number of words to lemmatise to <number>. If the input contains more than <number> words, then only the first <number> words are lemmatised. If <number> is 0 (zero), then the lemmatiser does not limit the number of words (default).

4.5. Option files

Often-used combinations of option settings can be stored in option files. Instead of writing the options on the command file, you instruct the program to read the option file. The syntax for option file inclusion is

```
cstlemma -@ option file name
```

You can include more than one option files on the same command line. You can have both 'normal' options and option file inclusions on the same command line. You can overrule option settings by specifying them more than once: only the last setting takes effect. You can include option files inside other option files.

In an option file each option occupies one line, but lines can also be empty or contain comments. Comments are introduced by a semicolon and can also follow after an option setting on the same line.