Modelling morphology with finite-state transducers using HFST

Why make a finite-state transducer?

Finite-state transducers are amazing, they allow for both analysis and generation in a single model, you can use them for testing your morphological rules and find errors in how you analyse morphology.

You might be really into neural networks because you can just take some manually annotated data and produce a wonderful model with a lot of weights that gets really high F-score. But where does that data come from and how can you evaluate your model? Well, one way to generate lots of data instead of manually annotating it — think of the poor annotators ;(— is to build a finite-state model.

A simple lexical transducer

The most basic lexical transducer can be made with a `Multichar_Symbols` definition and a `Root` lexicon. These are two parts that are necessary for the transducer to compile. In the Multichar_Symbols section we define our grammatical tags that we are going to use, and we can use the `Root` lexicon to store a few lexemes. For example,

```
Multichar_Symbols

%<n%>                ! Имя существительное, Noun

LEXICON Root

урам%<n%>:урам # ;   ! "улица", "street"
```

In `Multichar_Symbols` we define a grammatical tag for noun and leave a comment (the part after the `!` symbol) to indicate what the symbol stands for. Tags can take any form, but my preference is for grammatical tags to appear between less than < and greater than > symbols, which need to be escaped with the symbol `%`. The `Multichar_Symbols` section ends when the first `LEXICON` appears. Other things that go in the `Multichar_Symbols` section include archiphonemes and helper symbols for the phonological rules (often called "diacritic symbols" in the literature).

The remainder of the transducer is made up of a set of "continuation lexica", these have unique names and are prefixed with the line "`LEXICON`". These lexica are read from `Root` and may call each other, including recursively. There is a special lexicon which is pre-defined and called "`#`" which indicates the end of the string.

The remainder of the transducer is made up of a set of "continuation lexica", these have unique names and There are two sides which are separated by a colon, `:`. These two sides may be referred to in different ways in the literature. Get used to working out which side is being referred to. This is complicated by the fact that some ways of referring to them are ambiguous:

- left side, upper side, lexical side, lower side
- right side, lower side, morphotactic side, upper side, surface side

I will try and be consistent and use the following:

- lexical form/side, to refer to the lemma + tags
- morphotactic form/side, to refer to the stem + suffixes
- surface form/side, to refer to the final surface form

A pair of lexical/morphotactic strings should be followed by an obligatory continuation class, which may be `#` for end of string and then a semicolon, `;`. Comments may be included anywhere in the file by using a `!` symbol which applies to the end of the line.

So, now we've gone through that explanation, let's try compiling our lexicon file. You should open a new file with your favourite text editor and type in the code above. Save it as `chv.lexc` in a new directory and navigate there on the command line. Give the following command:

```
$ hfst-lexc chv.lexc -o chv.lexc.hfst
```

This command says to use the HFST lexc compiler to convert the lexicon file, `chv.lexc` into a binary representation and store the output in `chv.lexc.hfst`. The command should give the following output:

```
hfst-lexc: warning: Defaulting to OpenFst tropical type
Root...
```

and you should get a file called `chv.lexc.hfst`:

```
$ ls -l chv.lexc.hfst
-rw-r--r-- 1 fran fran 491 des  3 19:57 chv.lexc.hfst
```

As we didn't tell the compiler what kind of transducer we wanted to produce it has defaulted to OpenFst (a backend transducer library) and the tropical weight transducer (a way of distributing weights over an FST).

There are a few things we need to get used to doing with the lexical transducer. The first is printing it out, we can do this in a couple of ways. First we can print out the strings that the transducer covers, using the `hfst-fst2strings` command,

```
$ hfst-fst2strings chv.lexc.hfst
урам<n>:урам
```

Our fairly limited transducer only recognises one string pair so far. We can also print out the FST that was produced:

```
$ hfst-fst2txt chv.lexc.hfst
0       1       у       у       0.000000
1       2       р       р       0.000000
2       3       а       а       0.000000
3       4       м       м       0.000000
4       5       <n>     @0@     0.000000
5       0.000000
```

Here the first column is the input state, the second column is the output state, the third and fourth columns are the input and output symbols respectively and the fifth column is the weight of the transition. The `@0@` is the default symbol for epsilon — that is no input/output and final states have only two columns the first being the state id and the second being the weight.

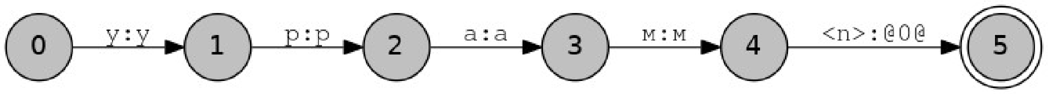
As you can imagine it is fairly easy to write a program to turn this tabular format into a format appropriate for visualising with GraphViz or some other graph visualisation library. Feel free to write your own visualisation method using your preferred library, or use this one I prepared earlier:

```
import sys

print('digraph G { rankdir="LR"')
print('node [fontname="Tahoma",shape=circle,fontsize=14,fixedsize=true,fillcolo
print('edge [fontname="FreeMono",fontsize=14]')
for line in sys.stdin.readlines():
        line = line.strip()
        row = line.split('\t')
        if len(row) >= 4:
                print('%s [label="%s"];' % (row[0], row[0]))
                print('%s -> %s [label="%s:%s"];' % (row[0], row[1], row[2], ro
        elif len(row) == 2: # Final state
                print('%s [label="%s",shape=doublecircle];' % (row[0], row[0]))

print('}')
```

You can save it in a file called `att2dot.py` and run it as follows:

```
$ hfst-fst2txt chv.lexc.hfst | python3 att2dot.py  | dot -Tpng -ochv.lexc.png
```

You should get an output file that looks something like:



Being able to visualise the transducer and see which strings it accepts is vital for being able to debug your code. Now, go back to your `chv.lexc` file and add some more stems, for example *пахча* "сад, garden", *хула* "город, city" and *канаш* "совет, council". Then recompile and rerun the other steps up to visualisation.

Morphotactics

The morphotactics of a language is the way that morphemes combine to make surface forms. If you are one of those people that believes in morphemes then you probably also believe that they can be combined and that there are language-specific constraints on their combination, for example in Russian if the plural locative morpheme is -ах then applying it to a stem should result in *городах* and not **ахгород*.

Continuation classes

In finite-state transducers the the morphotactic ordering constraints are implemented by means of continuation classes. These are sets of suffixes which can appear in the same position. For example let's suppose we want to add the plural suffix in Chuvash, which in the nominative is invariant.

```
Multichar_Symbols

%<n%>                ! Имя существительное, Noun
%<pl%>               ! Множественное число, Plural

%>                   ! Граница морфемы, Morpheme boundary

LEXICON Root

Nouns ;

LEXICON PLURAL

             # ;
%<pl%>:%>сем # ;

LEXICON N

%<n%>: PLURAL ;

LEXICON Nouns

урам:урам N ;     ! "улица", "street"
пахча:пахча N ;   ! "сад", "garden"
хула:хула N ;     ! "город", "city"
канаш:канаш N ;   ! "совет", "council"
```

This `lexc` file defines three new continuation classes:

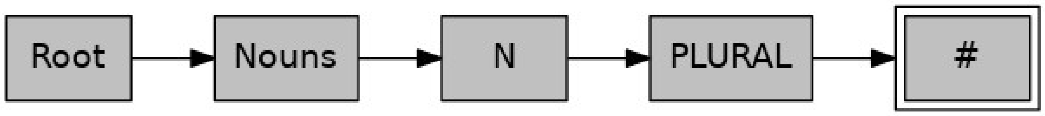
- `Nouns`: This is used for our list of stems, usually there is one continuation class per major lexical category (part of speech)
- `N`: This is our continuation lexicon for nouns, here we give a part of speech tag and pointers to the set of suffixes that can attach directly to the stem.
- `PLURAL`: Here we define the plural suffix and say that this is (for now) the end of the word. Note that we could also put a tag for singular here if were linguistically expedient.

The exact way you lay out the continuation classes will be different depending on the language you are working on. For fusional languages you might like to divide words according to stem class and then have a separate continuation class for each stem class.

It's also worth noting that we can plot the graph of continuation classes in a similar way to the letter transducer we plotted before. There is a script (https://ftyers.github.io/2017-%D0%9A%D0%9B_%D0%9C%D0%9A%D0%9B/scripts/lexc2dot.py) that will produce a GraphViz file from a `.lexc` file. For example if you run:

```
cat chv.lexc | python3 lexc2dot.py | dot -Tpng -ochv.lexc.png
```

Then you should get the following result:



We can also, as before compile and list the accepted strings. Let's do that to make sure that everything идёт по плану.

```
$ hfst-lexc chv.lexc -o chv.lexc.hfst
hfst-lexc: warning: Defaulting to OpenFst tropical type
Root...1 PLURAL...2 N...1 Nouns...

$ hfst-fst2strings chv.lexc.hfst
урам<n>:урам
урам<n><pl>:урам>сем
пахча<n>:пахча
пахча<n><pl>:пахча>сем
канаш<n>:канаш
канаш<n><pl>:канаш>сем
хула<n>:хула
хула<n><pl>:хула>сем
```

And now run it through `hfst-fst2txt` to visualise the resulting transducer.

Archiphonemes

Now let's try and add a case. We can start with one of the easier ones, the instrumental, which is `-пА`, that is *-па* with back vowels and *-пе* with front vowels. At this point we have two choices, we can either make two continuation classes for the cases, one for back vowel contexts and one for front vowel contexts, for example:

```
LEXICON CASES-BACK

%<ins%>:%>па # ;

LEXICON CASES-FRONT

%<ins%>:%>пе # ;
```

The advantage with this is it makes for easier debugging in some respects because all information is in one place. The disadvantage is that it means you have to duplicate all continuation classes into those which have front and those which have back vowels. If you imagine you have to split for every phonological process (elision, vowel harmony, lenition, voicing, etc.) then you can see that it could produce a very large number of continuation classes. For example, in one implementation of Finnish splitting the classes by phonological process resulted in 516 noun classes, where an unsplit implementation had five.

So instead, what we do is provide a placeholder (archiphoneme) instead. I usually write these placeholders inbetween , `{...}`. For example we could write:

```
LEXICON CASES

%<ins%>:%>п%{A%} # ;

LEXICON PLURAL

             CASES ;
%<pl%>:%>сем CASES ;

LEXICON N

%<n%>: PLURAL ;
```

⚠ **WARNING!** don't forget to define `%{A%}` as a multicharacter symbol.

So, if we save this into our file and recompile, we should get the following output:

```
$ hfst-lexc chv.lexc | hfst-fst2strings
hfst-lexc: warning: Defaulting to OpenFst tropical type
Root...1 CASES...1 PLURAL...2 N...1 Nouns...
пахча<n><ins>:пахча>п{A}
пахча<n><pl><ins>:пахча>сем>п{A}
урам<n><ins>:урам>п{A}
урам<n><pl><ins>:урам>сем>п{A}
канаш<n><ins>:канаш>п{A}
канаш<n><pl><ins>:канаш>сем>п{A}
хула<n><ins>:хула>п{A}
хула<n><pl><ins>:хула>сем>п{A}
```

There are two things remaining to make these morphotactic forms (on the right) into legit surface forms: 1) We need to make sure to output `{A}` as -a- or -e- depending on context, and (2) we need to remove the morpheme boundary, `>`. Both of these can be taken care of using phonological rules using the `twol` formalism.

Phonological rules

Before we get started with writing rules in `twol` it is worth giving some background information. First of all `twol` rules are not Chomsky-Halle style ordered rewrite rules. There is no ordering in the rules, they may look similar, but the way they are applied is very different. Consider the following two rulesets, (a) presents ordered rewrite rules, while (b) presents two-level constraint style rules.

(a)

```
a → b / c
b → d / c
```

acaca → acbcb → acdcd

(b)

```
a:b <=> c
b:d <=> c
```

acaca → acbcb

They both look superficially very similar, but the result is quite different, this is because `twol` rules are not applied in order, the output of one rule is not piped into another rule. All rules are applied at the same time. If it's not clear now, don't worry, it's just something to bear in mind.

Let's look at a real example, make a new text file called `chv.twol` and type in the Chuvash alphabet, including our `{A}` archiphoneme as follows:

```
Alphabet
  а ӑ е ё ӗ и о у ӳ ы э ю я б в г д ж з к л м н п р с ç т ф х ц ч ш щ й ь ъ
  А Ӑ Е Ё Ӗ И О У Ӳ Ы Э Ю Я Б В Г Д Ж З К Л М Н П Р С Ç Т Ф Х Ц Ч Ш Щ Й Ь Ъ
 %{A%}:а %{A%}:е
;
```

The alphabet determines the set of possible output strings. The morphotactic side of the transducer (e.g. the strings that look like `урам>сем>п{A}`) is multiplied by this alphabet into the set of forms on which the constraint rules apply. For example, given the string `урам>сем>п{A}` after running through the alphabet expansion, the result would be the following string pairs:

```
урам>сем>п{A}:урам>сем>па
урам>сем>п{A}:урам>сем>пе
```

We can now write our first rule, a simple one to remove the morpheme boundary:

```
Rules

"Remove morpheme boundary"
%>:0 <=> _ ;
```

A rule is composed of a one-line description in between double quotes, `"..."` followed by a constraint in the form:

```
! a:b   CONSTRAINT_OPERATOR   LEFT_CONTEXT   _   RIGHT_CONTEXT ;
```

Where `a` is an alphabetic symbol on the morphotactic side, `b` is an alphabetic symbol on the surface side and `a:b` is the *centre* of the rule; `CONSTRAINT_OPERATOR` is an arrow defining the constraint type (more on that later); `LEFT_CONTEXT` is the context to the left and `RIGHT_CONTEXT` is the context to the right. Both left and right contexts are represented by regular expressions over symbol pairs.

So this rule means "constrain the surface representation of `>` to be `0`, that is empty in all contexts". We can now try compiling the rule and our alphabet to see the results:

```
$ hfst-twolc chv.twol -o chv.twol.hfst
Reading input from chv.twol.
Writing output to chv.twol.hfst.
Reading alphabet.
Reading rules and compiling their contexts and centers.
Compiling rules.
Storing rules.
```

In order to apply our ruleset to our compiled lexicon we use the `hfst-compose-intersect` program. This takes as input two arguments, our compiled lexicon, `chv.lexc.hfst` and our compiled twol file, `chv.twol.hfst`. It might be convenient at this point that we set up a `Makefile` to make the compilation turnaround faster, so open a new file called `Makefile`, and write in:

```
all:
        hfst-lexc chv.lexc -o chv.lexc.hfst
        hfst-twolc chv.twol -o chv.twol.hfst
        hfst-compose-intersect -1 chv.lexc.hfst -2 chv.twol.hfst -o chv.gen.hfs
```

Then go to the command line and type `make`,

```
$ make
hfst-lexc chv.lexc -o chv.lexc.hfst
hfst-lexc: warning: Defaulting to OpenFst tropical type
Root...1 CASES...1 PLURAL...2 N...1 Nouns...
hfst-twolc chv.twol -o chv.twol.hfst
Reading input from chv.twol.
Writing output to chv.twol.hfst.
Reading alphabet.
Reading rules and compiling their contexts and centers.
Compiling rules.
Storing rules.
hfst-compose-intersect -1 chv.lexc.hfst -2 chv.twol.hfst -o chv.gen.hfst
```

This compiles the lexical transducer, then the two-level rules and finally composes the lexicon with the rules. Composition basically means that we take the output of the first transducer and we give it as input to the second transducer, then we throw away the intermediate part. So,

`a:b` ∘ `b:c` → `a:c`.

We can see the output of the process by using `hfst-fst2strings` as before,

```
$ hfst-fst2strings chv.gen.hfst
канаш<n><ins>:канашпа
канаш<n><ins>:канашпе
канаш<n><pl><ins>:канашсемпа
канаш<n><pl><ins>:канашсемпе
пахча<n><ins>:пахчапа
пахча<n><ins>:пахчапе
пахча<n><pl><ins>:пахчасемпа
пахча<n><pl><ins>:пахчасемпе
урам<n><ins>:урампа
урам<n><ins>:урампе
урам<n><pl><ins>:урамсемпа
урам<n><pl><ins>:урамсемпе
хула<n><ins>:хулапа
хула<n><ins>:хулапе
хула<n><pl><ins>:хуласемпа
хула<n><pl><ins>:хуласемпе
```

As you can see, now we have all of the possible forms, this is the key to `twol`, we first expand all the possibilities and then constrain them. So, if we want to write a constraint for vowel harmony, what might it look like ? First we have to define what vowel harmony means.

- The archiphoneme `%{A%}` should be -a- after a back vowel and any number of consonants, and it should be -e- after a front vowel and any number of consonants.

The first thing we should do is define some sets for what back vowel, front vowel and consonant mean:

```
Sets

BackVow = ă а ы о у я ё ю ;

FrontVow = ĕ э и ӳ е ;

Cns = б в г д ж з к л м н п р с ç т ф х ц ч ш щ й ь ъ ;
```

This code should go between the end of the `Alphabet` and the beginning of `Rules`. Once we have done that we can go on to define our first phonological rule:

```
"Back vowel harmony for archiphoneme {A}"
%{A%}:a <=> BackVow: [ Cns: | %>: ]+ _ ;
```

This rule says that the symbol pair `%{A%}:a` should only be considered valid if there is a previous back vowel followed by one or more consonants. Go and save this and compile it and look at the output.

```
$ hfst-fst2strings chv.gen.hfst
канаш<n><ins>:канашпа
канаш<n><pl><ins>:канашсемпе
пахча<n><ins>:пахчапа
пахча<n><pl><ins>:пахчасемпе
урам<n><ins>:урампа
урам<n><pl><ins>:урамсемпе
хула<n><ins>:хулапа
хула<n><pl><ins>:хуласемпе
```

Note that we haven't said anything about `%{A%}:e`, so why are the front vowels after back vowels removed as well as the back vowels after front vowels ? The answer lies in the type of operator. What the <=> operator says is that:

1. If the symbol pair `%{A%}:a` appears it must be in the context `BackVow: [ Cns: | %>: ]+ _`
2. If the lexical/morphotactic `%{A%}:` appears in the context `BackVow: [ Cns: | %>: ]+ _` then it must correspond on the surface to "`a`"

So, (1) constrains the correspondence of `%{A%}:a` to only be in the context we have specified and (2) constrains the correspondence of `%{A%}:` to only be "`a`" in the context we have specified. There are three other operators (or arrows):

Rule type	**Interpretation**
`a:b => _ ;`	If the symbol pair `a:b` appears it must be in context _
`a:b <= _ ;`	If lexical `a` appears in the context _ then it must correspond to surface `b`
`a:b /<= _ ;`	Lexical `a` never corresponds to `b` in context _

Now try out the other arrows with your rule, recompile and look at the output.

Rule interactions

You might be wondering at this point how we can do complex transformations if we can only work with changing a single symbol at once and have no concept of rule ordering. Let's take a look at the Chuvash genitive to get an idea of how rules can interact.

Context	Form
`-а, -е`	-ăн, -ĕн -н
`-и`	-йĕн -н
`-C`	-ăн, -ĕн
`-Că, -Cĕ`	-CCăн, -CCĕн
`-Cу, -Cӳ`	-Căвăн, -Cĕвĕн
`-сен`	-сем, -сенĕн

The actual story is a bit more complicated, but this is enough to get our teeth into for now. So, let's remember that our current `.lexc` file looks like:

```
Multichar_Symbols

%<n%>                ! Имя существительное, Noun
%<pl%>               ! Множественное число, Plural
%<nom%>              ! Именительный падеж, Nominative
%<ins%>              ! Творительный падеж, Instrumental

%{A%}                ! Архифонема [а] или [е], Archiphoneme [a] or [e]

%>                   ! Граница морфемы, Morpheme boundary

LEXICON Root

Nouns ;

LEXICON CASES

%<ins%>:%>п%{A%} # ;

LEXICON PLURAL

             CASES ;
%<pl%>:%>сем CASES ;

LEXICON N

%<n%>: PLURAL ;

LEXICON Nouns

урам:урам N ;     ! "улица", "street"
пахча:пахча N ;   ! "сад", "garden"
хула:хула N ;     ! "город", "city"
канаш:канаш N ;   ! "совет", "council"
```

Let's assume for simplicity that we are dealing with just the stems that we have in the file, we need to generate the following forms:

Stem	Singular	Plural
урам	`урамăн`	урамсен
канаш	`канашăн`	канашсен
пахча	`пахчан`	пахчасен
хула	`хулан`	хуласен

So what are the possible options? Well, first of all, the *-м* at the end of the plural morpheme looks special, because it changes to *-н* in the plural genitive. So we should change our plural morpheme to `%>се%{м%}`. The next question is what do we do with that?

1. The genitive morpheme will be `%>%{Ă%}н` after both singular and plural
   - `%{Ă%}:0` if there is a surface vowel before the morpheme boundary

  - `%{Ă%}:0` if there is previous `%{м%}:0`
  - `%{м%}:0` if there is a following `%{Ă%}:` followed by `н`
  - `%{Ă%}:ă` or `%{Ă%}:ĕ` according to vowel harmony if previous consonant
2. The genitive morpheme will be `%>%{н%}` in plural and `%>%{Ă%}н` in singular
  - `%{н%}:н` if it is the end of the string
  - `%{м%}:0` if there is a following `%{н%}:н`
  - `%{Ă%}:0` if there is a surface vowel before the morpheme boundary
  - `%{Ă%}:ă` or `%{Ă%}:ĕ` according to vowel harmony if previous consonant

Does it matter which variant we choose? Well, that depends on the task(s) we're planning to use the transducer for. If we are just interested in morphological analysis then it is really a matter of personal taste or belief (what is more convenient computationally?). However, if you are also planning to use the transducer for morphological segmentation, then you should perhaps think about what kind of segments you want and what is going on linguistically.

Now implement option (1) above in your `.lexc` file, you should end up with the following output:

```
$ hfst-fst2strings chv.lexc.hfst | grep урам | grep gen
урам<n><gen>:урам>{Ă}н
урам<n><pl><gen>:урам>се{м}>{Ă}н
```

If you are having problems doing it, then ask for help. Remember to declare your archiphonemes! When you have updated your `.lexc` file, add the following lines to the alphabet of your `.twol` file:

```
 %{Ă%}:ă %{Ă%}:ĕ %{Ă%}:0
 %{м%}:м %{м%}:0
```

Then compile and look at the output,

```
$ hfst-fst2strings chv.gen.hfst | grep урам | grep gen
урам<n><gen>:урамн
урам<n><gen>:урамăн
урам<n><gen>:урамĕн
урам<n><pl><gen>:урамсен
урам<n><pl><gen>:урамсеăн
урам<n><pl><gen>:урамсеĕн
урам<n><pl><gen>:урамсемн
урам<n><pl><gen>:урамсемăн
урам<n><pl><gen>:урамсемĕн
```

As you can see we have a fair number of invalid forms generated by our alphabet expansion this time.

So the first thing we can take care of would be vowel harmony. We can quite easily get rid of the surface forms with invalid vowel harmony by taking our previous rule and adapting it for `%{Ă%}`,

```
"Back vowel harmony for archiphoneme {Ă}"
%{Ă%}:ă <=> BackVow: [ ArchiCns: | Cns: | %>: ]+ _ ;
```

I've added one new set here, `ArchiCns = %{м%} ;` to represent the archiphoneme for the consonant that can be `0` on the surface. Save the rule and recompile and test,

```
$ hfst-fst2strings chv.gen.hfst | grep урам | grep gen
урам<n><gen>:урамăн
урам<n><pl><gen>:урамсен
урам<n><pl><gen>:урамсеĕн
урам<n><pl><gen>:урамсемн
урам<n><pl><gen>:урамсемĕн
```

That's better, what should we try and tackle next ? How about constraining `%{Ă%}` to be `0` when following `%{м%}` ?

```
"Non surface {Ă} in plural genitive"
%{Ă%}:0 <=> %{м%}: %>: _ н ;
```

⚠ **WARNING!** Remember to put the **:** after the `%{м%}`, an easy mistake to make is to just put `%{м%}` but remember that without the **:** that would mean the symbol pair `%{м%}:%{м%}` and archiphonemes should never be found on the surface side.

And let's compile:

```
$ make
hfst-lexc chv.lexc -o chv.lexc.hfst
hfst-lexc: warning: Defaulting to OpenFst tropical type
Root...1 CASES...3 PLURAL...2 N...1 Nouns...
hfst-twolc chv.twol -o chv.twol.hfst
Reading input from chv.twol.
Writing output to chv.twol.hfst.
Reading alphabet.
Reading sets.
Reading rules and compiling their contexts and centers.
There is a <=-rule conflict between "Back vowel harmony for archiphoneme {Ă}" a
E.g. in context __HFST_TWOLC_.#.:__HFST_TWOLC_.#. ё:ё ç:ç {м}:м >: _ н:н __HFST

Compiling rules.
Storing rules.
hfst-compose-intersect -1 chv.lexc.hfst -2 chv.twol.hfst -o chv.gen.hfst
```

Argghhh! What happened ? Well, the problem is that we have two rules that say conflicting things:

(a)

```
"Back vowel harmony for archiphoneme {Ă}"
%{Ă%}:ă <=> BackVow: [ ArchiCns: | Cns: | %>: ]+
```

(b)

```
"Non surface {Ă} in plural genitive"
%{Ă%}:0 <=> %{м%}: %>: _ н ;
```

So let's take a look at the string pairs:

```
(a)    у р а м > с е {м} > {Ă} н
       у р а м 0 с е  0  0  ӗ  н
                   ^--------^

(b)
       у р а м > с е {м} > {Ă} н
       у р а м 0 с е  0  0  0  н
                   ^--------^
```

So the easiest way to fix this is with an `except` clause to the rule. These are written as follows:

```
"Back vowel harmony for archiphoneme {Ă}"
%{Ă%}:ă <=> BackVow: [ ArchiCns: | Cns: | %>: ]+ _ ;
        except
                             %{м%}: %>:  _ н ;
```

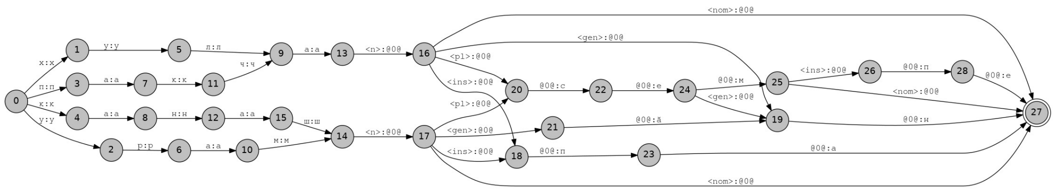
We basically limit the rule to work in all contexts apart from those in the `except` clause.

I leave the rule for `%{м%}` as an exercise for the reader. In the end the result should be:

```
$ hfst-fst2strings chv.gen.hfst |grep gen
канаш<n><gen>:канашăн
канаш<n><pl><gen>:канашсен
пахча<n><gen>:пахчан
пахча<n><pl><gen>:пахчасен
урам<n><gen>:урамăн
урам<n><pl><gen>:урамсен
хула<n><gen>:хулан
хула<n><pl><gen>:хуласен
```

You will also likely need to add a rule or expand a context to cause `%{Ă%}` to surface as `0` after stem vowels, otherwise you will end up with output like **пахчаăн* and **хулаăн*.

The final transducer should look something like:



If it looks a bit different try running:

```
$ hfst-minimise chv.gen.hfst  | hfst-fst2txt
```

What does minimisation do ?

Some more rule syntax

Multiple environments

You can use list multiple environments by just putting them one after another,

```
"x is constrained to be y before a and after b"
x:y <=>    _ :a ;
        :b _ ;
```

The rule above maps `x` to `y` before a surface `a` and after a surface `b`.

Matched correspondences

You can do pairs of symbols in the same environment using a single rule, like this:

```
"{A} must correspond to the vowel before it"
%{A%}:Vy <=> :Vx _ ;
      where Vx in ( a e o )
            Vy in ( a e o )
      matched ;
```

This rule turns `%{A%}` into either `a`, `e`, or `o` to match the character before it.

A similar rule:

```
"Vowels raise after a null-realised {x}"
Vx:Vy <=> %{x%}:0 _ ;
      where Vx in ( a e o )
            Vy in ( ə i u )
      matched ;
```

More on morphotactics

Morphotactic constraints

What we have looked at so far has been limited to suffixing morphology, but languages exhibit inflection in different ways. Either with suffixes, with prefixes, both, or circumfixes. However, when coming to implementing this kind of stuff in a finite-state transducer we are often presented with the problem that we would like the inflection on the left, but the morphological tag on the right. Let's take Avar for example, some classes of verbs inflect for agreement using a prefix and for tense etc. using suffixes. Check out this example of the aorist of the verb *бицине* "говорить", "to say":

Form	**Morphemes**	**Analysis**
бицуна	б-иц-уна	`бицине`, Aorist, Neuter
йицуна	й-иц-уна	`бицине`, Aorist, Feminine
вицуна	в-иц-уна	`бицине`, Aorist, Masculine
рицуна	р-иц-уна	`бицине`, Aorist, Plural

To represent this in the lexicon we have to decide a number of things,

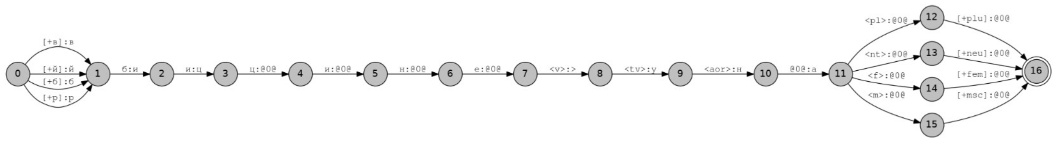
- Where do we want the tag representing the agreement morpheme ?
  - Before the stem or after the stem ?
- If we want the tag before the stem should we use constraints or flag diacritics ?
  - Constraints work like constraints in `twol`, they disallow certain paths at compile time.

- Flag diacritics maintain all the forms in the final transducer but have symbols which are evaluated at runtime to discard paths.

The first approach we are going to take is to mark strings with special symbols and then use `twol`-style constraints to disallow the strings that have conflicting symbols. Make a new file called `ava.lexc` and add the following code:

```
Multichar_Symbols

%<v%>                ! Глагол, Verb
%<tv%>               ! Переходный, Transitive
%<aor%>              ! Аорист, Aorist
%<m%>                ! Мужский род, Masculine
%<f%>                ! Женский род, Feminine
%<nt%>               ! Средный род, Neuter
%<pl%>               ! Множественное число, Plural

%[+в%]               ! Префикс в-, Prefix v-
%[+й%]               ! Префикс й-, Prefix j-
%[+б%]               ! Префикс б-, Prefix b-
%[+р%]               ! Префикс р-, Prefix r-

%[+msc%]             ! Согласование мужского рода, Masculine agreement
%[+fem%]             ! Согласование женского рода, Feminine agreement
%[+neu%]             ! Согласование средного рода, Neuter agreement
%[+plu%]             ! Согласование множественного падежа, Plural agreement

%>                   ! Граница морфемы, Morpheme boundary

LEXICON Root

Prefixes ;

LEXICON Prefixes

%[+в%]:в%< Verbs ;
%[+й%]:й Verbs ;
%[+б%]:б Verbs ;
%[+р%]:р Verbs ;

LEXICON AGR

%<m%>%[%+msc%]: # ;
%<f%>%[%+fem%]: # ;
%<nt%>%[%+neu%]: # ;
%<pl%>%[%+plu%]: # ;

LEXICON V-TV

%<v%>%<tv%>%<aor%>:%>уна AGR ;

LEXICON Verbs

бицине:иц V-TV ; ! "говорить", "say"
```

The tags within square brackets, `[` and `]` are constraints that we can use in writing `twol` rules, for example we might want to say that `[+в]` symbol must have `[+msc]` symbol somewhere later in the string. Note that we need to define all of these constraint symbols as multicharacter symbols in the header of the file. When we compile the transducer and view it, it should look like this:



And we should get the following output from `hfst-fst2strings`,

```
[+в]бицине<v><tv><aor><pl>[+plu]:виц>уна
[+в]бицине<v><tv><aor><nt>[+neu]:виц>уна
[+в]бицине<v><tv><aor><f>[+fem]:виц>уна
[+в]бицине<v><tv><aor><m>[+msc]:виц>уна
[+й]бицине<v><tv><aor><pl>[+plu]:йиц>уна
[+й]бицине<v><tv><aor><nt>[+neu]:йиц>уна
[+й]бицине<v><tv><aor><f>[+fem]:йиц>уна
[+й]бицине<v><tv><aor><m>[+msc]:йиц>уна
[+б]бицине<v><tv><aor><pl>[+plu]:биц>уна
[+б]бицине<v><tv><aor><nt>[+neu]:биц>уна
[+б]бицине<v><tv><aor><f>[+fem]:биц>уна
[+б]бицине<v><tv><aor><m>[+msc]:биц>уна
[+р]бицине<v><tv><aor><pl>[+plu]:риц>уна
[+р]бицине<v><tv><aor><nt>[+neu]:риц>уна
[+р]бицине<v><tv><aor><f>[+fem]:риц>уна
[+р]бицине<v><tv><aor><m>[+msc]:риц>уна
```

We can then go ahead and write a constraint rule in a new file, `ava.twoc`:

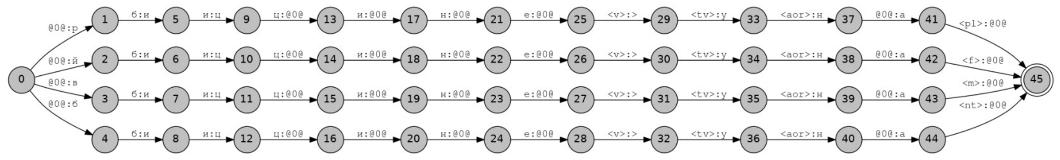
```
Alphabet

%[%+б%]:0 %[%+в%]:0 %[%+й%]:0 %[%+р%]:0 %[%+neu%]:0 %[%+msc%]:0 %[%+fem%]:0 %[%

Rules

"Match agreement prefixes with agreement tags"
Sx:0 /<= _ ;
   except
       _ ( : )* Sy:0 ;
   where Sx in ( %[%+б%]   %[%+в%]   %[%+й%]   %[%+р%]  )
         Sy in ( %[%+neu%] %[%+msc%] %[%+fem%] %[%+plu%] )
   matched ;
```

We can use the following commands to compile this rule and apply it to our lexical transducer:

```
hfst-lexc ava.lexc -o ava.lexc.hfst
hfst-twolc ava.twoc -o ava.twoc.hfst
hfst-invert ava.lexc.hfst | hfst-compose-intersect -1 - -2 ava.twoc.hfst | hfst
```

The first two commands are fairly straightforward, the third command is slightly different. This time we are applying constraints on the *morphotactic* side, not on the surface side, so we need to first `invert` the transducer before we apply the constraints. The `-1 -` means take the first transducer from standard input.

A downside of this approach is that it involves spelling out all intermediate strings, as you will see in the following graphic,



This means that the compiled transducer may get large and difficult to manage. However, depending on the exact linguistic data, it may not matter that much.

Flag diacritics

An alternative approach is to use *flag diacritics*, these are "invisible" symbols which allow the expression of constraints between non-consecutive parts of words. They have the following format:

```
    @FLAGTYPE.FEATURE.VALUE@
```

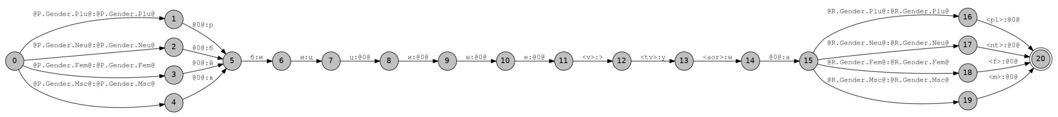
The flag values and features are arbitrary strings, and up to the user to decide. During application of a transducer, the runtime will decide whether or not a word is to be accepted depending on which flags co-occur in the same word. Note that when we define flag diacritics they **must** appear on both the lexical side and on the morphotactic side of the transducer.

```
Multichar_Symbols

%<v%>                ! Глагол, Verb
%<tv%>               ! Переходный, Transitive
%<aor%>              ! Аорист, Aorist
%<m%>                ! Мужский род, Masculine
%<f%>                ! Женский род, Feminine
%<nt%>               ! Средный род, Neuter
%<pl%>               ! Множественное число, Plural

@P.Gender.Msc@       ! Согласование мужского рода, Masculine agreement
@P.Gender.Fem@       ! Согласование женского рода, Feminine agreement
@P.Gender.Neu@       ! Согласование средного рода, Neuter agreement
@P.Gender.Plu@       ! Согласование множественного падежа, Plural agreement

@R.Gender.Msc@       ! Согласование мужского рода
@R.Gender.Fem@       ! Согласование женского рода
@R.Gender.Neu@       ! Согласование средного рода
@R.Gender.Plu@       ! Согласование множественного падежа

LEXICON Root

Prefixes ;

LEXICON Prefixes

@P.Gender.Msc@:@P.Gender.Msc@в Verbs ;
@P.Gender.Fem@:@P.Gender.Fem@й Verbs ;
@P.Gender.Neu@:@P.Gender.Neu@б Verbs ;
@P.Gender.Plu@:@P.Gender.Plu@р Verbs ;

LEXICON AGR

@R.Gender.Msc@%<m%>:@R.Gender.Msc@ # ;
@R.Gender.Fem@%<f%>:@R.Gender.Fem@ # ;
@R.Gender.Neu@%<nt%>:@R.Gender.Neu@ # ;
@R.Gender.Plu@%<pl%>:@R.Gender.Plu@ # ;

LEXICON V-TV

%<v%>%<tv%>%<aor%>:%>уна AGR ;

LEXICON Verbs

бицине:иц V-TV ; ! "говорить", "say"
```

Now try compiling and running the following commands,

```
$ hfst-fst2strings ava.lexc.hfst
$ hfst-fst2strings -X obey-flags ava.lexc.hfst
```

What difference do you note ?

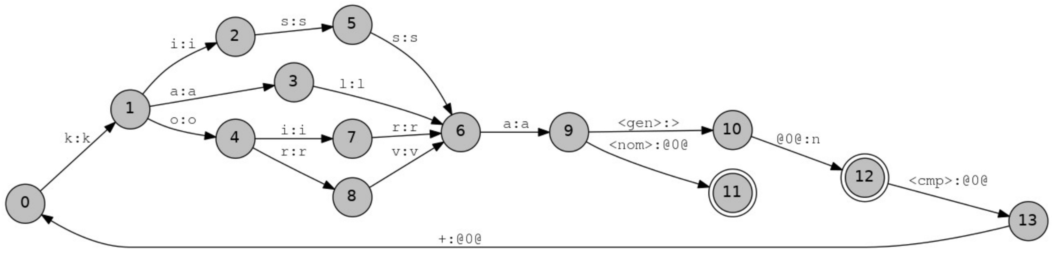

@P.Gender.Plu@:@P.Gender.Plu@
@P.Gender.Neu@:@P.Gender.Neu@
@P.Gender.Fem@:@P.Gender.Fem@
@P.Gender.Msc@:@P.Gender.Msc@
@0@:р
@0@:б
@0@:й
@0@:в
б:и
и:ц
ц:@0@
и:@0@
н:@0@
е:@0@
<v>:>
<tv>:у
<aor>:н
@0@:а
@R.Gender.Plu@:@R.Gender.Plu@
@R.Gender.Neu@:@R.Gender.Neu@
@R.Gender.Fem@:@R.Gender.Fem@
@R.Gender.Msc@:@R.Gender.Msc@
<pl>:@0@
<nt>:@0@
<f>:@0@
<m>:@0@

Compounding

So far all of our transducers have been acyclic — containing no cycles — but many languages have productive compounding which entains cyclic relations between items in the lexicon. Many languages in Russia write compound words separately, but Finnish doesn't. You can make a new lexicon called `fin.lexc`,

```
Multichar_Symbols

%<n%>                   ! Nimisana
%<cmp%>                 ! Yhdyssana
%<nom%>                 ! Nominatiivi
%<gen%>                 ! Genetiivi

%>                      ! Morfeemiraja
%+                      ! Yhdyssanaraja (leksikaalinen puoli)
%#                      ! Yhdyssanaraja

LEXICON Root

Nouns  ;

LEXICON Cmp

           # ;
%<cmp%>%+:%# Nouns ;

LEXICON 9  ! kala

%<nom%>:    # ;
%<gen%>:%>n Cmp ;

LEXICON 10 ! koira

%<nom%>:    # ;
%<gen%>:%>n Cmp ;

LEXICON Nouns

kissa%<n%>:kissa 9 ;  ! "кошка", "cat"
kala%<n%>:kala 9 ;    ! "рыба", "fish"
korva%<n%>:korva 10 ; ! "ухо", "ear"
koira%<n%>:koira 10 ; ! "собака", "dog"
```

If you compile this and try running it through `att2dot.py` then you should get the following transducer:



But note that if you try and print out this transducer you will get an error.

```
$ hfst-fst2strings fin.lexc.hfst
hfst-fst2strings: Transducer is cyclic. Use one or more of these options: -n, -
```

The reason for this error is that printing out all of the strings in a cyclic transducer would take forever, in terms of never terminate, so it asks you to specify what you actually want to do. The main options are:

- `-n` *n* give the first *n* strings
- `-N` *n* give the first *n* best strings (by weight)
- `-r` *n* give *n* random strings
- `-c` *n* follow at most *n* cycles

The options `-l` and `-L` limit the length of the input and output string respectively.

```
$ hfst-fst2strings -n 5 fin.lexc.hfst
kissa<n><gen>:kissa>n
kissa<n><gen><cmp>+kala<n><nom>:kissa>n#kala
kissa<n><gen><cmp>+kala<n><gen>:kissa>n#kala>n
kissa<n><gen><cmp>+kala<n><gen><cmp>+koira<n><nom>:kissa>n#kala>n#koira
kissa<n><gen><cmp>+kala<n><gen><cmp>+koira<n><gen>:kissa>n#kala>n#koira>n

$ hfst-fst2strings -r 5 fin.lexc.hfst
kala<n><gen>:kala>n
kala<n><nom>:kala
kissa<n><gen>:kissa>n
koira<n><nom>:koira
korva<n><gen>:korva>n
```

So we can easily analyse productive compounds like *kissankala* "cat fish" and *kissankalankoira* "cat fish dog". Note that this is not a full model of Finnish compounding, there is a difference between compounding from genitive and compounding from nominative and any model of compounding, especially such a simplistic model is likely to severely overgenerate/overanalyse.

Productive derivation

Next up productive derivation, for this we can go back to Chuvash. In Chuvash there is a derivational suffix *-лĂх* which attaches to nouns to make new nouns or adjectives with abstract meaning, for example:

- *тĕс* "вид", "aspect"; *тĕслĕх* "пример", "example"
- *патша* "царь", "tsar"; *патшалăх* "государство", "state"
- *куç* "глаз", "eye"; *куç* "очки", "glasses"

The typical way of dealing with this is to just add it with another continuation class, for example, try adding this snippet to your `chv.lexc` file:

```
LEXICON SUBST

PLURAL ;

LEXICON DER-N

%<der_лăх%>:%>л%{Ă%}х SUBST ;

LEXICON N

%<n%>: SUBST ;
%<n%>: DER-N ;

LEXICON Nouns

урам:урам N ;     ! "улица", "street"
пахча:пахча N ;   ! "сад", "garden"
хула:хула N ;     ! "город", "city"
канаш:канаш N ;   ! "совет", "council"
тĕс:тĕс N ;       ! "вид", "aspect"
патша:патша N ;   ! "царь", "tsar"
куҫ:куҫ N ;       ! "глаз", "eye"
```

Note: Remember to define the new multicharacter symbols! Also remember to include the other lexicons (e.g. `PLURAL`, `CASE` etc.)

However, what do we do in the case that we already have the derived word in the lexicon, for example for some applications we may want the derived analysis, but for others we may want the lexicalised analysis. Add *патшалăх* "государство", "state" to the noun lexicon and see what happens:

```
$ echo патшалăх | hfst-lookup -qp chv.mor.hfst
патшалăх        патшалăх<n><nom>        0,000000
патшалăх        патша<n><der_лăх><nom>  0,000000
```

To get around this problem we can start working with weights. In a weighted finite-state transducer each arc and consequently each path is assigned a weight. The weight of a path is usually some combination of the weights of the arcs (addition or multiplication). In `lexc` we can define weights for specific arcs in a special section after the continuation class, for example:

```
LEXICON DER-N

%<der_лăх%>:%>л%{Ă%}х SUBST "weight: 1.0" ;
```

This sets the weight of the *-лĂх* morpheme to be 1.0. Normally if we are working with probabilities (for example from a corpus) we work with negative log probabilities (so that a lower value ... e.g. `5/100 = 0.05` and `-log(0.05) = 2.99` is "heigher weight" than a higher value, e.g. `50/100 = 0.5`, `-log(0.5) = 0.69`). But if we are working with arbitrary manually specified weights we work with positive numbers and try and ensure that the less desirable analyses (typically derived and compound analyses if we want to favour lexicalisation) are higher weight, thus "worse".

```
$ echo патшалăх | hfst-lookup -qp chv.mor.hfst
патшалăх        патшалăх<n><nom>        0,000000
патшалăх        патша<n><der_лăх><nom>  1,000000

$ echo патшалăх | hfst-lookup -qp -b 0 chv.mor.hfst
патшалăх        патшалăх<n><nom>        0,000000

$ echo "тĕслĕх" | hfst-lookup -qp -b 0 chv.mor.hfst
тĕслĕх  тĕс<n><der_лăх><nom>    1,000000
```

The `-b 0` option means for the analyser to return only those analyses which are the same weight as the best scoring analysis. For the programmatically inclined, the `-b` stands for *beam search*. Later on in the tutorial we will learn how to learn analysis weights automatically from corpora.

Note that if the lower weight analysis is *not* available then the higher weight analysis will be returned in any case.

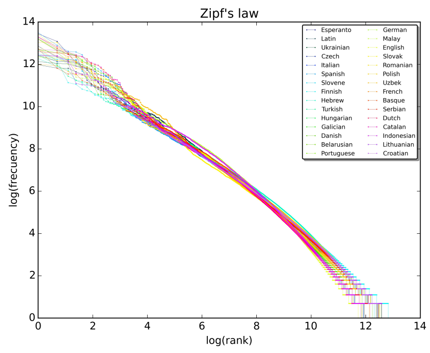
Lexicon construction

So, you've got your morphotactics more or less sorted out, and you've made a good start on your morphophonological rules. Where are you going to get the lexemes ? Depending on the language you choose you may already have a beautifully prepared list of stems with declension categories, for example for Russian there is Zaliznyak's system and for Finnish there is the Kotus categorisation. For other languages there might be a dictionary with less elaborate systems of declension classes, perhaps traditional ones that underspecify the morphology, or perhaps ones that don't specify the morphology at all. In the worst case you may have to just rely on a Swadesh list or a simple list of inflected forms from a corpus.

In any case, whether you have a complete system or just a list of inflected forms, the best idea is to work adding words to the lexicon by **frequency** of appearance in whichever corpus you have. You might have easy access to a corpus, if not you can use a Wikipedia database backup dump (https://dumps.wikimedia.org/backup-index.html) or you can write a webcrawler for some online newspaper. Here is a simple `bash` line to make a frequency list given that you have your Chuvash plain text corpus in a file called `chv.crp.txt`:

```
$ cat chv.crp.txt  | sed 's/[^а-яăĕăĕççА-ЯĂĔĂĔÇÇ]\+/ /g' | tr ' ' '\n' | sort -
```

The output will look something like the list on the left. The column in the middle gives the part of speech and gloss in English. While the column on the right shows a graphical representation of Zipf's law.



4175	те	CCONJ	"and"
3212	тат	CCONJ	"and"
2876	та	CCONJ	"and"
1833	вăл	PRON	"it, she, he"
1669	пир	ADP	"about"
1441	уйă	NOUN	"of month"
1404	мĕш	NUM	"on the Nth"
1360	тăр	NOUN	"zone, region
1323	мар	ADV	"not"
1291	пул	VERB	"was"
1273	çул	NOUN	"year"
1133	май	NOUN	"possibility"
1062	пĕр	NUM	"one"
1055	мĕш	NUM	"th"
1020	тес	VERB	"saying"
973	Вăл	PRON	"it, she, he"
967	пĕл	VERB	"happened"

As you can see the most frequent words are unsurprising for a text from a newspaper, words about time and place, closed category words like conjunctions, adpositions and pronouns and copula verbs. This was from a corpus of around 367,168 tokens, so the list won't be so representative for the language as a whole, but the bigger and more balanced the corpus the more legit the frequency list is going to be and the better it is going to represent the language that your analyser is likely to be asked to process.

More on morphotactics

When moving from making a morphological model of a few words from a grammar to making one that covers a whole corpus you are bound to come across some examples of things that are either underdescribed or not described at all in the extant grammars. For example, how loan words are treated and how to deal with numerals or abbreviations which take affixes according to how the words are pronounced and not how they are written.

Loan words

Let's take the example of the French → Russian → Chuvash loan word *специалист* "specialist". In Chuvash this takes back harmony affixes for genitive and other cases instead of the front harmony ones it should orthographically take going by the last vowel *-и-*. For example, the genitive form is *специалистăн* and not **специалистĕн*.

So, how do we deal with this? The easiest way is to come up with some symbols to let the phonological rules know that the word should be treated differently. These will not show up in the surface form but will be available in the morphotactic form. For example, we might define the multicharacter symbol `%{ъ%}` to force back harmony, and then use it next to the morphotactic side in the lexicon:

```
специалист:специалист%{ъ%} N ; ! "специалист"
```

We could then in our phonological rules add this symbol to our `BackVow` set:

```
BackVow = ă а ы о у я ё ю %{ъ%} ;
```

Which would result in the right vowel harmony variant being chosen by our harmony rule:

```
"Back vowel harmony for archiphoneme {Ă}"
%{Ă%}:ă <=> BackVow: [ ArchiCns: | Cns: | %>: ]+ _ ;
```

Similar examples can be found in other languages, for example in some languages consonant clusters may be simplified, Kazakh *съезд-GA → съезге* "к съезду", "to the congress" or epenthetic vowels may be inserted or Kyrgyz, *диалект-DA → диалектиде* "на диалекте", "in dialect".

Numerals and abbreviations

Numerals and abbreviations present a similar problem to loan words, but unlike loan words they do not (necessarily) provide us with any information at all about how the word should be pronounced. Take a look at these following example:

Сăмах	май,	«СУМ»	электрон	лавккара	чăвашла	открыткăсен	йышĕ	18-тан	та	иртрĕ	ĕнтĕ.
Word	by,	«SUM»	electronic	shop-LOC	Chuvash	greeting.card-PL-GEN	amount	18-ABL	and	pass	alread

"By the way, in the «SUM» online shop more than 18 greetings cards have already come out in Chuvash."

Let's consider the ablative morpheme, *-TAн*. After a stem, this can appear as *-тан*, *-тен*, *-ран*, *-рен* depending on the phonology of the stem.

- *-ран*, *-рен* after a vowel or consonant excluding *-н*, *-л*, and *-р*
- *-тан*, *-тен* after the consonants *-н*, *-л*, or *-р*

So, what tells us that it should be *-тан* after 18 ? Well, basically the way that the numeral is pronounced, *вун саккăр* is "eighteen" in Chuvash (lit. ten eight). How do we deal with this? Well, the way is to create a set of symbols, similar to the back vowel symbol that cover all the possible cases. We need at least:

- Front vowel, back vowel
- *-н*, *-л*, or *-р* consonant, other consonant

Thus we could have `%{э%}` for front vowel and `%{а%}` for back vowel and `%{л%}` for *-н*, *-л*, or *-р* and `%{с%}` for other. Here is an excerpt from a transducer in `lexc` that handles numeral expressions:

```
LEXICON NUM-DIGIT

%<num%>:%- CASE ;

LEXICON LAST-DIGIT

1:1%{э%}%{л%}    NUM-DIGIT ; ! "пӗр"
2:2%{с%}%{э%}    NUM-DIGIT ; ! "иккӗ"
3:3%{с%}%{э%}    NUM-DIGIT ; ! "виççӗ"
4:4%{с%}%{а%}    NUM-DIGIT ; ! "тӑваттӑ"
5:5%{э%}%{с%}    NUM-DIGIT ; ! "пиллӗк"
6:6%{с%}%{а%}    NUM-DIGIT ; ! "улттӑ"
7:7%{с%}%{э%}    NUM-DIGIT ; ! "çиччӗ"
8:8%{э%}%{л%}    NUM-DIGIT ; ! "саккӑр"
9:9%{э%}%{л%}    NUM-DIGIT ; ! "тӑххӑр"

LEXICON LOOP

                 LAST-DIGIT ;
                 DIGITLEX ;

LEXICON DIGITLEX

%0:%0 LOOP ;
1:1   LOOP ;
2:2   LOOP ;
3:3   LOOP ;
4:4   LOOP ;
5:5   LOOP ;
6:6   LOOP ;
7:7   LOOP ;
8:8   LOOP ;
9:9   LOOP ;
```

It is not complete but it should give you enough pointers to be able to implement it. Note that in the `Alphabet` in your `twol` file you can specify these kind of symbols as always going to `0`:

```
Alphabet
   а ӑ е ё ӗ и о у ӳ ы э ю я б в г д ж з к л м н п р с ç т ф х ц ч ш щ й ь ъ
   А Ӑ Е Ё Ӗ И О У Ӳ Ы Э Ю Я Б В Г Д Ж З К Л М Н П Р С Ç Т Ф Х Ц Ч Ш Щ Й Ь Ъ

 %{э%}:0 %{л%}:0 %{с%}:0 %{а%}:0

 %{А%}:а %{А%}:е
 %{Ӑ%}:ӑ %{Ӑ%}:ӗ %{Ӑ%}:0
 %{н%}:н %{н%}:0
 %{м%}:м %{м%}:0
```

This saves you writing rules to make them correspond with nothing on the surface.

Unit testing

Once you have got to this stage, you will probably have noticed that it gets increasingly difficult to write rules without knowing what you are breaking. It becomes increasingly useful to have a way of testing rules as you are writing, and possibly even to focus on working by doing *test-driven development* [rus (https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7 %D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%87%D0%B5 %D1%80%D0%B5%D0%B7_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80 %D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)] [eng (https://en.wikipedia.org /wiki/Test-driven_development)].

There are no pre-made testing frameworks for finite-state transducers to my knowledge, but it is fairly easy to come up with one of your own using a combination of `.tsv` files and a script written in the Python programming language. A sample `.tsv` file might look like:

```
_	урам<n><nom>	урам
_	урам<n><gen>	урамӑн
>	урам<n><gen>	урамӑн
_	урам<n><pl><nom>	урамсем
_	урам<n><pl><gen>	урамсен
```

Where the first column is an indication of direction restriction (e.g. should we test this for analysis, >, generation < or both _), the second column is the lexical form of the word and the third column is the surface form of the word. You can use the Python bindings (see below) to write a script that runs the file through your analyser and checks that you get the right results.

Evaluation

As with all tasks, there are various ways to evaluate morphological transducers. Two of the most popular are *naïve* coverage and precision and recall.

Coverage

The typical way to evaluate coverage as you are developing the transducer is by calculating the *naïve* coverage. That is we calculate the percentage of tokens in our corpus that receive at least one analysis from our morphological analyser. We can do this easily in `bash`, for example:

```
$ total=`cat chv.crp.txt  | sed 's/[^а-яӑӗăĕҫçА-ЯӐӖĂĔҪÇ]\+/ /g' | tr ' ' '\n'
$ unknown=`cat chv.crp.txt  | sed 's/[^а-яӑӗăĕҫçА-ЯӐӖĂĔҪÇ]\+/ /g' | tr ' ' '\n'
$ calc "(($total-$unknown)/$total)*100"
        ~0.12990425758725153332
```

So our current Chuvash analyser has a coverage of 0.12%... not very impressive yet!

Alternatively it would be possible to write a Python script. A good rule of thumb is that given an an average sized corpus (under one million tokens) for averagely inflected language (from English to Finnish) 500 of the most frequent stems should get you around 50% coverage, 1,000 should get you around 70% coverage, and 10,000 should get you around 80% coverage. Getting 90% coverage will mean adding around 20,000 stems.

Precision and recall

The second way is to calculate precision, recall and F-measure like with other NLP tasks. To calculate this you need a *gold standard*, that is a collection of words annotated by a linguist and native speaker which contain all and only the valid analyses of the word. Given the gold standard, we can calculate the precision and recall as follows:

- Precision, *P* = number of analyses which were found in both the output from the

morphological analyser and the gold standard, divided by the total number of analyses output by the morphological analyser
- Recall, *R* = number of analyses found in both the output from the morphological analyser and the gold standard, divided by the number of analyses found in the morphological analyser plus the number of analyses found in the gold standard but not in the morphological analyser.

Thus, suppose we had the following:

Form	Gold standard	Analyser output
парăм	`пар<n><px1sg><nom>` "my pair" `парăм<n><nom>` "duty"	`парăм<n><nom>`
урамсен	`урам<n><pl><gen>` "the streets'"	`урам<n><pl><gen>` `ура<n><px1sg><pl><gen>`
çулăм	`çулăм<n><nom>` "flame" `çул<n><px1sg><nom>` "my way"	`çул<n><px1sg><nom>`

We would get:

- $\mathrm{P} = 3/4 = 0.75$
- $\mathrm{R} = 3/(4{+}2) = 0.6$
- $F1 = 2\,\frac{PR}{P+R} = 2\,\frac{0.75 \cdot 0.6}{0.75+0.6} = 0.66$

There is a script evaluate-morph (https://ftyers.github.io/2017-КЛ_МКЛ/scripts/evaluate-morph.py) that will calculate precision, recall and F-score for output from an analyser for a set of surface forms.

As these kind of collections are hard to come by — they require careful hand annotation — some people resort to producing a *pseudo*-gold standard using an annotated corpus. However, note that this may not contain all of the possible analyses so may misjudge the quality of the analyser by penalising or not giving credit to analyses which are possible but not found in the corpus.

In all cases a random sample should be used to ensure coverage of cases from across the frequency range.

Generating paradigms

A finite-state transducer can be used in addition to produce either paradigm tables or all the forms of a given word. First we produce a transducer which accepts the stem we are interested in plus the lexical category and any number of other tags, `?*`:

```
$ echo "у р а м %<n%> ?*" | hfst-regexp2fst -o uram.hfst
```

Then we compose intersect the whole morphological generator with this transducer to find only those paths which start with that prefix:

```
$ hfst-compose-intersect -2 chv.gen.hfst -1 uram.hfst | hfst-fst2strings
hfst-compose-intersect: warning:
урам<n><nom>:урам
урам<n><gen>:урамӑн
урам<n><ins>:урампа
урам<n><pl><gen>:урамсен
урам<n><pl><nom>:урамсем
урам<n><pl><ins>:урамсемпе
урам<n><der_лӑх><nom>:урамлӑх
урам<n><der_лӑх><gen>:урамлӑхӗн
урам<n><der_лӑх><ins>:урамлӑхпе
урам<n><der_лӑх><pl><gen>:урамлӑхсен
урам<n><der_лӑх><pl><nom>:урамлӑхсем
урам<n><der_лӑх><pl><ins>:урамлӑхсемпе
```

This is less useful than it may appear because it also produces all of the derived forms. If we want to just include the inflected forms we can just list the forms we want to generate for a given part of speech in a file, for example:

```
$ cat noun-paradigm.txt
%<n%> %<nom%>
%<n%> %<gen%>
%<n%> %<ins%>
%<n%> %<pl%> %<gen%>
%<n%> %<pl%> %<nom%>
%<n%> %<pl%> %<ins%>
```

Then compile this file into a transducer in the same way:

```
$ cat noun-paradigm.txt | sed "s/^/у р а м /g"  | hfst-regexp2fst -j > uram.hfs
```

Then finally compose and intersect with our transducer:

```
$ hfst-compose-intersect -2 chv.gen.hfst -1 uram.hfst | hfst-fst2strings
урам<n><nom>:урам
урам<n><gen>:урамӑн
урам<n><ins>:урампа
урам<n><pl><gen>:урамсен
урам<n><pl><nom>:урамсем
урам<n><pl><ins>:урамсемпе
```

This kind of full-form paradigm is useful for debugging, and also for generating training data for fancy machine-learningy type things. It can also help with low-resource dependency parsing (systems like UDpipe (https://github.com/ufal/udpipe) allow the use of external dictionaries at training time).

Guessers

A finite-state guesser can also be integrated fairly easily into our transducer. A guesser can have two uses, if you are at an early stage of development, you can use the guesser to get stem candidates from a corpus. If you are at an advanced stage of development you can use the guesser to boost the coverage of your transducer at the expense of accuracy.

To make the guesser work we need three parts:

- A regular expression defining the pattern of symbols that can be considered a stem
- A pattern restricting which morphological forms should not be guessed (often those where there is no inflection) — in Turkic languages often the nominative singular for nouns and 2nd person singular imperative for verbs.
- An special symbol in the `.lexc` file that we are going to inject our stem regular expression into

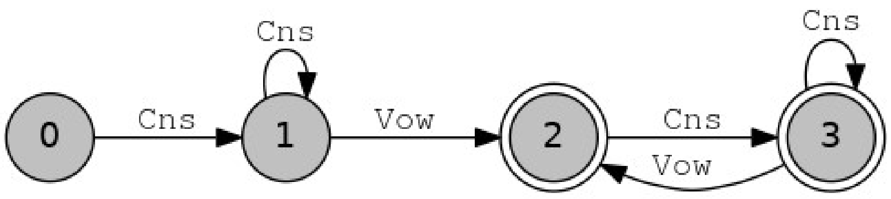
Let's take a look. Start out with the stem regular expression. The format for defining this is a little different from what we have seen so far, but it should be fairly familiar. Make a new file called `chv.stem.regex`

```
define Vow [ ă | а | ы | о | у | я | ё | ю | ĕ | э | и | ӳ ] ;

define Cns [ б | в | г | д | ж | з | к | л | м | н | п | р | с | ç | т | ф | х

define Syll [ Cns+ Vow Cns* ] ;

define Stem [ Syll+ ]  ;

regex Stem ;
```

Now compile it using the following command:

```
$ echo -e "source chv.stem.regex\nsave stack chv.stem.hfst\nhyvästi" | hfst-xfs
hfst[0]: source chv.stem.regex
 Defined
 'Vow'
Defined
 'Cns'
Defined
 'Syll'
Defined
 'Stem'
? bytes. 4 states, 120 arcs, ? paths
hfst[1]: save stack chv.stem.hfst
hfst[1]: hyvästi
.
```

This opens the regex, compiles it, saves it then quits the `hfst-xfst` interface. Like all good computer programs, `hfst-xfst` understands some Finnish. The word *hyvästi* means "farewell". The transducer should look something like:



The next step is to edit your Chuvash `.lexc` file, and add the following lines:

```
LEXICON Root

Guesser ;


LEXICON Guesser

🂡%<guess%>:🂡 N ;
```

This defines a symbol, 🂡 , which we are going to replace with our stem regex. The symbol is arbitrary, I chose a playing card because it's kind of guessing, but you can choose any symbol that will only be found once in your transducer. Now compile your `.lexc` file and then run the following command:

```
$ hfst-substitute -i chv.lexc.hfst -o chv.lexc_guesser.hfst  -f '🂡:🂡' -T chv.st
```

This substitutes the arc transition 🂡:🂡 with the whole stem regex. You can try it out by comparing the transducer files before and after:

```
$ echo "ва<guess><n><ins>" | hfst-lookup -qp chv.lexc.hfst
ва<guess><n><ins>       ва<guess><n><ins>+?     inf

$ echo "ва<guess><n><ins>" | hfst-lookup -qp chv.lexc_guesser.hfst
ва<guess><n><ins>       ва>п{А} 0,000000
```

As you can see, afterwards we get a guessed stem. The next step is to compose our lexicon including the guesser with the phonological rules. You probably know how to do that by now:

```
$ hfst-compose-intersect -1 chv.lexc_guesser.hfst -2 chv.twol.hfst -o chv.gen_g
```

We can try it out by trying to generate something that isn't in our lexicon, for example a form of the Chuvash word *лайк* "like":

```
$ echo "лайк<guess><n><pl><ins>" | hfst-lookup -qp chv.gen_guesser.hfst
лайк<guess><n><pl><ins> лайксемпе       0,000000
```

The next step is to restrict remove the ability to predict nominative singular and to reorder the `<guess>` tag to the end. Make a new file called `chv.restrict.regex`,

```
~[?* "<guess>" "<n>" "<nom>" ] .o. [  [..] -> "<guess>" || "<guess>" ?* _ .#. .
```

This is a pretty gnarly piece of code. The first line rejects any string that contains the tag sequence `<guess><n><nom>`. This is then composed with a regex which takes the `<guess>` tag and moves it to the end. We can compile this as follows:

```
$ hfst-regexp2fst chv.restrict.regex -o chv.restrict.hfst
```

And now prepare our morphological analyser by first inverting the generator and then applying the restrictions:

```
$ hfst-invert chv.gen_guesser.hfst | hfst-compose -1 - -2 chv.restrict.hfst -o
```

And finally we can test it:

```
$ echo "лайксемпе" | hfst-lookup -qp chv.mor_guesser.hfst
лайксемпе       лайк<n><pl><ins><guess> 0,000000
```

You will note how it will generate a guess analysis even if the stem is in the lexicon. How do you think you might be able to avoid this analysis ?

Weighting

Surface forms

An unannotated machine-readable text corpus of a language is usually fairly easy to come by... given the language has some kind of orthography. So, what can you use it for ? Well, you could use it to arbitrate which form is better to generate in the case of having free variation. For example, in Chuvash the *-сть* in Russian loanwords in the nominative singular can be written as *-ç* or *-сть*.

One possibility would be to have a lexicon like:

```
LEXICON N/сть

%<n%>:ç SUBST "weight: 0.5" ;
%<n%>%<nom%>:сть # "weight: 1.0" ;
```

Which would always prefer the form with *-ç*. But perhaps it is lexicalised or depends on some other factors. Let's start by making a frequency list:

```
$ cat chv.crp.txt  | sed 's/[^а-яăĕăĕççА-ЯĂĔĂĔÇÇ]\+/ /g' | tr ' ' '\n' | sort -
```

We can convert this frequency list into a format suitable for building a weighted transducer using the following Python code:

```
import sys, math as maths
f = {}; fs = []
total = 0
for line in sys.stdin.readlines():
        row = line.strip().split(' ')
        if len(row) < 2: continue
        form = row[1]
        freq = int(row[0])
        fs.append(form)
        f[form] = freq
        total += freq
for form in fs:
        print('%s\t%.4f' % (form, -maths.log(f[form]/total)))
```

If we call this `freq2prob.py` we can call it as follows:

```
$ cat chv.freq.txt | python3 freq2prob.py | hfst-strings2fst -j -o chv.surweigh
```

Check that it works:

```
$ echo "область" | hfst-lookup -qp chv.surweights.hfst
область область 11,393500

$ echo "облаç" | hfst-lookup -qp chv.surweights.hfst
облаç   облаç   9,977600
```

The next thing we need to do is add a path in the weight transducer for unknown words (those words that are not found in the corpus). This should be the maximum weight of any word in the transducer.

```
$ echo "?::0" | hfst-regexp2fst | hfst-repeat | hfst-reweight -e -a 15.0 | hfst
```

Then we union the max weight transducer with the surface weights:

```
$ hfst-union -1 chv.surweights.hfst -2 chv.maxweight.hfst -o chv.weights.hfst
```

And compose this all with the surface side of the generator:

```
$ hfst-compose -1 chv.gen.hfst -2 chv.weights.hfst -o chv.gen_weighted.hfst
```

Analyses

Let's imagine for a second that we have a massive gold standard annotated corpus of Chuvash. We could use that to assign different weights to the different analyses of our surface forms. For example our generator might look like:

```
$ hfst-fst2strings chv.lexc.hfst
парăм<n><nom>:парăм
пар<n><px1sg><nom>:парăм
```

Let's say that we saw *парăм* as "duty" 150 times in our corpus and *парăм* as "my pair" 3 times. It would be fairly straightforward to write a Python script to convert these into a file with weights:

```
парăм<n><nom>:парăм     0.01980
пар<n><px1sg><nom>:парăм        3.9318
```

Remember we define weights as negative log probabilities, e.g. $w = -\log(r)$ where *r* is the relative frequency in the corpus.

```
$ cat chv.weights | hfst-strings2fst -j -m chv.symbols -o chv.strweights.hfst
$ echo "?::5.13579" | hfst-regexp2fst | hfst-repeat -o chv.maxweight.hfst
$ hfst-union -1 chv.strweights.hfst -2 chv.maxweight.hfst -o chv.weights.hfst
```

The file `chv.symbols` is a file with a list of multicharacter symbols (e.g. `<n>`) specified one per line.

```
$ hfst-invert chv.lexc.hfst | hfst-compose -2 chv.weights.hfst | hfst-invert |
```

Python bindings

It's completely possible to use HFST transducers in Python by using the Python bindings, the following code loads a transducer. There is also a nice quick start guide (https://hfst.github.io/python/3.12.1/QuickStart.html) on the HFST site.

```
import hfst

ifs = hfst.HfstInputStream('chv.gen.hfst') # set up an input stream
transducer = ifs.read()                    # read the first transducer
transducer.invert()                        # invert the transducer
transducer.lookup('урамӑн')                # analyse a token
```

Final thoughts

Congratulations if you got this far! ❤ FSTs

Troubleshooting

If you are not getting any error messages (like rule conflicts or compile errors) and your code looks like it should work, but it still isn't working, try checking the following:

- Check that you have defined all of your multicharacter symbols.
- Check that you are looking at the right unicode codepoints. Some characters look the same, but are represented differently by your computer. For example,
  - «ă» → `U+0103 LATIN SMALL LETTER A WITH BREVE (ă)`
  - «ӑ» → `U+04D1 CYRILLIC SMALL LETTER A WITH BREVE (ӑ)`
- Check to make sure you don't have any weird non-printing space characters like,
  - « » → `U+00A0 NO-BREAK SPACE`

Further reading

- Kimmo Koskenniemi (1983) *Two-level morphology : a general computational model for word-form recognition and production*. Publications (Helsingin yliopisto. Yleisen kieliteteen laitos 11)
- Kenneth R. Beesley and Lauri Karttunen (2003) "Two-level morphology (https://web.stanford.edu/~laurik/.book2software/twolc.pdf)" in *Finite State Morphology* (CLSI: Stanford)
- HFST Team (2017) hfst-twolc − A Two-Level Grammar Compiler (https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstTwolC)
- Lauri Karttunen (1991) Finite-State Constraints (https://web.stanford.edu/~laurik/publications/fsc-91/fsc91.html)