Правительство Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования
Национальный исследовательский университет
«Высшая школа экономики»

Факультет гуманитарных наук
Образовательная программа
«Фундаментальная и компьютерная лингвистика»

Картина Элен Геннадьевич

Правиловый морфологический парсер для шугнанского языка: существительные, глаголы и прилагательные

Выпускная квалификационная работа студента 4 курса бакалавриата группы БКЛ211

Академический руководитель образовательной программы
канд. филологических наук, доц.
Ю.А. Ландер

____

«_____» _____________ 2025 г.

Научный руководитель
канд. филологических наук, доц.
Г.А. Мороз

____

Научный консультант
Стажёр-исследователь
М.Г. Меленченко

____

Москва 2025

Abstract

In this work I present a rule-based morphological analysis tool based on Helsinki Finite-State Technology (HFST) for the Shughni language (ISO: sgh; glottocode: shug1248), a language of the Iranian branch of the Indo-European family, a member of 'Pamiri' areal language group. While one existing rule-based parser exists for Shughni (Melenchenko, 2021), it does not utilize finite-state transducer technology **TODO: И что? Я бы перечисли какие-то конкретные недостатки, чтобы мотивировать работу.**. This work proposes the first HFST-based morphological parser implementation for Shughni, offering the advantages of this well-established framework for morphological analysis. The parser is presented in two variations: a morphological parser that breaks each word-form into stem and morphemes and assigns morphological tags to each one of them; a morphological generator that outputs word-forms taking a stem and morphological tags as an input. **TODO: prev sentence is questionable** This is a continuation of my previous work, where nouns, pronouns, prepositions and numerals were implemented (Osorgin, 2024). This project covers **TODO: what**

TODO: Review abstract after finishing the work

Contents

1 Methods

1.1 Input and output format

Every FST converts between two types of strings: wordforms (e.g. ‘дарйойен’ ≈ ‘rivers’) and glossed strings (e.g. ‘дарйо<n>><pl>’ = ‘дарйо.N-PL’, POS tag and gloss descriptions are listed in Tables **??**, **??** and **??**). Glossed strings format is based on Apertium format, which is a standardized format for HFST-based transducers. The choice of this format is motivated by the fact that this is the standard, and I want to keep it consistent with existing tools and practices in the field. Important key points of the format are:

- Grammatical tags are enclosed with angular brackets and are lowercase.
  stone.V = stone<v>
- Part of speech tags (POS) are obligatory and stand next to the stem or lemma.
  stone.SG = stone<n><sg>
- Different morpheme tags are separated with a single right angular bracket ‘>’.
  stone-PL = stone<n>><pl>
- Multiple stems in a single word are possible.
  stem1<adj>><morph>>stem2<adj>

The Shughni morphological parser in this work is presented in a variety of input and output formats. A full list of .hfst files with their formats of input and output is shown in Table 1. Motivation for this many variations of the morphological parser was support of different formats. Parser comes with four format variables:

- FST directionality: analyzer or generator. This simply shows the direction of a FST, analyzers take wordforms as input and return glossed strings, generators take glossed strings and return wordforms
- Glossed string stem glosses: Shughni stems or Russian lemmas. Notated in file names as stem and rulem respectively.
  дарйо<n>><pl> (Shughni stem); река<n>><pl> (Russian lemma)
- Wordform morpheme segmentation: plain word or segmented word (morphemes separated with (‘>’) delimeter symbol). Notated in file names as word and segm respectively.
  дарйойен (plain word); дарйо>йен (segmented word)
- Wordform script: Latin or Cyrillic.
  дарйойен (Latin); daryoyen (Cyrillic)

Four binary variables result in 16 ($= 2^4$) FST variations. Every FST listed in Table 1 can be built with make command as shown in Code block 1. Regular .hfst binary FSTs are not recommended using in production environments, as they are not optimized. For production use an optimized format called .hfstol (HFST **O**ptimized **L**ookup), which can also be compiled automatically for every listed FST using make.

Transducer file name	Input example	Output example
sgh_gen_stem_segm_cyr.hfst	`дарйо<n>><pl>`	`дарйо>йен`
sgh_gen_stem_word_cyr.hfst	`дарйо<n>><pl>`	`дарйойен`
sgh_gen_rulem_segm_cyr.hfst	`река<n>><pl>`	`дарйо>йен`
sgh_gen_rulem_word_cyr.hfst	`река<n>><pl>`	`дарйойен`
sgh_gen_stem_segm_lat.hfst	`дарйо<n>><pl>`	`daryo>yen`
sgh_gen_stem_word_lat.hfst	`дарйо<n>><pl>`	`daryoyen`
sgh_gen_rulem_segm_lat.hfst	`река<n>><pl>`	`daryo>yen`
sgh_gen_rulem_word_lat.hfst	`река<n>><pl>`	`daryoyen`
sgh_analyze_stem_segm_cyr.hfst	`дарйо>йен`	`дарйо<n>><pl>`
sgh_analyze_stem_word_cyr.hfst	`дарйойен`	`дарйо<n>><pl>`
sgh_analyze_rulem_segm_cyr.hfst	`дарйо>йен`	`река<n>><pl>`
sgh_analyze_rulem_word_cyr.hfst	`дарйойен`	`река<n>><pl>`
sgh_analyze_stem_segm_lat.hfst	`daryo>yen`	`дарйо<n>><pl>`
sgh_analyze_stem_word_lat.hfst	`daryoyen`	`дарйо<n>><pl>`
sgh_analyze_rulem_segm_lat.hfst	`daryo>yen`	`река<n>><pl>`
sgh_analyze_rulem_word_lat.hfst	`daryoyen`	`река<n>><pl>`

Table 1: A full list of available HFST transducers

```
$ make sgh_gen_stem_segm_cyr.hfst
$ make sgh_gen_stem_segm_cyr.hfstol
```

Code 1: Example of FST compilation with `make`.

1.2 FST compilation pipeline

Analyzers and generators

First, there are two main types of FSTs: generators and analyzers. They differ only in the directionality of a FST. Analyzers take wordforms as input and return glossed strings as output. Generators work in reverse, as shown in Code block 2.

```
$ echo "дарйойен" | hfst-lookup -q sgh_analyze_stem_word_cyr.hfst
дарйойен        дарйо<n>><3pl>  0.000000
дарйойен        дарйо<n>><pl>   0.000000
$ echo "дарйо<n>><pl>" | hfst-lookup -q sgh_gen_stem_word_cyr.hfst
дарйо<n>><pl>   дарйо-йен       0.000000
дарйо<n>><pl>   дарйойен        0.000000
```

Code 2: FST analyzer vs generator output formats.

The `lexd` source code is written as a generator, meaning by default, compiled FST takes glossed stem or lemma as input and returns a wordform. To compile any analyzer, a corresponding generator is inverted, as shown in Code block 3.

```
$ hfst-invert sgh_gen_stem_word_cyr.hfst  \
           -o sgh_analyze_stem_word_cyr.hfst
```

Code 3: FST analyzer creation from a FST generator.

Shughni stems and Russian lemmas

The next format only applies to the glossed side of FSTs (to the analyzers' output and to the generators' input). It sets whether a Cyrillic Shughni stem or a Russian translated lemma will be used as a stem's gloss, as shown in Code block 4. Shughni stems can have multiple Russian candidates (*'дарйо'* can be translated as *'река'* or *'море'*). This leads to composed transducer having more output candidates. This works both ways, meaning Russian lemmas can translate as multiple Shughni stems (*'река'* can be translated as *'дарйо'* or *'х̌ац'*).

```
$ echo "дарйо<n>><3pl>" | hfst-lookup -q sgh_gen_stem_word_cyr.hfst
дарйо<n>><3pl>      дарйо-йен       0.000000
дарйо<n>><3pl>      дарйойен        0.000000
$ echo "река<n>><3pl>" | hfst-lookup -q sgh_gen_rulem_word_cyr.hfst
река<n>><3pl>       дарйо-йен       0.000000
река<n>><3pl>       дарйойен        0.000000
река<n>><3pl>       х̌ац-ен          0.000000
река<n>><3pl>       х̌ацен           0.000000
$ echo "дарйойен" | hfst-lookup -q sgh_analyze_rulem_word_cyr.hfst
дарйойен            море<n>><3pl>   0.000000
дарйойен            море<n>><pl>    0.000000
дарйойен            река<n>><3pl>   0.000000
дарйойен            река<n>><pl>    0.000000
```

Code 4: Shughni stem vs Russian lemma versions of FST.

The `lexd` source code contains lexicons with Shughni stems on the glossed side, meaning default compiled FST contains only Shughni stems. The process of creating a FST that works with Russian lemmas on the glossed side is more complicated. It is achieved with the help of a second FST `rulem2sgh.hfst`, its only purpose is translating stems to lemmas. It is attached to the input of a generator FST, creating a pipeline

'`река<n>><pl>`' → `rulem2sgh` → '`дарйо<n>><pl>`' → `generator` → '`дарйойен`'

It can be done with 'compose' transducer operation (see Code block 5), which takes two FSTs, directs first's output to the second's input and returns the resulting composed FST. Details of the translator FST's development are described in Section 1.7.

```
$ hfst-compose rulem2sgh.hfst sgh_gen_stem_word_cyr.hfst
  -o sgh_gen_rulem_word_cyr.hfst
```

Code 5: Shughni stem translator composition.

Latin script support

The source code contains only Cyrillic lexicons. The support of Latin script comes with the help of a separate transliterator FST, as it was the case for Russian lemmas support. Transliteration is only applied to the wordform side of FSTs. The glossed side's stems and lemmas are left Cyrillic. The pipeline for transliteration is shown below:

‘дарйо<n>><pl>’ → generator → ‘дарйойен’ → cyr2lat → ‘daryoyen’
‘daryoyen’ → lat2cyr → ‘дарйойен’ → analyzer → ‘дарйо<n>><pl>’

The compilation process is the same as it is for Russian lemmas shown in Code block 5. The only difference is that transliterator is applied to the wordform side of a FST, while Russian lemma translator is applied to the glossed side. See Section 1.6 for more details about the transliterator development.

Wordform segmentation

Theoretical linguists sometimes need wordforms to have morphemes separated by a special symbol that does not appear naturally in a language. In this morphological parser I choose the right angular bracket symbol ’>’ for this role to keep it consistent with glossed strings morpheme separator. This presented a slight challenge to the development, as regular wordforms often may contain hyphens (‘-’) between some morphemes. For an example, ‘дарйойен’ can also be spelled as ‘дарйо-йен’. But for the morpheme separated FST wordform must contain only ‘дарйо>йен’, with no extra optional hyphens.

This was solved with the help of twol rules. The base lexd FST (sgh_base_stem.hfst) contains both hyphens and morpheme separators, which looks like ‘дарйо>-йен’ and ‘дарйо>йен’ on the wordform side. Then the filtering is done with help of two FSTs that contain twol rules shown in Code block 6. The sep.hfst removes all separator symbols (‘>’) from the wordform, leaving ‘дарйо-йен’ and ‘дарйойен’. And hyphen.hfst removes all hyphens from the wordform, leaving ‘дарйо>йен’ and ‘дарйо>йен’, which then fold into a single FST path after optimization.

```
# hyphen.twol
Rules
"Hyphen removal to avoid '>-'/'->' situations"
%-:0 <=> _ ;

# sep.twol
Rules
"Morpeme separator removal"
%>:0 <=> _ ;
```

Code 6: twol rules that filter morpheme border.

For this case compose-intersect operation is applied, that allows twol-compiled FSTs to be composed with regular lexicon lexd FSTs. Compilation command is shown in Code block 7.

This way, every FST listed in Table 1 can be compiled using a specific combination of bash HFST tools shown in this section.

```
$ hfst-compose-intersect sgh_base_stem.hfst twol/sep.hfst
  -o sgh_gen_stem_word_cyr.hfst
$ hfst-compose-intersect sgh_base_stem.hfst twol/hyphen.hfst
  -o sgh_gen_stem_morph_cyr.hfst
```

Code 7: Plain text and morpheme separated text FST versions compilation.

1.3 `lexd` rule declaration

The choice of lexicon compiler was made in favor of `lexd` as it provides everything that `lexc` does and in addition has some extra useful functional in form of the tag system, which will be taken advantage of.

The `lexd` source code is stored in the `lexd/` directory. I decided to go with a modular file structure for `lexd` source code, as it helps to keep the source code organized. The `lexd/` directory contains `.lexd` source code files with morpheme lexicons (suffixes, clitics, prefixes, etc.) and lexicon combination patterns. Stem lexicons are stored separately in the `lexd/lexicons/` directory. For the most part `lexd/lexicons/` directory contains lexicons obtained from database dumps provided by Makarov et al. (2022). The stem lexicon processing is described in Section 1.7.

There is no module import feature in `lexd`. So in order to be able to make a modular `.lexd` source file structure compilable into a single `.hfst` file we can concatenate every `.lexd` module into a single large temporary `.lexd` file and feed it to the compiler. This is achieved with `bash` command shown in Code block 8. The `lexd` compiler outputs FST in AT&T format and `hfst-txt2fst` converts it to a binary `.hfst` file.

```
$ cat lexd/*.lexd lexd/lexicons/*.lexd > sgh.lexd
$ lexd sgh.lexd | hfst-txt2fst -o sgh_base_stem.hfst
```

Code 8: Bash command pipeline compiling multiple `.lexd` files into a single FST.

In the sections below I will describe some important `lexd` decisions, that had to be made. As well as general inflectional and derivational information.

Global patterns

As was mentioned before, by default, `lexd` in this project is implemented with morpheme segmented wordforms. For this purpose, various wordform separators were defined (see Code block 9). These separator patterns are inserted into morphological patterns between morphemes, resulting in wordforms looking like *'prefix->stem>-suffix'* by default, with hyphens and morpheme separators mixed together. This is later resolved with the help of `twol` in the compilation pipeline, as was shown in Section 1.2. The `_morph_hyphen_` pattern is used for suffixes, and the `_hyphen_morph_` pattern is used for prefixes.

There is also a global lexicon `Adv`, that contains an adverbalizer suffix '-(й)aθ', that supposedly attaches to nouns and adjectives: *'pů̊nd-aθ'*='road-ADV' (Parker, 2023, p. 139), *'zūr-aθ'*='great-ADV' (Parker, 2023, p. 433).

```
PATTERN _morph_
[>:>]

PATTERN _hyphen_
[:-]?

PATTERN _morph_hyphen_
_morph_ _hyphen_

PATTERN _hyphen_morph_
_hyphen_ _morph_

LEXICON Adv
<adv>:{Й}аθ
```

Code 9: Global patterns and lexicons. Remember, that the default FST is implemented as a generator, meaning that on the left side of ':' is a glossed string, and on the right side is a wordform. So hyphens do not show up in glosses strings and are optional on the wordform side by default.

Clitics

Clitics is Shughni for the most part can attach to any part of speech. For example, verbal person agreement in past tense happens with the help of PSC (Personal clitics), which attaches to the right side of the first syntactic constituent in a clause (Parker, 2023, p. 262). This exceeds the scope of morphology, so in order to take this into account, clitics were made global, in our formal model they attach to everything. An example of clitics lexicon is shown in Code block 10. The exact possible order of all different clitics is also unknown, so the final `GlobClitics` pattern contains a rule that states: 'any zero to three clitics'. The downside is that this significantly increases overgeneration. There is also a prepositional clitic *'ик-'*(EMPH), that is not mentioned by Parker (2023), but is described by Karamshoev (1988–1999) that is implemented as global.

Nouns

Noun inflection implementation consists of number and case, number inflection is shown in Code block 11. A plural suffix *'-(й)ен'* applies to every singular noun form. Other plural suffixes only attach to different semantic categories such as in-law family members (Parker, 2023, p. 148). Parker also lists a plural suffix *'-(а)чев'* that applies to times of day and year, creating a meaning 'in the evenings' if combined with *'evening'* stem. But it conflicted with the internal unpublished glosses of the HSE expeditions to Tajikistan, where it was listed as a derivational suffix glossed 'TIME'. I decided to stick to the field glosses, as these researchers are the target users of this morphological parser.

Irregular forms of plural nouns were also extracted from the digital dictionary (Makarov et al., 2022) and put into `LexiconNounPlIrregular` lexicon. In the dictionary both regular and irregular plural noun forms are present. Regular ones were filtered out algorithmically by checking if they have any regular suffixes and their stems were put into `LexiconNounPlRegular` lexicon, in case any of them are missing from the `NounBase` lexicon. This was done with the help of `scripts/lexicons/db_dumps/filter.py` script.

Adverbs lexicon `LexiconAdv` is listed here, since they do not have their own POS tag. According to the unpublished field works glosses, the state of adverbs is an open question. I have decided

```
PATTERN GlobClitics
(AnyClitic > AnyClitic > AnyClitic)?

PATTERN AnyClitic
PronClitics
...
FutureClitic

PATTERN PronClitics
_morph_hyphen_ PCS

PATTERN FutureClitic
_morph_hyphen_ FUT

...

LEXICON PCS
<1sg>:{Й}ум
...
<3pl>:{Й}ен

LEXICON FUT
<fut>:та
```

Code 10: A fragment of `lexd/clitics.lexd`. '`...`' is not a part of the source code, here it denotes a content skip in order to take less space and stay informative.

```
PATTERN NounNumberBase
NounBase                                                   [<sg>:]
NounBase                                  _morph_hyphen_   [<pl>:{Й}ен]
NounBase[pl_in-laws]                      _morph_hyphen_   [<pl>:орч]
NounBase[pl_cousins]                      _morph_hyphen_   [<pl>:ӯн]
NounBase[pl_sisters]                      _morph_hyphen_   [<pl>:дзинен]
LexiconNounPlRegular               [<n>:] _morph_hyphen_   [<pl>:{Й}ен]
LexiconNounPlRegular               [<n>:]                  [<sg>:]
LexiconNounPlIrregular             [<n><pl>:]
LexiconAdv                         [<n>:]                  [<sg>:]?
LexiconAdv                         [<n>:] _morph_hyphen_   [<pl>:{Й}ен]
```

Code 11: A segment of `lexd/noun.lexd` showing noun number inflection pattern.

to put it with nouns since at least some are inflected by number as nouns (e.g. *'ахӣб'*='yesterday', *'ахӣб-ен'*='yesterday-PL')(Karamshoev, 1963). This decision is increasing overgeneration, but in my opinion, it is more important to cover such basic lexicon.

Nouns also have several derivational suffixes like the diminutive *'-(й)ик'*/*'-(й)ак'*, the suffix of origin *'-чи'* (*'аммӯм'*='sauna', *'аммӯм-чи'*='sauna-ORIG' ≈ 'sauna operator') and others. The final noun main pattern is shown in Code block 12.

Verbs

There are four base verb stems in Shughni: non-past, past, infinitive and perfect ones. They can be regularly derived from one another, but there are also irregular verb stems. The regular pattern

```
PATTERNS
NounPrefix NounNumberBase NounDeriv? NounSuffix

PATTERN NounPrefix
GlobCliticsPrep (NounPrepos _hyphen_morph_)?

PATTERN NounSuffix
(_morph_hyphen_ NounAdpos)? (_morph_hyphen_ Adv)? GlobClitics
```

Code 12: A segment of `lexd/noun.lexd` showing a final noun morphological pattern.

for infinitive and past stems is the addition of '-т/-д' to the non-past stem. And for the perfect stem is the addition of '-ч/ч' to the non-past stem (Parker, 2023, p. 257). For the inflection, verbs can inflect in person, number, gender and negation. Everything except for gender is done via affixation, while the gender inflection is marked with stem-internal alterations (Parker, 2023, p. 261). Gender inflection unfortunately was not implemented regularly in this work, it was done with the help of digital dictionary, that contained great amount of gender-inflected verb stems.

Person and number inflection is done via suffixation in present stems and via clitics with past stems. The latter is irrelevant for this work, as it exceeds morphology and was taken into account with global clitics. The present stem `lexd` pattern is presented in Code block 13. All the person- and number-specific lexicons like `LexiconVerbNpst1sg` were exported from the digital dictionary and stripped of any inflection. This was done to take into account any irregular stem alterations that were not present in the main `LexiconVerbNpst` lexicon.

```
PATTERN NPastVerbBase
LexiconVerbNpst|LexiconVerbNpstSh   [<v><prs>:]
LexiconVerbNpst                     [<v>:] _morph_ [<prs>:] PresSuffixes
LexiconVerbNpst1sg                  [<v>:] _morph_ [<prs><1sg>:м]
LexiconVerbNpst2sg                  [<v>:] _morph_ [<prs><2sg>:{Й}и]
LexiconVerbNpst2pl                  [<v>:] _morph_ [<prs><2pl>:{Й}ет]
LexiconVerbNpst3sg                  [<v>:] _morph_ [<prs><3sg>:{vДТ}]
LexiconVerbNpst3pl                  [<v>:] _morph_ [<prs><3pl>:{Й}ен]
LexiconVerbNpst1sgSh                [<v><prs><1sg>:]
LexiconVerbNpst2sgSh                [<v><prs><2sg>:]
LexiconVerbNpst2plSh                [<v><prs><2pl>:]
LexiconVerbNpst3sgSh                [<v><prs><3sg>:]
LexiconVerbNpst3plSh                [<v><prs><3pl>:]
LexiconVerbNpstF                    [<v><prs><f>:]
LexiconVerbNpstF3sg                 [<v><prs><f><3sg>:]
```

Code 13: A fragment of `lexd/verb.lexd` showing a non-past verb inflection pattern. `PresSuffixes` contains person-number present inflection suffixes.

Verbs also have short forms, which lexicon names end with `Sh`. They were not stripped of inflection and were left as they are without segmentation, meaning all glosses are considered to belong to the stem.

Other verb stems mostly do not inflect in person or number, but in other aspects they were implemented generally the same with some tense specific minor inflections. The last important details can be shown with perfect stems (see Code block 14). Verbal resultive participle is formed only with masculine perfect stems (Parker, 2023, p. 370). So perfect lexicon had to be split to take this into account. Perfect lexicons also show how irregular stems are handled. The digital dictionary contains perfect verb forms inflected for plural number, which have regular and irregular forms mixed up together. They were split into two separate lexicons with the help of `scripts/lexicons/db_dumps/filter.py` script, that tried to backtrack verb inflection and categorized failed to backtrack verbs as irregular. Regular verb stems were again stripped of inflection, so they can be inflected and segmented later by FST, and irregular verb stems were left untouched.

```
### Perfect ###
PATTERN PerfVerbBase
PerfVerbBaseMasc    (_morph_hyphen_ VerbPluQuamPerf)?
PerfVerbBaseOther   (_morph_hyphen_ VerbPluQuamPerf)?

PATTERN PerfVerbBaseMasc
LexiconVerbPerfM            [<v><prf><m>:]
LexiconVerbNpst             [<v>:] _morph_ [<prf>:{vЧЧ}]

PATTERN PerfVerbBaseOther
LexiconVerbPerfF            [<v><prf><f>:]
LexiconVerbPerfPlRegular    [<v>:] _morph_ [<prf><pl>:{vЧЧ}]
LexiconVerbPerfPlIrregular  [<v><prf><pl>:]

### Participle2 ###
PATTERN Participle2Base
PerfVerbBaseMasc _morph_hyphen_ [<ptcp2>:ак]
```

Code 14: A fragment of `lexd/verb.lexd` showing perfect verb inflection.

Pronouns and demonstratives

Demonstratives and personal pronouns' lexicons were listed manually. The implementation is for the most part straightforward, they inflect in two cases: locative '-(а)нд' and dative '-(а)рд'. A second-person plural personal pronoun can also take a '-йет'(HON) suffix, which is used to address someone honorably.

Common indefinite pronouns were implemented in an elegant way. A list of 20 indefinite pronouns was presented by Parker (2023, Table 6.4). These pronouns were easily converted into a simple pattern, as they basically consisted of combinations of 5 bases and 4 prefixes with some stem irregularities (see Code block 15).

Adjectives

Shughni adjectives besides some inflection can be derived from other parts of speech. As shown in Code block 16, adjectives can be derived from noun stems with the help of suffixes like '-(й)ин'(≈ 'made of') (Parker, 2023, p. 169). In addition, adjectives can take noun stems to form compound adjectives like *'рӯшт-куртā'*='red.ADJ-shirt.N', resulting in an adjective meaning ≈'red shirt wearing' (Parker, 2023, p. 173).

```
LEXICON IndefinitePronRoots # Parker 2023 p.193
чӣз[g1,g2]
чай[g1,g2]
царāнг[g1]
цаwахт[g1]
рāнг[g2]
wахт[g2]
ҷой[g1,g2]

PATTERN IndefinitePronouns # Parker 2023 p.99
[ар]  _hyphen_ IndefinitePronRoots[g1] _hyphen_ [ца]    # Assertive
[йи]  _hyphen_ IndefinitePronRoots[g1]                  # Elective
[йи]  _hyphen_ IndefinitePronRoots[g1] (_hyphen_ [аθ])? # Negative
[фук] _hyphen_ IndefinitePronRoots[g2] (_hyphen_ [аθ])? # Universal
```

Code 15: A `lexd/pron.lexd` fragment showing an implementation of all 20 common indefinite Shughni pronouns.

```
PATTERN AdjBase
LexiconAdj [<adj>:] (_morph_hyphen_ AdjSuffix|Adv)?
### Derivation from Nouns ###
LexiconNoun [<n>:] _morph_hyphen_ NounAdjectivatorsSuffix
NounAdjectivatorsPrefix _hyphen_morph_ LexiconNoun [<n>:]
### Compound Adjectives ###
LexiconAdj [<adj>:] _morph_hyphen_ LexiconNoun [<n>:]
```

Code 16: A `lexd/adj.lexd` fragment.

Numerals

Shughni employs a numeral system that contains both Shughni and Tajik borrowed lexicon (Parker, 2023, p. 176). Both systems are widely used, so they both were implemented. No derivation or inflection is happening with numerals, except for clitic '=ат'(and) used for complex number formation when attaching to *'δūс'*('ten') in Shughni native numeral system.

Other parts of speech

Everything else is implemented in quite straightforward manner: interjections, conjunctions, prepositions, postpositions and particles contain lexicons with only morphological patterns being global clitics attachment.

1.4 `twol` phonology

1.4.1 Global patterns

Shughni is not rich for morphonological rules. The only global phonological rule is 'й' (Latin: 'y'; IPA: /j/) drop on some morphemes' borders after a consonant. In the rule below '>' symbols stands for morpheme border:

$$\rightarrow \emptyset \ / \ [+consonant] > \ \_$$

A special multichar symbol is introduced to stand in place of 'й' letters that are affected by this phonological rule: '{Й}'. We capitalize it and frame it with curly brackets so it is never confused with plain letters. A FST treats multichar symbols as single unique symbols, even though they visually consist of multiple characters. This feature allows us to mark which lexicon entries are affected by this phonological rule and which are not. Examples of lexicon definitions with this special symbol were shown in Section 1.3 (Code blocks 8, 10, 11 and 13).

Without `twol` rules these morphemes will remain as they are specified in `lexd`, meaning that feeding '`вирод<n>><dim>`'(='brother.N-DIM') to the input of a generator will output literally '`вирод{Й}ик`'. After applying a `twol` rule (shown in Code block 17) it generates wordforms '`виродик`' (='brother.N-DIM') and '`туйик`' (='you.PERS.2SG-DIM'). The composition process of `twol` rules with the main FST is the same as it was for morpheme border filtration (Code block 7).

```
"й drop after consonant"
%{Й%}:0 <=> Consonant (%>) (%-) (%>) _ ;

"й drop after consonant"
%{Й%}:й <=> Vowel (%>) (%-) (%>) _ ;
```

Code 17: A `twol` rule for 'й' drop depending on the previous morpheme's segment. Symbol '%' in `twol` is used to escape a character. Morpheme separators are enclosed with brackets, which in `twol` syntax is a way to make anything inside brackets optional.

1.4.2 Verb phonology

Some verb stems can regularly be formed from non-past tense stems. This was briefly discussed in 'Verbs' subsection of Section 1.3. Parker (2023, p. 256) describes two context groups for these rules. The first group contains voiced obstruents, vowels and semivowels 'w'(Latin: 'w'; IPA: /v/) and 'й', these contexts are followed by 'д'(Latin: 'd'; IPA: /d/) for past and infinitive stems and by 'ҷ'(Latin: 'ǯ'; IPA: /d͡ʒ/) for perfect stems. The second group contains everything else and is followed by 'т'(Latin: 't'; IPA: /t/) for past and infinitive and by 'ч'(Latin: 'č'; IPA: /tʃ/) for perfect stems. The `twol` formalism rules for verb stem-forming endings are shown in Code block 18.

1.4.3 Pronoun phonology

Pronouns in Shughni can be inflected in locative and dative case with suffixes '-(а)нд' and '-(а)рд' respectively. The optional 'а' letter also depends on the final letter of the previous segment. The contexts are the same here as for the 'й' drop rule, so the `twol` rules are very similar (see Code block 19).

```
"Past and Inf verb stem endings"
%{vДТ%}:д => [VoicedObstruent | Vowel | Semivowel] (%>) _ ;

"Past and Inf verb stem endings"
%{vДТ%}:т => [VoicelessConsonant | Nasal | Liquid] (%>) _ ;

"Perf verb stem endings"
%{vҶЧ%}:ҷ => [VoicedObstruent | Vowel | Semivowel] (%>) _ ;

"Perf verb stem endings"
%{vҶЧ%}:ч => [VoicelessConsonant | Nasal | Liquid] (%>) _ ;
```

Code 18: `twol` rules for verb endings phonology.

```
"pronouns loc and dat"
%{A%}:a <=> Consonant (%>) (%-) (%>) _ ;

"pronouns loc and dat"
%{A%}:0 <=> Vowel (%>) (%-) (%>) _ ;
```

Code 19: `twol` rules for noun suffixes phonology.

1.5 Stem lexicons processing

Describing morphological rules and morphemes is not sufficient to make a morphological parser. An important part of the development was gathering stem lexicons for various parts of speech. For some POS, like conjunctions or numerals, lexicons were listed by hand. But for others, like verbs, manual lexicon construction is an extremely time-consuming task. Such lexicons were extracted from the `SQLite` database provided by Makarov et al. (2022).

The database in question contains digital versions of Shughni main dictionaries: Karamshoev (1988–1999) and Zarubin (1960). It has a decently organized structure, which allows selecting dictionary entries by parsed metadata like POS or glosses. To transfer stems from the database to a `lexd`-formatted file it follows the following pipeline:

`Database` → `noun.csv` → `form_lexd.py` → `noun.lexd`

Exported `noun.csv` fragment

```
cyrillic,meaning
аббуст,рукавица
аббустеҷ,материал
абед,обед; время обеда
абӣн,ненавистница
абкӯ̊мпāрт,обком партии
```

Generated `noun.lexd` fragment

```
LEXICON LexiconNoun[pl_all,sg]
аббуст      # рукавица
аббустеҷ    # материал
абед        # обед; время обеда
абӣн        # ненавистница
абкӯ̊мпāрт   # обком партии
```

Code 20: An example of exported noun stems from the database converted to `lexd` source code.

An example of real `csv` and resulted `lexd` fragments are shown in Code block 20. I decided to keep meanings from the source as `lexd` comments. Later it makes it easier to debug lexicons manually, this should not influence the compilation process as comments are ignored. The `form_lexd.py` script is located in `scripts/lexicons/` directory. Besides formatting it also preprocesses the Shughni stems: removes stresses, unifies Cyrillic script and filters out affix entries (source dictionaries have entries for affixes alongside with regular stems). For convenience the script also sorts stems alphabetically.

The `form_lexd.py` script does not read command line arguments. The configuration is 'hard-coded' to read all the `.csv` files from `scripts/db_dumps/` directory and save the generated `.lexd` files to `lexd/lexicons/`.

1.6 Transliteration

The transliteration in this work is implemented for Latin and Cyrillic scripts of the Shughni language. It is developed as a pair of separate FSTs: `translit/lat2cyr.hfst` (Latin to Cyrillic transliteration) and `translit/cyr2lat.hfst` (Cyrillic to Latin transliteration). I found it more convenient to compile transducers with `lexd` (opposed to raw 'strings' format). `Lexd` allows splitting characters and special symbols into separate lexicons and writing comments, which makes the source file much more readable. Also, as a side bonus, `lexd` syntax allows repeating (making it infinitely cyclic) the transducer without the need to repeat it via HFST tools in the `Makefile`.

The transliterator only works for wordforms (e.g. '`daryoyen`'='river.N-PL'). It does not work for glossed strings (e.g. '`daryo<n>><pl>`'). The reason for this is that in case of Latin to Cyrillic transliteration the FST would also transliterate grammatical tags, outputting '`дарйо<н>><пл>`' given '`daryo<n>><pl>`' as input. This could be solved by listing all the possible grammatical tags as multichar symbols in the transliterator's lexicon, but this seems unnecessarily complicated to me for transliteration purposes.

As shown in Code block 21, transliteration for wordforms works for both plain text wordforms and morpheme-separated wordforms (the difference was explained in Section 1.2: *Morpheme borders*). This ensures that the transliteration FST can be successfuly composed with any version of wordform side of FSTs.

```
$ cat lat.txt | hfst-lookup -q translit/lat2cyr.hfst
na>čis>en   на>чис>ен   0.000000
na-čis-en   на-чис-ен   0.000000
načisen     начисен     0.000000
$ cat cyr.txt | hfst-lookup -q translit/cyr2lat.hfst
на>чис>ен   na>čis>en   0.000000
на-чис-ен   na-čis-en   0.000000
начисен     načisen     0.000000
```

Code 21: An example of transliteration FST work. The mentioned word '`na>čis>en`' can be glossed as *NEG-look.V.PRS=3PL*

Two separate `lexd` files were used to generate transliterators of `lat2cyr` and `cyr2lat` directionalities. Earlier in the development, only one direction was defined with `lexd`, while the other was compiled by inverting the default direction. This brought some complications, as Cyrillic and Latin scripts are not strictly standardized. Some letters have different variations, for example the /θ/

representation, which can be ther be representated either by ‘ϑ’(Unicode ‘U+03D1’) or ‘θ’(Unicode ‘U+03B8’) (lower- and upper-cased variations of a symbol, for human eye it looks the same in some fonts, so it gets confused sometimes). The another example is variations of Latin /ð/, which can be represented by both ‘δ’(Unicode ‘U+03B4’) and ‘ð’(Unicode ‘U+00F0’) symbols, but only by ‘δ’ for Cyrillic. The issue with having a FST definition for only a single direction is that one of the directions will always be overgenerated for all the letter variants. A real example is shown in Code block 22. The `lat2cyr` direction standardizes different /ð/ variations, while the inverted `cyr2lat` FST generates all 4 possible combinations of two /ð/ instances.

```
$ lexd translit/lat2cyr.lexd | hfst-txt2fst -o translit/lat2cyr.hfst
$ cat lat.txt | hfst-lookup -q translit/lat2cyr.hfst
δāδān   δāδāн   0.000000
ðāðān   δāδāн   0.000000
$ hfst-invert translit/lat2cyr.hfst -o translit/cyr2lat.hfst
$ echo "δāδāн" | hfst-lookup -q translit/cyr2lat.hfst
δāδāн   ðāðān   0.000000
δāδāн   ðāδān   0.000000
δāδāн   δāðān   0.000000
δāδāн   δāδān   0.000000
```

Code 22: A transliterator FST pair example, where one of the directions is compiled via the inversion of other.

It was solved by two separate `lexd` definitions for each of the directions. This approach made transliteration more flexible, as it was possible to control each direction separately. A real example of the final transliterator FST version is shown in Code block 23. Transliterator FSTs are later attached to the FST generators' output and analyzers' input. This ensures that both transliteration directions minimize variability, which prevents unnecessary overgeneration on the transliteration stage.

```
$ lexd translit/lat2cyr.lexd | hfst-txt2fst -o translit/lat2cyr.hfst
$ cat lat.txt | hfst-lookup -q translit/lat2cyr.hfst
δāδān   δāδāн   0.000000
ðāðān   δāδāн   0.000000
$ lexd translit/cyr2lat.lexd | hfst-txt2fst -o translit/cyr2lat.hfst
$ echo "δāδāн" | hfst-lookup -q translit/cyr2lat.hfst
δāδāн   δāδān   0.000000
```

Code 23: A transliterator FST pair example, where bith directions are compiled from individual `lexd` definitions.

1.7 Lemma translation

The Shughni stem translator is a separate FST which translates Shughni stems to Russian lemmas. It is supposed to be applied to the glossed side of transducers as shown in Code block 24. Shughni has some cases of compound words which consist of multiple stems, e.g. *‘дароз-зӯɣ’* (=‘long-sleeved’) (Parker, 2023, p. 173). This was taken into account, so the translator FST translates any amount of stems in a single word. This FST is blind to morphology rules and will translate anything that contains a set of existing stems and tags, even if the combination is nonsensical. But after composition with a morphology model FST ungrammatical and nonsensical strings are left outside the of resulted FST's paradigm.

```
$ cat glossed.txt | hfst-lookup -q translate/sgh2rulem.hfst
зӯў<n>                     рукав<n>                     0.000000
дароз<adj>>зӯў<n>          длинный<adj>>рукав<n>        0.000000
дароз<adj>><dim>           длинный<adj>><dim>           0.000000
$ cat nonsense.txt | hfst-lookup -q translate/sgh2rulem.hfst
<adj><v>>зӯў<n><pl><sg>   <adj><v>>рукав<n><pl><sg>    0.000000
<dim><pl><sg><1sg><2sg>    <dim><pl><sg><1sg><2sg>      0.000000
$ hfst-compose translate/rulem2sgh.hfst sgh_gen_stem_word_cyr.hfst
  -o sgh_gen_rulem_word_cyr.hfst
$ cat nonsense.txt | hfst-lookup -q sgh_gen_rulem_word_cyr.hfst
<adj><v>>зӯў<n><pl><sg>   <adj><v>>зӯў<n><pl><sg>+?   inf
<dim><pl><sg><1sg><2sg>    <dim><pl><sg><1sg><2sg>+?    inf
```

Code 24: An example of Shughni stem translator work. '`+?`' at the end of second column's string and '`inf`' in the third column means that FST did not find a valid path for the current word.
'`зӯў`'=*a sleeve*; '`дароз`'=*long*
Note: output is edited, contextually insignificant lines are removed to keep Code section small.

The `lexd` source code pattern is shown in Code block 25. `RuLemmasBase` pattern contains lexicons grouped by POS. This ensures that homographs (wordforms with same graphical form but different meanings) with different POS will not share contexts. An example of a homograph in Shughni is *'ди'*, which can stand for a verb (*'hit.V.PRS/IMP'*), a demonstrative (*'D2.M.SG'*) or a conjunction (=*'when.CONJ'*). An example of FST with and without POS-grouping is shown in Code block 26. A wordform '`ди`', when fed into `analyze_pos_ignored.hfst`, where homographs are not taken into account, returns ungrammatical glosses like *'когда<v>'*('when<v>') and *'этот<v>'*('this_one<v>'), which is an unwanted result. But when fed into `analyze_pos_fixed.hfst`, where translator has POS tags are glued to stems, the result contains only verbal Russian lemma '*бить<v>*'('hit<v>').

```
PATTERNS
(RuLemmasBase|RuLemmasTags)+

PATTERN RuLemmasBase
RuLemmasAdj [<adj>]
RuLemmasAdv [<adv>]
...
RuLemmasPost [<post>]
RuLemmasV [<v>]
```

Code 25: A fragment of translator FST's `lexd` source code. The fragment contains only pattern rules.

```
$ hfst-compose translate/rulem2sgh_no_pos.hfst sgh_base_stem.hfst |
  hfst-invert -o analyze_pos_ignored.hfst
$ echo "мā>ди" | hfst-lookup -q analyze_pos_ignored.hfst
ди      бить<v><imp>                0.000000
ди      когда<v><imp>               0.000000
ди      этот<v><imp>                0.000000
$ hfst-compose translate/rulem2sgh.hfst sgh_base_stem.hfst |
  hfst-invert -o analyze_pos_fixed.hfst
$ echo "мā>ди" | hfst-lookup -q analyze_pos_fixed.hfst
ди      бить<v><imp>            0.000000
```

Code 26: A comparison of FST analyzers. The first one called `analyze_pos_ignored.hfst` treats all homographs independently of their part of speech, which results in incorrect output. The second one (`analyze_pos_fixed.hfst`) returns only verb Russian lemmas.
Note: output is edited: contextually insignificant lines are removed to keep Code section small.

Translation lexicon generation

Translation lexicons (`RuLemmasAdj`, `RuLemmasConj`, etc.) are compiled using a `Python` script `scripts/ru_lemmas/form_lexd.py`. This FST module (`sgh2rulem.hfst`) does not share source code with the main morphology FST, so its lexicons are stored in a separate directory `translate/lexd/`. For convenience every POS lexicon is stored in a separate file under this directory. Compilation process is done by the same principle as for the main FST (see Code block 8).

The biggest challenge of this process was the extraction of Russian lemmas from the database dictionary (provided by Makarov et al. (2022)). Gloss lemmas are preferred to be concise. They usually consist of a single word like (e.g. *swim*) or maximum of 2-3 words (e.g. *swim_deep*) if lemma language can not describe semantics in a single word, and it is important to mention this semantic nuance. Dictionary entries in the database are parsed from real dictionaries, that were written by hand and therefore do not have a strict format. This fact makes it challenging to consistently extract perfect lemmas, some examples are presented in Table 2. In addition, as shown by ‘чи’ entry, sometimes a real lemma (in this case ‘кто’) can be hidden in the middle of a text.

Fortunately, most of the lemmas can be extracted automatically without a problem. Russian lemma size minimal statistics are provided in Table 3. Most of the Russian lemmas consist of a couple of words: 76.4% consist of 1 word; 89.1% of ≤ 2 words; 94.6% of ≤ 3 words; 96.8% of ≤ 4 words. Nevertheless, there are still many long lemmas similar to the ones shown in the Tables 2 and 3. Considering that a single Shughni stem often has multiple Russian lemmas, the probability of getting at least one long lemma rises.

Word	Dictionary entry	Extracted lemma (by script)
цирӯвд	горевать, тосковать, печалиться; скорбеть	горевать
х̆ӣвдоw	сбивать палкой орехи, бить по орешнику палкой	сбивать_палкой_орехи
цу̊н	подчинительный союз: сколько ни, как ни	сколько_ни
чи	косвенная форма к прямой форме вопросительного местоимения ЧĀЙ кто (см.), употребляемая в различных косвенных позициях	косвенная_форма_к_прямой_форме_ _вопросительного_местоимения_чй_ _кто

Table 2: Examples of extracted Russian lemmas from dictionary entries.

Type	Mean	Median	Max
Length (characters)	10.536	8	105 ‘указывает_на_косвенное_дополнение_со_ _значением_адресата_действия_при_ряде_ _глаголов_и_глагольных_сочетаний’
Length (words)	1.480	1	16 ‘косв_форма_мн_ч_указ_мест_дальн_ст_к_ _прямой_форме_w_них_и_т_п’

Table 3: Examples of extracted Russian lemmas from dictionary entries.

1.8 Testing

In context of this work, testing is creating a list of pairs of wordforms and their glossed strings and comparing it to the morphology model's output. These pairs are taken from reliable sources such as field work data or examples from other papers. Testing is useful mostly for the developer for debugging purposes: it lets the developer know if anything stopped working as intended after a change in the source code.

Testing is integrated in the project's `Makefile`. In order to run all existing tests '`make test`' bash command can be used. It runs `scripts/testing/runtests.py` script, which reads pairs of wordforms and glossed strings from all `.csv` files under the `scripts/testing/tests/` directory. An example of a `.csv` with test cases is shown in Code block 27. The script allows testing not only morphology, but any other FSTs too. Currently, this script tests morphology and transliteration. Morphology test cases mainly come from Parker (2023) and unpublished materials of HSE expeditions to Tajikistan. Transliteration tests come from the dictionary database provided by Makarov et al. (2022) project. Every database's dictionary entry has Cyrillic and Latin word versions, which were exported and formatted to match my testing `.csv` format. The only downside of the transliteration test cases is that most of Latin words most likely were generated automatically from Cyrillic words. Therefore, running transliteration tests ensures that my transliteration FST matches the transliteration tool developed by Makarov et al. (2022). Unfortunately I do not possess other reliable transliteration test data.

A fragment of `scripts/testing/tests/verb.csv`

```
analysis,form,mustpass,hfst,source,page,description
вāр<v><prs>><1pl>,вāр>āм,pass,sgh_gen_stem_morph_cyr,[Parker 2023],258,…
тойд<v><pst><f>,тойд,pass,sgh_gen_stem_morph_cyr,[Parker 2023],113,…
вирӯд<v><pst>,вирӯд,pass,sgh_gen_stem_morph_cyr,[Parker 2023],115,…
находить<v><pst>,virūd,pass,sgh_gen_rulem_morph_lat,[Parker 2023],115,…
```

A fragment of `scripts/testing/tests/translit.csv`

```
input,output,mustpass,hfst,source,page,description
агāн,agān,pass,translit/cyr2lat,https://pamiri.online/,,
агāнт,agānt,pass,translit/cyr2lat,https://pamiri.online/,,
агāнтow,agāntow,pass,translit/cyr2lat,https://pamiri.online/,,
агāнч,agānč,pass,translit/cyr2lat,https://pamiri.online/,,
```

Code 27: An example of a file with test cases for verbs. First two columns contain reference input and output pair. `hfst` column specifies which FST's format variation must be used for current row. `mustpass` column allows to ignore some cases without the need to remove them from the file. Other columns are ignored by the script, their purpose is to keep track of test case sources.

A **test case** in current context is a single row of a `.csv` file (e.g. Code block 27). For every test case the `scripts/testing/runtests.py` script does the following: (1) feeds the string from the first column to the FST specified in the fourth column, (2) marks the test case as passed if the string from the second column matches **any** string in the FSTs output, (3) logs failed cases. Finally, after all test cases were processed, the script prints general statistics. Script's output example is shown in Code block 28.

The testing output statistics are not considered as a metric of morphology model's quality. It is

```
$ python3 scripts/testing/runtests.py
Loaded 27300 test cases from 1 files
Fail [translit.csv:translit/cyr2lat] анę:anę (fst output: анę+?)
... 44 more Fail lines ...
Fail [translit.csv:translit/cyr2lat] лǫ:lǫ (fst output: лǫ+?)
Loaded 61 test cases from 7 files
Fail [verb.csv:sgh_gen_stem_morph_cyr] вуδч<v><prs>:вуδч (fst output: вуδч
<v><prs>+?)
Transliteration: 27254 (99.83%) passed; 46 failed; 27300 total
Morphology: 60 (98.36%) passed; 1 failed; 61 total
Testing done in 1.415sec
```

Code 28: Example of testing script's output with some failed test cases.

only utilized as a development tool: to make sure that FST was compiled as intended.

1.9 Metrics

1.9.1 Quantitative metrics

Coverage metric is defined as relation of the amount tokens that were given any glossed output by FST ($N_{recognized}$) to the total amount of tokens (N_{total}) given to the FST:

$$Coverage = \frac{N_{recognized}}{N_{total}}$$

$N_{recognized}$ does not take into account if the given glosses are correct. It only shows the fraction of words that exist in the FST model's paradigm.

The evaluation is done via a `Python` script `scripts/coverage/eval.py`. It reads Shughni plain text from `stdin`, which must be cleared of all punctuation. Then the script tokenizes it and feeds every token to the FST. Input texts come both in Cyrillic and Latin scripts, so the script detects writing system and calls corresponding FST variation (`sgh_analyze_stem_word_cyr.hfst` or `sgh_analyze_stem_word_lat.hfst`). The script also calculates 5 most frequent unrecognized words and most frequent unrecognized morphemes (it considers any hyphen-separated word fragments as morphemes). An example of its work is shown in the Section **??**.

Qualitative metrics

There are four qualitative metrics: *Precision*, *Recall*, *F-Score* and *Accuracy(any)*. The first three metrics are evaluated conventionally:

$$Precision = \frac{TP}{TP+FP};\ Recall = \frac{TP}{TP+FN};\ FScore = \frac{2 * Precision * Recall}{Precision + Recall}$$

Where `TP` = '*True positive*', `FP` = '*False positive*' and `FN` = '*False negative*'. The principle of these values' evaluation by `scripts/metrics/eval.py` is the following:

1. The script loads the Gold Standard's pairs of wordforms and glossed strings

2. For each unique wordform$_i$:

   (a) all possible Gold Standard's glossed strings are gathered for wordform$_i$ ($= G_i$)

   (b) wordform$_i$ is fed into a FST analyzer and all possible predicted glossed strings are gathered ($= P_i$)

   (c) the following is calculated:
   $TP_i = |G_i \cap P_i|$ (amount of elements in intersection)
   $FN_i = |G_i \setminus P_i|$ (amount of elements in G_i that are not in P_i)
   $FP_i = |P_i \setminus G_i|$ (amount of elements in P_i that are not in G_i)

3. Total TP, FN and FP are calculated from the sum of respective TP_i, FN_i and FP_i

Transducer file name	**Input example**	**Output example**

Table 4: A full list of available HFST transducers