Practical Application of Finite State Morphology

*** an Exercise with the German Plural ***

1 Preparation

Your task during this practical part of the tutorial will be to create a finite state morphology for the **German Plural** (not a simple problem).
On your Desktop you will find a direction called FSM_clt. Open it and you will find some files and software. We have prepared three empty files for you which you should use during the exercise:

- In **lex.txt** you should work on a lexicon for the German plural forms
- In order to deal with the stem alternations by the means of regular expressions, use **rules.txt**
- **script.txt** is where all your stack commands should go.
- Apart from these files, please open the **xfst** — this is your interface that compiles and manipulates your files.

help: If you need any help, there is an example lexicon, a table with regular expressions and list of useful commands at the end of this file.
Within your directory you can also find the slides that we have just presented.
Furthermore: feel free to ask anyone of us!!

2 The Data

This section introduces the data that you should now turn into a finite state morphology.

The German Plural includes a lot of different suffixes added to the stem. Three of them are *-e, -n* and *-er*.
Furthermore, the appearance of these suffixes can cause a vowel alternation within the stem, which is called *Umlaut*. This alternation causes the vowel to become a little bit higher:

- a → ä (or **ae** as we will spell it in the exercise)
- o → ö / oe
- u → ü / ue
- *e* and *i* do not form an *Umlaut*

See the following table for some data, which you should work on.
The German Plural is mostly regular, but the Umlaut will **not always** be there.

word	**-er**	**-n**	**-e**	**Translation**
Maus			Ma**e**use	mouse
Zug			Zu**e**ge	train
Hund			Hunde	dog
Hand			Ha**e**nde	hand
Ratte		Ratten		rat
Raute		Rauten		hash symbol
Natter		Nattern		(kind of) snake
Haus	Ha**e**user			house
Mann	Ma**e**nner			man
Land	La**e**nder			country
Huhn	Hu**e**hner			chicken

some brief advice:

1. First, write a Lexicon that covers the stems and the basic suffixes in **lex.txt**.
2. To change the stem vowel, you will have to write rules with the help of regular expressions in **rules.txt**.
3. Type all the stack commands into your **script** (**script.txt**): this saves you a lot of time.

.... but let's start step by step:

3 The Plural -e

Let's start with the plural *-e* — this way you only have to deal with four stems and can concentrate on the problems coming along with them.

3.1 The Plural -e — starting with the lexicon

First, let us start with a lexicon describing the German plural suffix *-e*:

- Open up the **lex.txt**, create a "LEXICON Root" and include the four singular **forms that have *-e* as their Plural**.
- Second, you will have to have one **continuing lexicon** for singular and plural (+Sg and +Pl).
- Do not forget to create a **Multichar_Symbols** section at the top where you declare the Plural and Singular tags.

Let's see how you did: open the **xfst-interface** and type
read lexc < *lex.txt* and then
print lower-words
you have accomplished your task when you have a singular and a plural-form (which has an -e attached to it) for all of your words.

3.2 Starting the script

Open up your **script.txt** and type
`clear` in the first line and
`read lexc < lex.txt` in the second line.
Everytime you want to reload your lexicon (or any other files) simply type
source script.txt into the xfst-interface.

3.3 Working on the rules for the plural -e

Now we begin to work on the *Umlaut*. Open up **rules.txt**:

- Our first rule should change **a, o, u** into **ae, oe, ue**. We can accomplish this by writing a rule like the following: a − > a e.
  Note: parallel rules like the change of the letters above can be done by inserting a comma in between them: a − > a e , o − > o e ....

- Next, we need a **condition** for this vowel alternations to take place, since we only want them to appear with the plural forms. The best condition would be the plural tag itself. However, in our current lexicon, the +Pl only appears on the *upper* side of the lexicon — in order to be able to use it within the condition sector, we also need it to appear on the *lower* side.
  We accomplish this by **returning to our lexicon** and add the +Pl on the lower side:
  `+Pl:e+Pl #;`
  Now we can use it as a condition...

- Our condition should be something like:
  "a becomes ae iff +Pl follows somewhere".
  Note that there will be other letters between the vowel and the +Pl. You should therefore add the ?* (0 to infinite number of elements) between the place-holder _ and your condition "+Pl" (which should be in quote marks since we are in the regular expression modus). Do not forget the semicolon and the newline at the end o your rule.

Now let us see how you are doing so far:

3.4 composing the network

Go back to the script and add
`read regex < rules.txt` to the already existing commands.

In order to compose in the right order, we need to
`turn stack` and then
`compose net`

Return to the interface and type in *source script.xfst* and *print lower-words*:
If your output has the vowel alternations in it — well done, go on! If not, then try to rewrite your rule.

Even you have accomplished the Umlaut, there are several problems left:

1. **Maeuese** is wrong — the *u* should not form an Umlaut
2. **Huende** is not the correct form — it should be **Hunde**
3. **+Pl** should not be in our output

Let us deal with these problems one by one:

3.4.1 *Maeuese

When we look at the data, the vowel that changes is always on second position. We therefore should extend our rule. If we describe the environment **before** our place holder _ closely, we can avoid the *u* in *Maeuse* changing. We therefore need to define the word boundary (represented by .#.) and allow only **one** letter between the boundary and the place holder: ? (without a star the question mark represents exactly one letter).
Extend your rule, save the file again and rerun your script. Check the output: if you have **Maeuse** then go on to the next problem, if not, then rewrite your rule.

3.4.2 *Huende

This problem is a little bit more complicated. How can we avoid the vowel within *Hunde* alternating? We need a special indicator on this word so that we can use it within the conditions. Another tag however will be very confusing ... you therefore should mark this word with a flag.
Return to your Lexicon, invent/name a flag of your choice @P.?.?@ and attach this flag **directly** to the stem form in your `LEXICON Root`. Don't forget to declare it within the Multichar_Symbols section.
Return to your rules. The easiest way is to write a new rule which is composed with the rule we have already written. It should translate into
"u e becomes u iff there is following "@P.?.?@" somewhere".
Write your rule **after** your first rule — this way, u e will already exist.
compose the two rules after the following scheme:

```
[Regular Expression/Rule 1]
.o.
```

```
[Regular Expression/Rule 2];
```

Save your files and rerun your script. If the output of the plural form of *Hund* is *Hunde* you accomplished your task and can go on.

3.4.3 +Pl

As a last step we want to get rid of the plural tag on the lower side. Therefore we need to compose our two rules with one more: “+Pl becomes zero”.
Rerun your script and check your output closely.... if all the plural forms are right:

Congratulations!!

4 Advanced task

If you would like to do more, continue by working on the other German plural forms.

5 Some help

Below is our testlexicon, which we used beforehand for the English data. Perhaps this will help you to write the lexicon for the German Plural:

```
Multichar_Symbols

+Sg +Pl +N +A
@P.LIQUID.yes@  @D.LIQUID@
@P.COUNT.yes@ @R.COUNT.yes@

LEXICON Root

milk@P.LIQUID.yes@          Noun;
car@P.COUNT.yes@            Noun;
pity                        Noun;

LEXICON Noun
+N:0                        SgPl;
@D.LIQUID@                  Adj;

LEXICON SgPl

< "+Pl":s "@R.COUNT.yes@" >    #;
+Sg:0    #;

LEXICON Adj
+A:less                         #;
```

useful commands:

- *read lexc* < *lex.txt* →To load the lexicon onto the stack
- *print lower-words* → If you would like to print the output
- *clear stack* → To empty the stack
- *eliminate flag NAME* → Do not forget to eliminate the flags!
- *read regex* < *rules.txt* → will load the rules on the stack
- *source script.txt* → will load the script onto the stack and with it all the commands you wrote into it.

Perhaps, the following list of regular expressions will help you:

0 = EPSILON	? = ANY SYMBOL	.#. = BOUNDARY SYMBOL
() = OPTIONALITY	+ = CONCATENATION WITH ITSELF ONE OR MORE TIMES	* = CONCATENATION WITH ITSELF ZERO OR MORE TIMES
~ = NEGATION	_ = PLACE HOLDER	{} AND = CONCATENATION
[] = GROUPING	→ = BECOMES...	|| = IN THE CONTEXT OF...
| = UNION	& = INTERSECTION	.x. AND : = CROSSPRODUCT
.o. = COMPOSITION		

Good Luck! :)
If you have any further questions, feel free to ask!