Thesis submitted for the degree of Bachelor of Arts

Using finite state transducers for multilingual rule-based phonetic transcription

Author:
Thora Daneyko
Matrikelnr.: 3822667

Supervisor:
Prof. Gerhard Jäger

Eberhard Karls Universität Tübingen
Philosphische Fakultät
Seminar für Sprachwissenschaft

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt, alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe und dass die Arbeit weder vollständig noch in wesentlichen Teilen Gegenstand eines anderen Prüfungsverfahrens gewesen ist und dass die Arbeit weder vollständig noch in wesentlichen Teilen bereits veröffentlicht wurde sowie dass das in Dateiform eingereichte Exemplar mit den eingereichten gebundenen Exemplaren übereinstimmt.

Tübingen, den _____ _____

Thora Daneyko

Abstract

EVOLAEMP is a current research project on comparative historical linguistics which is compiling a large lexical database called NorthEuraLex covering most languages of Northern Eurasia. To be able to compare the lexemes of these languages with their different writing systems and orthographies, NorthEuraLex aims to provide phonetic transcription in IPA for all entries. Because expert transcriptions are hard to obtain for smaller languages, automatic rule-based transcriptors have been developed.

However, the current system is quite inefficient, resulting in bad performance especially on longer words. Thus, I develop and test a revision of this implementation using finite state transducers (FST), which have previously proven to be very suitable for modeling phonological processes. The new implementation should perform faster and be able to operate on the system-independent rule sets already developed for 107 languages.

The FST interface used in my system is the Helsinki Finite State Toolkit (HFST). It provides the possibility to directly construct FSTs from regular expressions. Hence, my transcriptors first translate the EVOLAEMP rule sets to regular expressions. The resulting transducer can then be accessed to convert the lexical data to IPA.

With this method, the output of the current system can almost entirely be replicated. However, the FST is not able to completely model the behavior of the current transcriptors, so a few changes to the rule sets may be necessary. Still, the new implementation operates significantly faster on longer paragraphs and is thus well suited for transcribing complete texts, but performs slightly worse on lists of single words, such as the entries in NorthEuraLex.

Thus, while being a good approach, it still needs some improvements before replacing the current EVOLAEMP transcriptors.

Contents

1 Introduction

In comparative historical linguistics, large-scale lexical databases have become invaluable for establishing cognacy relations and examining language evolution. However, comparing entries from languages with different writing systems and orthographies is complicated. Thus, it is necessary to have a uniform phonetic transcription for your lexical data.

The existing databases treat this problem differently. While the Austronesian Basic Vocabulary Database (ABVD) does not provide any phonetic description for its data (Greenhill, Blust, and Gray 2008)[1], the creators of the ASJP database have developed their own transcription scheme (Brown et al. 2008), which, however, only distinguishes 41 phonemes. This is not enough for properly reflecting the sound inventory of many languages. The Indo-European Lexical Cognacy Database (IELex) provides transcriptions in the more detailed IPA, but only for few languages (*Indo-European Lexical Cognacy Database* 2016).

The fragmentary and inconsistent state of phonetic transcription among the larger lexical databases stems from the lack of data for many of the smaller languages. Lexicons rarely provide pronunciation information for individual entries and if they do, they often make use of respellings in the other language's orthography instead of precise phonetic transcription. Finding a native speaker to provide information on pronunciation is time-consuming, and often impossible for languages on the brink of extinction. However, grammars usually provide detailed information on phonology, including phonological processes such as assimilation. While a human unfamiliar with the language might not gain much from reading these descriptions, programming computers to apply these rules and derive a phonetic transcription could prove beneficial. It should also facilitate corrections, since single rules can easily be added or modified to improve the output, while in case of manual transcriptions, the complete transcribed data would have to be examined. In fact, automatic rule-based transcription systems have already been successfully developed for text-to-speech applications (e.g. Braga and Coelho 2006; Toma and Munteanu 2009).

EVOLAEMP is a current research project on cultural language evolution at the University of Tübingen which has implemented such a transcription system for its own lexical database NorthEuraLex (*EVOLAEMP. Language Evolution: The Empirical Turn* 2016). The project aims at compiling word lists covering 1,016 concepts for the languages of Northern Eurasia as a basis for statistical research. Currently, near-complete data for 104 languages has been gathered, and data for more languages is still being collected (Dellert et al. 2016). Furthermore, automatic rule-based transcriptors for 107 of these languages have been created. Since the rewrite rules for these transcriptors are stored in plain text files, they can easily be modified and integrated into other programs.

In this thesis, I present a revision of EVOLAEMP's transcription system based on finite state transducers (FST). My implementation aims to exceed the old

[1] It does provide an IPA-like transcription for languages without an official orthography, but inconsistently. The glottal stop, for instance, is sometimes transcribed with IPA's ʔ and sometimes with an apostrophe (instead of the ʻokina), and long vowels may be marked by double letters, a macron and only rarely by the IPA sign ː. Compare e.g. these translations for 'belly': Hawaiian *ʾōpū*, Tahitian *ʾoopuu*, Rurutuan *ʔoopuu*, Musao *goːwa*.

transliterators in performance and make it applicable for purposes other than processing lists of short words, e.g. phonetic transcription of whole texts. Additionally, the FST-based transcriptors can more easily be integrated into other systems, since they employ fully functional transducers convertible into various well-known formats which may also be distributed independently. Ideally, the new implementation should generate the exact same results as the old system, so that the existing 107 rule sets do not have to be altered.

In section 2, I introduce the previous EVOLAEMP transliterator system and explain the format of its rule files, which are the basis for my FST implementation. In section 3, I then provide an overview over finite state transducers and the most important operations for reproducing the output of the previous system. The FST implementation I am using for my system is the Helsinki Finite State Toolkit (HFST), whose features I present in section 4. A special focus lies on the format of its regular expressions, which are crucial in converting the EVOLAEMP phonetic rules into finite state transducers. Section 5 then introduces and explains my implementation of an FST-based transcription system, pulling together the insights gained from the previous sections. The results of a runtime test performed by both the old and my FST system are presented in section 6 to show the advantages and disadvantages of the new implementation.

The complete code of the transliterator system is provided in the appendix, with the specific transliterator for Aleut and the corresponding rule files as an example.

2 EVOLAEMP Transliterators

EVOLAEMP's previous transliterator system was a quick and simple Java implementation by Johannes Dellert, which I will from now on refer to as the *naive transliterators*. Within this system, transcription rules are stored in one or more plain text files for each language and accessed by the corresponding transliterator class to convert the user input. To facilitate the creation of new transliterators, the files for the individual languages do not directly transliterate to IPA. Instead, the data is first mapped to X-SAMPA, an ASCII representation of IPA that can easily be input with any roman character keyboard (Wells 1995). The X-SAMPA transcription can then be unambiguously converted to IPA in a final transliteration step. Additionally, all input words are surrounded by word boundary marks `#` and converted to lower case before the actual transliteration process begins.

The naive implementation is working well for its purpose, i.e. transcribing word lists of about 1,000 items to IPA. However, as will be explained in this section and demonstrated in section 6, its short development time comes at the cost of bad performance, especially on larger data and actual texts. Hence, it was to be eventually replaced by a more efficient system, which I am presenting in this thesis.

This section gives an introduction to the current EVOLAEMP transliterator system. First, I introduce the format of the rule files, which is adopted by

the new FST implementation as well. I then quickly explain how the naive transliterator program works and where its weaknesses come from. Finally, I list the properties of the previous system resulting from the naive implementation and the rule format that have to be emulated by the new implementation to guarantee backward compatibility.

2.1 Rule files

The rule files are the core of the transliterator system, since they contain the language specific mappings from original orthography to phonetic transcription. The format of the rules is simple, but powerful, and can be used not only for grapheme-to-phoneme conversions, but also to perform phonological processes not represented in the script, such as assimilation. As an example of how the rules can be used to model the phonology of a language, I will construct part of the files for an Aleut-to-IPA transliterator.

While historically, Aleut has been written in Cyrillic script, the Aleut vocabulary in the EVOLAEMP database uses the Roman orthography developed in 1972 (Bergsland 1994, pp. xvi ff.). Many letters, such as `p`, `n` or `i`, equal their X-SAMPA (and IPA) realizations, so no rules are needed to transcribe them. The remaining characters and character sequences can be expressed in 20 rules and are stored in a file called `ale2xsampa`:

aa a:	hm m_0	oo o:
ch t)S	hn n_0	q q_h
d D	hng N_0	r r\
ĝ R	hw W	uu u:
g G	hy C	x̂ X
hd T	ii i:	y j
hl K	ng N	'

There is one rule per line, and the input and output side of each rule are separated by a single tab stop. The output side of a rule can also be empty to account for deletion of characters or character sequences. Aleut, for example, uses the apostrophe `'` to divide digraphs like `ng` into the separately pronounced letters `n` and `g`. Since the apostrophe does not carry any phonological information itself, we want to get rid of it during the transcription process, so we have added a rule `'` $\rightarrow \varnothing$.

Comments, i.e. lines of text that will not be interpreted by the transliterator program, can be added by prefixing two forward slashes (`//`). In our Aleut file, we might want to record the source of the rules we created, so we can add the following line:

```
// Source: Knut Bergsland - Aleut Dictionary (1994)
```

2.1.1 Context groups

Naturally, simple grapheme-to-phoneme mappings are usually not sufficient to properly render the pronunciation of a language. Unwritten sounds and processes such as assimilation or epenthesis need to be captured to arrive at a satisfying transcription. Our example language Aleut, for instance, features extensive vowel coloring in the vicinity of certain consonants. It would be tedious to write down all possible combinations of vowels and consonants in this case, so we can define sound groups within the rule file, allowing us to capture similar phonological processes in a single rule. In analogy to the sound contexts in actual phonological rules, we call these sound groups *context groups*.

This is how we could encode vowel retraction next to uvular consonants in Aleut (Bergsland 1994) in a file we call `ale-vowels`:

```
#def    uvular    [q_h X R]

[uvular]i    [.]e
[uvular]a    [.]A
[uvular]u    [.]o
i[uvular]    e[.]
a[uvular]    A[.]
u[uvular]    o[.]
i:[uvular]    e:[.]
a:[uvular]    A:[.]
u:[uvular]    o:[.]
```

The context group covering Aleut's uvular consonants is created using the keyword `#def`, the name of the group (`uvular`) and the glyphs or sounds it contains (`q_h`, `R`, `X`), all three separated by tab stops. The members of the group are enclosed in square brackets and separated by whitespaces. They do not need to be single characters, but can be of arbitrary length. Context groups are immutable, i.e. they can only be inserted into a rule to provide context for other characters. We cannot, for instance, define a context group `default-vowel` containing `i`, `a` and `u`, and map its members to those of another group `retracted-vowel` containing `e`, `A` and `o`.

To use our uvular group inside a rule, we place its name in square brackets (i.e. `[uvular]`) on the input side. This keyword will then be interpreted as 'any of the character strings contained inside this group'. On the output side, each context group introduced on the input side must be matched by the sequence `[.]`. Hence, contexts cannot swap positions, since the transliterator will always interpret the `[.]`s in the order the groups appeared on the input side.

2.2 Naive transliterator

After having constructed some rule files, we will now look at how the transliterator program works.

A transliterator for a language, such as Aleut, actually consists of multiple

```
int start = 0
while start < input.length
    int oldStart = start
    for rule in rules
        boolean match = false
        int end = line.length
        while end > start
            if rule.left matches line.substring(start, end)
                output += rule.right
                start = end
                match = true
                break
            else
                end--
        if match
            break
    if start == oldStart
        output += line.charAt(start)
        start++
```

Figure 1: Pseudocode of the conversion of a line of input by a naive transliterator covering a single rule file.

transliterators, one for each rule file. The main transliterator merely passes the user input through these individual transliterators in the specified order.

Upon construction, each of these transliterators reads its assigned rule file and stores all rules in a list. Then, it loops through this list, looking for a matching rule. For each rule, it considers the whole input string and then prefixes of decreasing length until one of them either matches the left side of the rule or the current substring is empty. In the latter case, it proceeds to the next rule. If no applicable rule has been found, the initial character is moved to the output string unchanged. After a rule has been successfully applied or the initial character has been cut off, it skips back to the first rule and repeats the whole procedure on the remaining substring until the input is converted completely. Figure 1 illustrates this procedure.

To illustrate this, let us look at how the transliterator would proceed on the Aleut word *chax̂* 'hand' using the simple grapheme-to-phoneme file `ale2xsampa` we developed in section 2.1. The first rule of the file is `aa` → `a:`. The transliterator will now unsuccessfully attempt to match `chax̂`, `cha`, `ch` and `c` to `aa`. Since the application of this rule failed, it will proceed to the next one, `ch` → `t)S`. `chax̂` and `cha` do not match `ch` either, but the next prefix, `ch`, does, so the program transliterates this bit and repeats the procedure with the remaining sequence `ax̂`. Since neither `ax̂` nor `a` are matched by any of the rules, the transliterator ends up moving `a` to the output string unaltered, which now reads `t)Sa`. Finally, all rules are run over `x̂`, which is ultimately matched by rule 19, `x̂` → `X`. Hence, the final output is `t)SaX`.

Needless to say, this procedure is highly inefficient, as its operating time grows significantly the longer the rule files and input strings are. For transliterating

the single words in the EVOLAEMP database, this is negligible, but not for other purposes such as transliterating whole texts, which is why a more efficient solution was needed.

2.3 Properties of the naive procedure

The interplay between the rule format and the proceeding of the naive transliterators entails several properties that need to be transferred to the FST implementation in order to be able to keep the existing rule files. Despite appearing to be similar at first glance, the EVOLAEMP transliteration also differ considerably from the application of classical phonological rules as described in Chomsky and Halle 1968.

The transliteration proceeds **left-to-right**. Rather than focusing on the rules and applying them one after the other wherever they match in the string like phonological rules, the system focuses on the input string, processing it from left to right and applying the rules as it encounters matching character sequences.

The rules are applied **in order**. Rules that are further up in the list are tried before those below them and the first rule that matches an arbitrarily long prefix of the input string is applied. Because of this, the ordering of the rules requires special attention when compiling a rule file. Consider the following two lines from `ale2xsampa`:

```
hn    n_0
hng    N_0
```

If we were to transliterate *kihngux̂* 'grief' using the rules in this order, we would receive the wrong result `kin_0GuX`, because when the string has been cut down to `hngux̂`, the first matching rule is `hn` → `n_0`. Actually, the rule `hng` → `N_0` can never be applied in this constellation, because its predecessor will always match before it. Hence, to arrive at the correct transliteration `kiN_0uX`, we need to swap the two rules:

```
hng    N_0
hn    n_0
```

What we actually want to have is a **greedy** transliterator, i.e. a transliterator that converts the longest prefix possible. So far, greediness has been implemented manually via the rule order, but it might be beneficial to keep the underlying concept in mind when designing the FST system.

Due to the left-to-right application, already converted material is **blocked** from further transliteration within the same rule file. When a `q` has been converted to `q_h`, the `q` inside that output cannot be converted to another `q_h` by the same transliterator. Obviously, this would not only be counter-intuitive to what we wanted to encode with that rule, it would also lead to an infinite loop. Similarly, a (fictitious) rule `h` → `x` in `ale2xsampa` could not be applied to that `q_h` either.

The application behavior resulting from this blocking mechanism is quite different from that of phonological rules, which always operate on the output of the

previous rules: EVOLAEMP rules may only apply to input not yet matched by another rule within the current file. Hence, phonological rule relationships such as feeding and bleeding, i.e. one rule explicitly generating or destroying input for another (Kiparsky 1968), can only take place between EVOLAEMP rule files. Another important difference is that while in phonological rules, contexts may be rematched for an arbitrary number of times by the same rule, the EVOLAEMP context groups are blocked by the transliterator as well. Hence, the vowel retraction file `ale-vowels` developed in section 2.1.1 does also not work as intended.

Consider the word *aquliχ* 'gate', already transliterated to `aq_huliX` by the preceding file. The vowel retraction transliterator would correctly match the prefix `aq_h` to the rule `a[uvular]` → `A[.]`, converting it to `Aq_h`. This would, however, reduce the remaining input to `uliX`, rendering an application of the rule `[uvular]u` → `[.]o` to the sequence `q_hu` impossible. Thus, the rules for vowel retraction must in fact be distributed over two files which we call `ale-vowels1` and `ale-vowels2`:

```
#def    uvular    [q_h X R]

[uvular]i    [.]e
[uvular]a    [.]A
[uvular]u    [.]o
```

```
#def    uvular    [q_h X R]

i[uvular]    e[.]
a[uvular]    A[.]
u[uvular]    o[.]
i:[uvular]    e:[.]
a:[uvular]    A:[.]
u:[uvular]    o:[.]
```

Now we can derive the correct transliteration in two steps: `aq_huliX` → `Aq_huleX` → `Aq_holeX`.

Hence, in order to keep the current rule files while achieving identical transliteration results, the new implementation must transliterate strings from left to right, apply the rules either greedily or in the order they appear in the file, and block any processed material from further modifications by the same rule file, even the members of the immutable context groups.

3 Finite State Transducer

A finite state transducer (FST) is a finite state machine with two tapes: An input tape and an output tape. While finite state automata simply read and accept or reject input, finite state transducers additionally map it to an output string (Kaplan and Kay 1994). Thus, a finite state *automaton* encodes a regular *language*, i.e. a set of strings, and a finite state *transducer* encodes a regular *relation*, i.e. a set of ordered pairs of strings (Beesley and Karttunen 2003).

The common notation for FST transitions is of the form `a:b`, where `a` is the input character and `b` is the output character. Similar to how epsilon transitions connect two states of an automaton without reading a character of the input

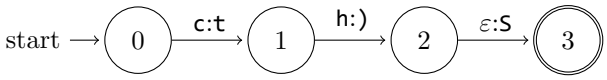

Figure 2: Finite state transducer with input `ch` and output `t)S`.

string, the empty string may appear on both the input and output tape of a transducer, accounting for insertion or deletion of characters. Figure 2 shows an example of a simple transducer implementing the rule `ch` → `t)S` from our `ale2xsampa` file.

Finite state transducer have proven to be very suitable for modeling phonological rules (Kaplan and Kay 1994). Thus, they should also be able to represent the EVOLAEMP transliteration rules quite well. In fact, FST systems have already been successfully implemented for grapheme-to-phoneme conversion (Bouma 2000). In Caseiro, Trancoso, and Oliveira 2002, a rule-based FST system for European Portuguese was even reported to perform better than a machine learning approach.

3.1 Transducer operations

We have seen that simple rewrite rules as in the EVOLAEMP rule format can be expressed as finite state transducers. But creating several small rule transducers is not going to solve the performance problems of the naive transliterator. Ideally we would like to have one large transducer that applies all rules correctly at the same time. Fortunately, finite state transducers can be combined in various ways. In this section I introduce the most useful ones for my implementation.

Concatenation: Just like finite state automata, two FSTs T_1 and T_2 can be concatenated to form a transducer $T_1 \cdot T_2$. This means that the final states of T_1 merge with the start state of T_2 and lose their accepting quality. An example is the FST in Figure 2, which is a concatenation of the single transducers `c:t`, `h:)` and ε`:S`.

Union/Disjunction: Two FSTs T_1 and T_2 may also be disjointed to form a transducer $T_1 \cup T_2$. In this case the start states of T_1 and T_2 are merged. The resulting FST accepts (and converts) the relations encoded by either original transducer. This operation could prove useful for combining rules within the same file.

Kleene closure: An FST T can be looped to form a transducer T^*. To encode a Kleene closure, an epsilon transition is drawn from all final states of T to the start state and the start state becomes accepting.

Composition: Composition is an operation peculiar to transducers; it cannot be applied to automata. When two FSTs T_1 and T_2 are composed, the output tape of T_1 serves as the input tape for T_2, i.e. the resulting transducer $T_1 \circ T_2$ has the input tape of T_1 and the output tape of T_2. This is exactly what happens between individual rule files.

4 Helsinki Finite State Toolkit

The Helsinki Finite State Toolkit (HFST) is not a finite state implementation itself, but merely provides an interface for other, already existing implementations, such as the Xerox Finite State Toolkit (XFST), the Stuttgart Finite State Toolkit (SFST), OpenFst and foma, with additional tools (Koskenniemi and Yli-Jyrä 2008). It is primarily designed for morphological analysis, but can be used for any finite state applications. The HFST tools can be addressed directly from the command line as well as via a Python and C++ API.

HFST was chosen as the FST compiler for my implementation for three main reasons. First, as an open source project, it still receives regular updates and bug fixes and can thus be expected to run on future systems as well. Second, it provides full Unicode support, which is indispensable for the range of writing systems the transliterators need to cover. Finally, while most of the other larger finite state toolkits have these features as well, they are all contained within HFST. Even more importantly, HFST is able to convert from one format to the other, so that any FST created with HFST can easily be used by one of the other implementations if needed.

4.1 XFST regular expressions

HFST provides a convenient tool, `hfst-regexp2fst`, to construct finite state transducers which converts a regular expression into an FST. The format of these regular expressions is the one described in Beesley and Karttunen 2003 for XFST. In this section I will provide an overview over the symbols and operators relevant for my transliterator system.

Simple relations are of the form `a:b`, where `a` is a single input character and `b` is a single output character. Identity relations can be formulated as either `a:a` or simply `a`. These two expressions are equivalent as long as `a` denotes a single character, but produce different transducers if `a` is a set of more than one character or string: `a:a` is the cross product of all strings contained in `a`, while the true identity relation `a` maps each member of `a` only to itself. Thus, the format `a` should always be the preferred one for identity relations.

To concatenate two or more relations, no special operator character is needed, they can just be typed one after the other (`a:ba:b`). For the sake of readability, one can, however, insert a whitespace between concatenated relations (`a:b a:b`).

The disjunction operator is `|`. The transducer `a:c | b:c` will map either `a` to `b` or `c` to `b`. This operator can also be applied to the input or output tape alone. Thus, the previous example could also be formulated as `[a|b]:c`. The square brackets are used to cancel operator precedence. Without them, the expression `a|b:c` would encode a transducer that either accepts `a` or maps `b` to `c`. We could also construct an FST `c:[a|b]`, which would transduce `c` to both `a` and `b`, yielding ambiguous output.

To compose two transducers, the operator `.o.` may be inserted. Thus, the transducer `a:b .o. b:c` is equivalent to the transducer `a:c`.

The symbols for the Kleene closure are the same as in other regular expressions: The asterisk * denotes 'zero or more of the preceding expression', as in `[a:b]*` or `a*`, while the plus sign `+` denotes 'one or more of the preceding expression', as in `[a:b]+` or `a+`.

Similarly, there are two symbols for the complement operation which have slightly different meanings: The tilde `~` denotes the set of all *strings*, including the empty string, that are not contained in the following expression, as in `~[a|b]` or `~a`. The backslash `\` then denotes the set of all *single characters* that are not contained in the following expression, as in `\[a|b]` or `\a`. Note that the complement operation can only be applied to languages, not to relations.

Additionally, there are some single symbols that bear a special meaning. The empty string is denoted by the number zero `0`. XFST further features a wildcard which stands for any single character, i.e. excluding the empty string, expressed by the question mark `?`. Hence, the identity relation `?` is the relation that simply accepts any single character which is encountered. In case we intend to create a transducer mapping from or to any of the characters having a special meaning or function in XFST, we can escape them using the percentage sign `%`. Thus, `%0:1` is the relation that maps the number zero `0` to a one `1`.

Combining what we have learned so far about XFST regular expressions, the transducer in Figure 2 could be written as `c:t h:) 0:S`. This nicely corresponds to the transitions of the resulting FST, but feels inconvenient, since what we really want to write is something like `ch:t)S`. The XFST framework provides two ways of treating multicharacter strings: They may be surrounded by quotation marks `"`. The string within these quotation marks is then interpreted as an atomic entity rather than as a concatenation of single symbols. Furthermore, several special characters are automatically escaped when occurring between quotation marks. We could thus formulate our rule as `"ch":"t)S"`. The resulting transducer, however, would not be equivalent to the one we get from `c:t h:) 0:S`. In addition, not all special characters, such as the newline `\n` and the tab stop `\t`, are escaped, which is inconvenient since the backslash occurs frequently in X-SAMPA. The second option is thus preferable: Enclosing a string in curly braces `{}` escapes all special characters except for the braces themselves, and it is interpreted as a concatenation of the single characters contained inside that string. Hence, `{ch}:{t)S}` is equivalent to `c:t h:) 0:S`.

Note that for each of the transducer operations discussed in section 3.1, HFST also provides a separate tool (`hfst-concatenate`, `hfst-disjunct`, `hfst-repeat` and `hfst-compose`, respectively), so that not all of the EVOLAEMP rules will have to be written to a single regular expression, but can be distributed over multiple smaller transducers and then merged using the aforementioned tools.

4.1.1 Replace expressions

XFST provides an alternative way to spell out finite state transducers, which is the *replace expression*. These expressions have been modeled after phonological rules and are thus of particular interest to us.

The most basic form of the replace expression is `A -> B`, where `A` is the input language and `B` is the output language. This is equivalent to the regular ex-

pression `[A:B | \A]*`: It maps all instances of `A` in the input to `B`, ignoring any other characters in between.

Two replace expressions `A -> B` and `C -> D` can be applied simultaneously when separating them with double commas, as in `A -> B ,, C -> D`. Any arbitrary number of replace expressions may be combined in this way. To exclude ambiguities, we can encode left-to-right greediness by rewriting the replace operator as `@->`, or left-to-right laziness by rewriting it as `@>`. Thus, a transducer `a @-> c ,, {ab} @-> c` will always prefer the second rule over the first if possible. Similarly, `[a|{ab}] @-> c` will always map the longer sequence `ab` to `c` and not `a` where applicable.

A replace expression can also be restricted by a context specification. The FST `A -> B || X _ Y` will only replace those instances of `A` with `B` that are preceded by `X` and followed by `Y`. The contexts `X` and `Y` must be languages, not relations, and are optional, i.e. not both have to specified. Just as in phonological rules, these contexts are reusable and may serve as contexts again for the same or another rule.

5 Implementation

In the previous sections I have extracted the properties of the naive implementation I want to reproduce in my system and introduced the FST framework and syntax relevant for my program.

This section describes my implementation of an FST-based transliterator system operating on the rule files specified in section 2.1. It has been programmed in Java. Because of this, I decided to interact with HFST via the command line and not the Python or C++ API, so an installation of HFST's command line tools is necessary to run the transliterators. In addition, the system directly addresses `bash` to execute the HFST commands, so it is not compatible with Windows computers, but should run on most Linux and OS X devices.

After demonstrating the usage of the final system at the example of the transliterator for Aleut, part of whose rule files we have developed in section 2.1, and giving an overview over the system's architecture, I go into the details of how it works. First, I explain how the underlying HFST transducer is built. A special focus lies on the design of the XFST regular expressions encoding the EVOLAEMP rule files. Finally, I describe the components responsible for the actual transliteration.

5.1 Demonstration

`AleutToIPATransliterator` can be used in two ways: We can start it in interactive mode, which enables us to spontaneously type words or sentences for the system to transliterate and immediately receive a result. Often, however, we might want to transliterate a whole file of lemmata at once, so `AleutToIPATransliterator` is also able to take a text file as input and write the transliteration to another file.

We will first test its ability interactively, by simply calling the Java class:

```
> java AleutToIPATransliterator
Building FST... (This might take a while.) Done.
AleutToIPATransliterator ready, waiting for input (Ctrl+C to
quit):
>
```

Upon launching, the system detects that this is the first time we use it, since it cannot find the transducer `translit.ol`. Thus, before activating the interactive mode, it starts building a new FST.

We may now enter an Aleut expression for it to transliterate:

```
> Ukuĝaan iχamnakuχ̂[2]
ukoʁɑːn eχɑmnɛkoχ
>
```

After returning the IPA transcription of our input, the system allows us to enter another bit of Aleut:

```
> Baluunaχ̂ liidaχ̂ ayχ̂aasim hnin![3]
bɛlyːnɛχ liːðɑχ ajχɑːsim n̥inˈ
>
```

Since the rule files were designed to transliterate single words and not complete sentences with punctuation, the exclamation mark is interpreted as X-SAMPA, where it encodes the IPA sign ˈ for a tonal downstep. In order to solve this problem, we would need to alter the rule files, but for the task of transliterating lists of words, which `AleutToIPATransliterator` was designed for, these kinds of mistakes are negligible.

We are now going to transliterate some Aleut words stored in a file `aleut-words.txt` and write the output to `aleut-ipa.txt`. In order to leave the interactive mode, we press `Ctrl+C`, and then give the command for transliterating the whole file:

```
> java AleutToIPATransliterator aleut-words.txt aleut-ipa.txt
Transliterating aleut-words.txt... Done.
```

The strings contained in `aleut-words.txt` have been successfully transliterated to IPA and saved in `aleut-ipa.txt`.

Because creating the underlying FST usually takes longer than the actual transliteration (as we will see in section 6), the system will not initiate the building process again upon starting as long as it finds an old `translit.ol` in its working directory. Hence, after altering the rule files, it is necessary to delete `translit.ol` before running the program again to see the changes. Alternatively, we can force the system to rebuild the FST using the keyword `compile`:

```
> java AleutToIPATransliterator compile
Building FST... (This might take a while.) Done.
```

It will then overwrite the existing `translit.ol` with a freshly built one.

[2] 'Pleased to meet you'; taken from `http://www.omniglot.com/language/phrases/aleut.php`

[3] 'My hovercraft is full of eels!'; taken from `http://www.omniglot.com/language/phrases/aleut.php`

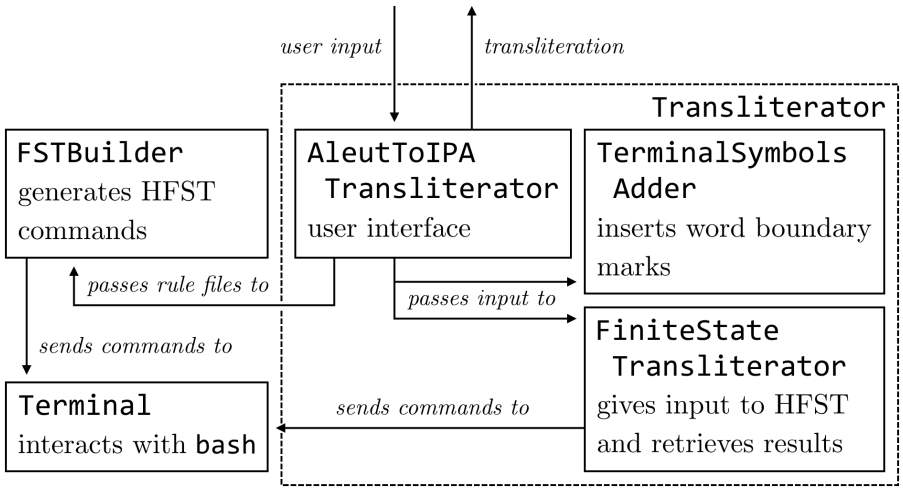

Figure 3: The architecture of the FST transliterator system.

5.2 System overview

Before going into the details of how the system converts the EVOLAEMP rule files into a transducer and processes the user input, we will take a quick look at its general architecture and how its various components interact with each other. Figure 3 shows a diagram of the classes and their relations.

As we have just seen, the only class the user directly interacts with is `AleutToIPATransliterator` (or the transliterator for any other language). This program contains information about the specific rule files of the language it encodes. Apart from that, it does not do much processing itself, but rather serves as a distributor that activates the other classes and provides them with the necessary data.

The core of the system is the `FSTBuilder` which receives the rule files from `AleutToIPATransliterator` and generates a series of HFST console commands to create the corresponding finite state transducer. These commands are then passed on to `Terminal`, a wrapper around Java's `ProcessBuilder` that can start system processes and thus interact with the `bash` where it executes the commands generated by `FSTBuilder`.

User input is run through two other `Transliterators`. First, it is passed to a `TerminalSymbolsAdder` which inserts the word boundary mark `#` that is used in the rule files. The output of this conversion is then handed to the `FiniteStateTransliterator`, the actual transliterator of the system. It creates the console commands to run the user input through the HFST transducer previously constructed by `FSTBuilder` and sends these to the Terminal. The results are then either directly piped into a file or sent back via `AleutToIPATransliterator` to the user, depending on the mode in which the system was started.

5.3 Building the transducer

Before the actual transliteration can take place, a transducer must be built from the EVOLAEMP rule files. As we have just seen, this is the task of `FSTBuilder`. In contrast to the other classes in the system, `FSTBuilder` is static and does not return any value or object.

The only public method of `FSTBuilder`, `build`, takes an array containing the rule files as an argument and processes each file in order. As described in section 2.1, a single non-empty line may contain either a comment, a group definition or a rule, which can be easily recognized by the following attributes:

- starts with `//` → comment
- starts with `#def` → context group definition
- contains `[` → context rule
- else → simple rule

Comments will just be skipped. All new context groups are stored in a map, while rules are transformed into an XFST regular expressions. This conversion will be described in the following subsections, where we will also see why simple and context rules demand a separate treatment. The single transducer resulting from these regular expressions accounts for the complete current file. It is then composed with the previous one, ultimately resulting in a single large transducer which applies all rule files in order.

After all files have been processed, the final transducer is minimized and converted into HFST's optimized lookup format to facilitate its later usage and distribution. Finally, the intermediate transducers and auxiliary files are deleted.

5.3.1 Replace approach

At first glance, the XFST replace expressions seem to be suited perfectly for representing our rules. Multiple rules may be executed simultaneously so that the output of one rule will not be matched by another, ambiguities may be resolved by the left-to-right greedy operator and characters not covered by the rules are simply ignored. The file `ale2xsampa` can thus be expressed by a single replace expression:

```
{aa} @-> {a:} ,, {ch} @-> {t)S} ,, {d} @-> {D} ,, {ĝ} @-> {R}
,, […] ,, {'} @-> 0
```

The transducer built from this expression models the effects of the rule file perfectly. However, first difficulties arise when we try to apply this approach to the context rules in `ale-vowels1` and `ale-vowels2`:

```
{i}[{q_h}|{X}|{R}] @-> {e}[{q_h}|{X}|{R}]
```

Because the `->` operator, like the `:` operator in regular expressions, creates the cross product of its two sides, this rule will generate ambiguous output. Unlike regular expressions, replace expressions have no means of formulating an identity relation. We can dissolve the context group and create three separate transducers from it:

```
{iq_h} @-> {eq_h} ,, {iX} @-> {eX} ,, {iR} @-> {eR}
```

Since this context group has only three members, this expression is still acceptable, but will create very long rule chains for larger context groups, especially for rules with multiple groups. This is not desirable, because `hfst-regexp2fst` becomes considerably slower the longer the expression it needs to parse. A rule with three contexts à 20, 5 and 20 items from EVOLAEMP's Punjabi transliterator, for example, produced 2000 simple rules and was processed for 15 minutes before I canceled it. Thus, another solution is needed.

The built-in context function of XFST replace expressions seem particularly tempting to use here:

```
{i} @-> {e} || _ [{q_h}|{X}|{R}]
```

However, XFST contexts are reusable and can still be matched by another rule. In case of the Aleut files, this is actually beneficial, since it will make the separation of rules over two files redundant. Indeed, many of the phonological processes modeled by the EVOLAEMP context notation, such as intervocalic consonant voicing, face the same problem as the Aleut vowel assimilation files, and could be described more naturally with a reusable context.

Due to EVOLAEMP contexts rather being *variables* for a certain collection of sounds or glyphs than actual phonological *contexts* though, there are examples where they cannot be properly exchanged with XFST contexts. Consider this (simplified) example from the German orthography-to-IPA transliterator accounting for vowel length:

```
[vowel][cons][cons]    [.][.][.]
[vowel]    [.]:
```

Generally, German vowels are short when followed by two or more consonant glyphs, and long in all other cases, i.e. when followed by another vowel, a single consonant or the end of the word. Hence, the 'short vowel case' is easier to encode than the 'long vowel cases', while being the unmarked one in the phonetic alphabet. The rule file now takes advantage of contexts being blocked as well, by introducing an identity rule for the 'short vowel case', which serves no other purpose than excluding it from further processing, and then lengthening all remaining vowels. There is no way to automatically encode the first rule using a replace expression, since it completely relies on context blocking.

The replace expression also fails in other cases, when the context rule encodes an actual transformation. It cannot, for instance, express metathesis. Consider this rule from the Punjabi transliterator:

```
ੱ[cons]    [.]:
```

In Gurmukhi script, the diacritic ੱ geminates the following consonant (Ager 2011). When transliterating to X-SAMPA, this gemination marker 'moves' to the other side of the consonant, where it is expressed as `:`. To encode this as a replace expression, we would need to distribute the changes over multiple composed transducers:

```
{ੱ} -> 0 || _ <cons> .o. 0 -> {:} || <cons> _
```

[4]

[4]For the sake of readability, `<cons>` is a placeholder for the regular expression encoding the corresponding context group.

However, the second transducer will insert `:` after any consonant, not only those formerly preceded by the gemination diacritic. The sequence ਪੱਪ, previously transliterated to `p@ੱp@`, will first be converted to `p@p@` and then wrongly to `p:@p:@` instead of `p@p:@`. Hence, we need to insert a marker, e.g. `!`, wherever we apply part of the rule, to keep track of our changes:

```
{ੱ} -> {!} || _ <cons> .o. 0 -> {:} || {!}<cons> _ .o. {!} -
> 0
```

This will yield the correct transliteration in three steps: `p@ੱp@` → `p@!p@` → `p@!p:@` → `p@p:@`.

Unfortunately, the compose operation disables the simultaneous notation, so we cannot integrate our composed context transducer into the rest of the file's rules. The following replace expression cannot be parsed by `hfst-regexp2fst`:

```
[{ੱ} -> {!} || _ <cons> .o. 0 -> {:} || {!}<cons> _ .o. {!} -
> 0] ,, {x} @-> {y}
```

Apparently, any rule modifying characters on both sides of a context cannot be converted to a replace expression, as long as it is not the only rule of a file. Hence, replace expressions are not suited for representing the EVOLAEMP rule files, since they cannot properly encode all context rules.

5.3.2 Regex approach

Since the convenient replace expressions do not work for us, we must resort to regular expressions. While they are not looped automatically and lack the greedy operator, they have a less restrictive syntax and could thus provide a way to encode context rules.

As already discussed in section 4.1, converting the simple rules from `ale2xsampa` into a regular expression is easy:

```
{ch}:{t)S}
```

Context rules, however, cannot be rewritten that way:

```
[{i}[{q_h}|{X}|{R}]]:[{e}[{q_h}|{X}|{R}]]
```

Again, this transducer unintentionally computes the cross product of its two sides, leading to a highly ambiguous output. Thus, we have to first divide it up into transformative and identity relations.

In order to do this, we first split both sides of our context rule into sequences of literals and individual context groups:

```
<i> <[uvular]>, <e> <[.]>
```

Then we align all of the context groups as well as the literals in between:

```
<i>  <[uvular]>
 ↕       ↕
<e>    <[.]>
```

This alignment can finally be translated into a concatenated sequence of individual transducers:

```
{i}:{e} [{q_h}|{X}|{R}]
```

The resulting transducer unambiguously corresponds to the relation expressed by our rule.

To further illustrate the alignment process, let us take a look at this very complex rule from the Punjabi transliterator:

```
#def    vowel    [@ a: E: e: I i: O: o: U u:]
#def    voiced    [b d_d d` d)Z g]
[vowel][voiced]_h[vowel]    [.]_R[.][.]_F
```

Formerly voiced aspirated consonants, still recognizable as such from the script, have caused neighboring vowels to receive tone depending on their position in the word while losing their aspiration. In the middle of the word, the preceding vowel receives rising tone and the following vowel receives falling tone (Bowden 2012). Splitting this rule gives us the following items:

```
<[vowel]> <[voiced]> <_h> <[vowel]>, <[.]> <_R> <[.]> <[.]> <_F>
```

Obviously, the two sides of this rule cannot be aligned one-to-one as in the Aleut rule above. The output side contains one more item and literal sequences get deleted and inserted between the two sides. However, literals can easily matched to the empty string on either side, which is why it is so important to align the contexts first and then match the intervening literal sequences either to each other or to empty strings. For the Punjabi rule, we get the following alignment:

```
<[vowel]>    0    <[voiced]>  <_h>  <[vowel]>    0
    ↕        ↕        ↕        ↕        ↕        ↕
  <[.]>    <_R>     <[.]>      0     <[.]>    <_F>
```

The regular expression for this rule then looks as follows (the individual transducers have been separated by line breaks instead of whitespaces for better readability):

```
[{@}|{a:}|{E:}|{e:}|{I}|{i:}|{O:}|{o:}|{U}|{u:}]
0:{_R}
[{b}|{d_d}|{d`}|{d)Z}|{g}]
{_h}:0
[{@}|{a:}|{E:}|{e:}|{I}|{i:}|{O:}|{o:}|{U}|{u:}]
0:{_F}
```

We are now able to convert both simple and context rules into XFST regular expressions. However, in contrast to replace expressions, regular expressions only match a single instance of their input side. They still have to be looped and designed to ignore all characters not covered by a rule. We can embed our single rule transducers in a larger regular expression of the form:

```
[<rule_1> | <rule_2> | … | <rule_n> | ?]*
```

This expression has two crucial deficiencies, though: First, it is neither greedy, nor does it apply the rules in order. Thus, whenever a sequence may be covered by two or more rules, the output becomes ambiguous. The sequence `hng`, which already gave us problems in section 2.3, will be transliterated as both `N_0` and `n_0G`. Second, the identity relation `?` can not only be applied when no other rule matches, but on every iteration. Each letter in the sequence `hng`, for example,

can also be matched by `?`. Hence, we get six ambiguous transliterations just for `hng`:

```
N_0
n_0G
n_0g
hN
hnG
hng
```

A solution for the second problem would be to replace `?` by the complement of the input sides of all rules, so that only characters not matched by any rule can be skipped. But especially with larger context groups, this would make the regular expression unnecessarily long. Also, it does not solve the problem with the ambiguities created by the rules themselves.

So while the regular expressions are able to represent both simple and context rules in isolation, they cannot be looped together over the input string without creating ambiguous output.

5.3.3 Hybrid approach

While XFST's replace expressions almost perfectly model the naive transliterator's behavior using the simultaneous notation and the greedy operator, they cannot represent all context rules. Regular expressions, on the other hand, are ideal for encoding both simple and context rules, but generate wrong output when looped, since they lack a way to prioritize certain rules. It could thus be worthwhile to combine the two approaches.

In order to benefit from the strengths of both expression types, the rules are applied in two steps. First, we utilize the greedy operator of the replace expressions to extract sequences for the rules to match. Then, we actually apply the rules using an unambiguously looped regular expression.

The replace expressions provide a special marking format primarily designed for enclosing certain parts of a string in brackets. It is of the form `A -> X ... Y`, and will place the expression `X` before and `Y` after every instance of `A`, while keeping `A` unchanged (Beesley and Karttunen 2003). Here, the sequence `...` is used as a placeholder for `A`. This enables us to also encode context rules in replace expressions.

In the first step, we can thus greedily bracket the input sides of our rules in a transducer `brac`:

```
[{aa} | {ch} | … | {'}] @-> {⟦} ... {⟧}
```

For context rules, it looks like this:

```
[[{i} [{q_h}|{X}|{R}]] | … | [{u:} [{q_h}|{X}|{R}]]] @-> {⟦}
... {⟧}
```

The characters ⟦ (U+27E6) and ⟧ (U+27E7) were chosen as brackets since they do not have a meaning in X-SAMPA and are overall very unlikely to appear in the input.

These bracketed input sides can then be matched by a regular expression. By including the brackets on the input side of the rule transducers, only those sequences marked by the greedy operator in the first step will be matched. In addition, all substrings which can be matched by a rule start with the same character, ⟦. Conversely, all intervening characters, which we want to skip, are not ⟦. Thus, we do not have to replace the identity relation `?` with the complement of all input sides, but only with the complement of ⟦.

So in the second step, we construct a transducer `trans` from the following regular expression:

```
[{⟦aa⟧}:{a:} | {⟦ch⟧}:{t)S} | … | {⟦'⟧}:0 | \{⟦}]*
```

For context rules, it looks like this:

```
[[{⟦}:0 {i}:{e} [{q_h}|{X}|{R}] {⟧}:0] | … | [{⟦}:0 {u:}:{o:}
[{q_h}|{X}|{R}] {⟧}:0] | \{⟦}]*
```

To create the single transducer for the whole rule file, `brac` and `trans` are composed.

5.3.4 Issues

The finite state transducer resulting from the hybrid expression achieves the same results as the naive transliterators in almost all cases. Still, it does not proceed in the same way and may rarely produce output different from that of the naive implementation.

Even though the greedy application of the FST approach models what should be achieved by the in-order application of the naive transliterators, they are not equal. Remember these two lines from our first version of `ale2xsampa`:

```
hn    n_0
hng    N_0
```

The naive transliterator wrongly converted all instances of `hng` to `n_0G` instead of `N_0`, because it applied the `hn` → `n_0` rule first and never tried the second one, which is why we swapped these rules in section 2.3. This is not necessary for the FST transliterator: Due to its greediness, it will always apply `hng` → `N_0` first, irrespective of its position in the rule file.

While the greediness of the FST implementation allows for greater flexibility in rule ordering and will thus usually produce better results than the naive implementation, it can rarely generate ambiguities. Consider these two rules I wrote when improving the German transliterator (they have been removed again for different reasons):

```
[vowel]:r#    [.]:6_^#
[vowel]:[cons]#    [.][.]#
```

The first rule maps word-final `r` to `6_^` while preserving the length of the preceding vowel, as in *kostbar* `kOstba:6_^` 'valuable'. The second rule removes the length marker in other word-final closed syllables, as in *etwas* `?Etvas` 'something'. But since it is used in some other rules in this file too, the group `cons` also contains `r`, so that there are actually two possibilities for transliterating the

sequence `[vowel]:r#`. This is no problem for the naive transliterator, because it does not look for alternatives once it has found a matching rule and will thus always apply the first one. But the FST transliterator will apply both, resulting in an ambiguous output.

So my aim to create a transliterator that can adopt the naive's rule files without changes has not quite been reached. Before exchanging the two systems, the output of both must be compared for each language to make sure that there are no ambiguous rules. But these should be very rare and easy to resolve.

5.4 Transliterating input

After having discussed how the rewrite rules are converted to a finite state transducer `translit.ol`, I will now explain how this transducer is used to transliterate the user input.

A language-specific transliterator such as `AleutToIPATransliterator` consists of a constructor, a `convert` method for files, a `convert` method for individual strings and a static `compile` method that calls `FSTBuilder` to create the transducer. Furthermore, it has a `main` method that parses the user command and activates the other methods accordingly.

As demonstrated in section 5.1, `AleutToIPATransliterator` may be addressed and used in three different ways:

- `java AleutToIPATransliterator` → start interactive mode
- `java AleutToIPATransliterator compile` → (re-)build transducer
- `java AleutToIPATransliterator <file_1> <file_2>` → transliterate content of `<file_1>` to `<file_2>`

Each of these modes needs a different method of `AleutToIPATransliterator`: The keyword `compile` simply activates the `compile` method. In the other two cases, the constructor is evoked, which initializes the required `TerminalSymbolsAdder` and `FiniteStateTransliterator`, and checks whether there already is a transducer `translit.ol` in the working directory. If not, it will also call the `compile` method. In interactive mode, the user input is then repeatedly sent to the single-string `convert` method, while in file mode, the files specified by the user are transferred to the file `convert` method.

Both `convert` methods first hand their input to the `TerminalSymbolsAdder` which insert the word boundary mark `#` around lines and between character sequences separated by whitespaces. Its output is then passed to the `FiniteStateTransliterator`, where the actual transliteration as specified in the rule files takes place.

Like the `FSTBuilder`, `FiniteStateTransliterator` uses the `Terminal` class to interact with HFST via the `bash`. It does not read and use the language's finite state transducer itself, but merely provides an HFST tool with the relevant input.

There are several tools which can be used to run a transducer on a string: `hfst-xfst`, `hfst-lookup` and `hfst-proc` (*HFST: Command Line Tools* 2016).

`hfst-xfst` is a wrapper around the original XFST interface, which is a full-fledged program for creating, manipulating and applying finite state transducers. However, it is interactive and thus cannot be executed in a single command, which makes it complicated to be addressed from within a Java program. `hfst-lookup` and `hfst-proc`, on the other hand, while also providing an interactive mode, may be run on an input file or string individually. While `hfst-proc` is designed for morphological analysis and generation, `hfst-lookup` is intended for general application of transducers to user input. Also, `hfst-lookup` turned out to perform faster than `hfst-proc`, which is why it was chosen as the FST execution utility for `FiniteStateTransliterator`. This is also why `FSTBuilder` converts the transducer it builds to HFST's optimized lookup format: This format guarantees optimal performance of `hfst-lookup`.

Like `AleutToIPATransliterator`, `FiniteStateTransliterator` has two separate `convert` methods for strings and files, though they only differ slightly in the commands they execute. For converting a whole file, the following command is run:

```
hfst-lookup -i <transducer> -I <input-file> | cut -f 2 | sed
-e '/^\s*$/d' > <output-file>
```

The `-i` and `-I` options specify the transducer to use and the input file to read from, respectively. The output of this also includes other information such as the original input string and the weights on each line, which is why it is piped to `cut` which extracts only the column containing the output strings. This list then still contains an empty line after each string, which is removed by `sed`. Finally, the output is stored in the specified output file.

Similarly, the single-string `convert` method executes:

```
echo '<input-string>' | hfst-lookup -i <transducer> | cut -f
2 | sed -e '/^\s*$/d'
```

Here, the input string is handed to `hfst-lookup` via `echo`. Instead of writing the output to a file, it is retrieved from the `Terminal` class and returned by the `convert` method, so that the interactive `AleutToIPATransliterator` can display it to the user.

6 Performance

In order to find out whether the FST implementation actually surpasses the naive implementation in terms of runtime, the transliterators for Aleut (ale), Avar (ava), German (deu), Malayalam (mal) and Punjabi (pan) of both types were tested on input files of varying size.

6.1 Method

These five languages were chosen for the different amount and complexity of rules as well as their different original orthographies. Avar so far only has a single simple grapheme-to-phoneme transliterator file with no context rules. Aleut, as we have seen in the previous section, employs a moderate number of

	ale	ava	deu	mal	pan
Files	3 (+2)	1 (+2)	6 (+2)	5 (+2)	6 (+2)
Rules	36 (+394)	56 (+394)	120 (+394)	155 (+394)	126 (+394)
Ctxt. ref.s	15	0	48	33	36
Group size	∼7	/	∼17	∼14	∼9

Table 1: Some statistics of the rule systems for Aleut, Avar, German, Malayalam and Punjabi: The number of rule files and rules (in addition to the 394 simple rules in `lowercase` and `xsampa2ipa`), the number of times context groups are referenced in the rules and their mean size.

files with some context rules. The German, Malayalam and Punjabi transliterators access a large number of files with complex phonological rules making heavy use of context groups. German in particular employs many large context groups, while Malayalam has the most rules overall. Detailed statistics about the five languages' rule systems are shown in Table 1. Their size and complexity should relate directly to the size and build time of the transducer created from them.

Aleut and German use a Latin script with some special characters, while Avar has a Cyrillic script and Malayalam and Punjabi employ two distinct abugidas. It might be interesting to see if the transliterators perform differently depending on the Unicode range they are converting from.

To test the transliterators in a realistic environment, the actual NorthEuraLex word lists for the five languages were used as input, despite their non-uniform sizes. German contains 1,016 words, while Aleut contains 1,032, Avar contains 1,224 and Malayalam contains 1,062. For Punjabi, there was no word list available yet, so a list of 1,036 words was extracted from Punjabi Wikipedia articles. The differing word counts within the NorthEuraLex word lists are due to missing or multiple synonymous translations for a given concept.

While they could have been easily cut to contain an equal number of words, I decided against it, because even this would not yield same-length files: Equal number of words does not mean equal number of characters. Malayalam, for instance, has much longer words than German due to the abugida script and its highly agglutinative morphology (Asher and Kumari 1997). Its word list contained 23,085 characters, while the German list only has 6,426 characters. There is no way to reasonably unify this; also, these are the environments the transliterators are expected to run in. The Malayalam transliterator will always be used to transliterate Malayalam and will thus on average always encounter longer words than its German counterpart.

The basic word lists were enlarged in two different dimensions to investigate the systems' performance in relation to input size:

Height: The number of lines was multiplied by factors 2, 4 and 6 by appending copies of the original list to test performance on longer word lists.

Width: The length of each line was multiplied by factors 2, 4 and 6 by repeating each word in a line to test performance on longer paragraphs.

I expect the FST implementation to perform better than the naive implemen-

	ale	ava	deu	mal	pan
States	484	437	1,753	2,046	2,667
Transitions	28,178	21,452	214,441	100,909	62,853
Build time	3.37 sec.	2.97 sec.	10.83 sec.	7.60 sec.	6.17 sec.

Table 2: Some statistics of the transducers for Aleut, Avar, German, Malayalam and Punjabi: The number of states and transitions of the resulting transducer and the time needed to build it.

tation on both sets of lists, but especially in the width dimension, where the at least quadratic increase in runtime of the naive system will probably make it extremely slow on longer paragraphs. The naive implementation should therefore also perform significantly worse on the long Malayalam words than on those of the other languages.

The tests were run in a VirtualBox equipped with Ubuntu 14.04 (64-bit), 2 GB RAM and 2 CPU cores. One can expect the transliterators to perform faster on physical machines with better hardware, but the runtimes measured in the virtual environment still provide a good comparison of the two implementations. To account for performance fluctuations, the values presented in the following sections are the medians of three consecutive test runs.

The times were measured within the respective transliterators from the start to the end of the `convert` method, and additionally from the start to the end of the `compile` method for the FST transliterator.

6.2 Results

6.2.1 Transducers

As expected, the size and build time of the finite state transducers reflects the size and complexity of the rule files they were derived from. Aleut, having the least rules, and Avar, having no context groups, yield the smallest transducers and lowest build times, the Aleut one being only slightly larger than the Avar one. The German, Malayalam and Punjabi transducers are significantly larger and need much more time to be created. German has by far the most transitions, and more than twice as many as Malayalam, perhaps due to its many huge context groups. It also has the highest build time, needing almost 11 seconds to be constructed. Curiously, Punjabi has much more states than both German and Malayalam, but much less transitions, even though its rule files are overall smaller than those of German and Malayalam. All values can be seen in Table 2.

6.2.2 Width

On the basic word lists, the naive transliterators still perform faster than the FST implementation, especially if you add the build time for the individual transducers. But as predicted, the runtime of the naive transliterators gets increasingly worse on longer paragraphs. At width 2, the FST system already

	Basic list		**Width 2**		**Width 4**		**Width 6**	
	Naive	*FST*	*Naive*	*FST*	*Naive*	*FST*	*Naive*	*FST*
ale	0.29	0.61	2.21	0.76	19.48	1.08	76.27	1.40
ava	0.23	0.58	1.81	0.73	16.42	1.02	65.02	1.35
deu	0.69	0.63	5.32	0.82	46.63	1.05	178.08	1.36
mal	2.09	2.20	17.50	2.73	165.21	3.82	665.17	4.85
pan	0.78	2.82	6.34	3.48	56.09	4.76	216.39	6.08

Table 3: Seconds needed to transliterate Aleut, Avar, German, Malayalam and Punjabi word lists with varying line lengths (normal length, length $*$ 2, $*$ 4, $*$ 6) by the naive and FST transliterators.

outperforms the naive one by far. At widths 4 and 6, the FST implementation is significantly faster even when adding build time. Table 3 shows the precise times for each run.

As expected, the Malayalam word list is the slowest to be processed by the naive transliterator. At width 6, it takes 11 minutes to be converted, while the FST implementation only needs less than five seconds to process the same input.

Interestingly, the Punjabi transliterator is the slowest in the FST implementation, while it performs similar to the German one in the naive implementation. For the naive transliterators, the runtime is directly related to the number of rules for each language. In the FST implementation, however, Aleut, Avar and German perform equally fast, while Malayalam and Punjabi need about four times as long on the same width. This cannot (only) be explained by word length: The Avar word list has both more lines and more characters than the Punjabi word list. Also, the basic Malayalam and Punjabi word lists are processed slower than the other three languages at width 6, where they all contain much more characters than width 1 Malayalam and Punjabi.

What distinguishes Malayalam and Punjabi from the other three languages is that both employ their own scripts which are encoded in the Unicode range for which UTF-8 needs 3 bytes per character, so the longer runtime could be owed to the script of their word lists. This, however, is not the case, as a quick test showed: The Malayalam transliterator was run on two files, one containing 1,000 lines of 50 Malayalam number zero ൦ characters and the other containing 1,000 lines of 50 `$` characters. Both ൦ and `$` are not covered by any of the rules in the Malayalam transliterator, including the `xsampa2ipa` file. The transliterator performed equally on these two files, processing both within 3.2 seconds. Thus, the code points in the input file are not responsible for the longer runtime of the Malayalam and Punjabi transliterators. The reason might as well lie in the encoding or structure of the transducer, but I did not further pursue this question here.

6.2.3 Height

Both the naive and the FST implementation's runtime increases linearly relative to the number of lines in the input file. While this can be expected for the

	Basic list		Height 2		Height 4		Height 6	
	Naive	*FST*	*Naive*	*FST*	*Naive*	*FST*	*Naive*	*FST*
ale	0.29	0.61	0.54	1.16	1.08	2.32	1.63	3.51
ava	0.23	0.58	0.49	1.17	0.88	2.26	1.34	3.38
deu	0.69	0.63	1.40	1.19	2.69	2.21	4.06	3.39
mal	2.09	2.20	4.20	4.35	8.40	8.61	12.66	12.83
pan	0.78	2.82	1.61	5.59	3.19	11.09	4.72	16.59

Table 4: Seconds needed to transliterate Aleut, Avar, German, Malayalam and Punjabi word lists with varying file lengths (normal length, length $*$ 2, $*$ 4, $*$ 6) by the naive and FST transliterators.

naive transliterators, it is surprising for the FST ones. Apparently, sequences separated by newlines are treated separately and not as a single stream. This leads to the naive implementation outperforming the FST one also at higher height values. Table 4 shows the precise times for each run.

6.3 Discussion

After having examined the actual runtimes of the two systems, I now discuss them regarding the goals of my FST implementation as stated in section 1. My transliterators were supposed to

1. achieve better overall performance than the naive implementation,
2. be suitable for transliterating whole texts, i.e. files with lines containing complete paragraphs instead of single words, and
3. generate the same output as the naive implementation for backward compatibility.

As shown in section 6.2.2, the FST system is much faster on longer paragraphs. Its runtime increases linearly in relation to the length of the line, and only slightly. The naive implementation displays a steep increase in runtime and processes longer paragraphs extremely slowly. Thus, the FST transliterators are well suited for processing real texts.

However, the naive implementation performs slightly better on the simple word lists. In section 6.2.3 we have seen that both the FST and naive implementation have the same runtime growth in relation to the number of lines, so the naive system still outperforms the FST one on simple word lists. Thus, the FST implementation does not achieve better overall performance.

Seeing that it processes the same amount of text significantly faster when divided into fewer, but longer lines, we could increase its overall performance greatly by removing newlines from the input before passing it to the transducer and reinserting them afterwards. Removing all line breaks, however, does not work. I also tested the two systems on real texts, but discovered that the longer paragraphs were not converted at all by the FST transliterators[5]. Apparently,

[5]Also, the naive transliterators took hours to convert the texts, which is why the test was canceled and replaced by the simpler one described previously.

the `hfst-lookup` utility has a character cap on lines which, depending on the input, lies somewhere around 2,500. `hfst-xfst` does not have such a limit, but will eventually run out of memory and crash when given very long paragraphs. This means that, in order to actually be able to process real texts, the FST system also needs to insert line breaks after a certain number of characters to guarantee conversion of the whole text. A preprocessing step which removes all newlines and inserts new line breaks at word boundaries after e.g. 2,000 characters, would solve both problems. This would compress the basic Malayalam word list to 12 lines and the German one to just four. The performance boost gained from this should make the FST system outperform the naive one even on long word lists.

All in all, the system as presented in this thesis does not quite reach the set goals. While it does achieve better results than the naive implementation on longer paragraphs and thus full texts, it performs worse on actual word lists, its primary purpose within the EVOLAEMP project. However, I have just presented a possible solution for this problem, which should greatly increase its processing speed. Also, while the FST implementation generally generates the same output as the previous system, it may produce ambiguities in rare cases, as explained in section 5.3.4, so the rule files need to be checked before implementing the new systems, though most of them should already be unambiguous.

7 Conclusion

In this thesis, I have developed, presented and tested a revision of the current EVOLAEMP transliterator system based on finite state transducers. Like the previous system, my implementation is based on rewrite rules designed specifically for each language. It converts these rules into XFST regular expression from which a single HFST finite state transducer is created. Input is then converted to IPA by feeding it to this transducer.

This system performs faster than the previous implementation on longer lines of characters, but is slower on lists of single words. Thus, it is quite suitable for transliterating full texts, but requires some improvements to be a replacement for the EVOLAEMP cognacy list transliterators. Also, it operates differently from the previous system and will on rare occasions produce different output. Thus, all language transliterators have to be checked for ambiguous rules before a potential migration to the FST implementation.

However, I presented a possible solution at least for the performance problem, which exploits the new system's good speed on larger paragraphs by splitting all input into maximally long lines in a preprocessing step. It will be interesting to see how much this boosts the FST implementation's overall performance.

To further improve the FST transliterators, the effect of input and rule encoding as well as transducer size on conversion speed should be examined. The Malayalam and Punjabi transliterators are significantly slower than the Avar, Aleut and German ones, which does apparently not relate to the Unicode code points of the input, the file's length and size or the number of states and transitions of the transducer. There must be another feature of Malayalam and Punjabi

specifically that leads to bad performance. A large-scale performance test on further languages might give some insights into the cause(s).

Still, the FST transliterator system is a good approach at an improvement of the previous one. After implementing the line compression, it should easily outperform the naive transliterators. Also, it is able to process even longer texts with linear runtime growth. Thus, it can also be applicable outside the scope of providing transcriptions of NorthEuraLex word lists. Additionally, the individual transducers are fully functional on their own and can be converted into HFST, XFST, SFST, OpenFST and foma formats by HFST, so that they may be distributed independently and easily be integrated into other applications. The `FSTBuilder` can also serve as a standalone program, providing a simpler way to formulate FSTs for people unfamiliar with regular expressions.

8 Bibliography

References

Ager, Simon (2011). *Omniglot: Punjabi language and the Gurmukhi and Shahmuhi scripts and pronunciation*. URL: http://www.omniglot.com/writing/punjabi.htm (visited on 09/14/2016).

Asher, Ronald E. and T. C. Kumari (1997). *Malayalam*. Psychology Press.

Beesley, Kenneth R. and Lauri Karttunen (2003). *Finite state morphology*. Center for the Study of Language and Information.

Bergsland, Knut (1994). *Aleut Dictionary (Unangam Tunudgusii). An Unabridged Lexicon of the Aleutian, Pribilof, and Commander Islands Aleut Language*. Alaska Native Language Center, University of Alaska Fairbanks.

Bouma, Gosse (2000). "A finite state and data-oriented method for grapheme to phoneme conversion." In: *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*. Association for Computational Linguistics, pp. 303–310.

Bowden, Andrea Lynn (2012). "Punjabi Tonemics and the Gurmukhi Script: A Preliminary Study." MA thesis.

Braga, Daniela and Luís Coelho (2006). "Letter-to-sound conversion for Galician TTS systems." In: *Actas de las IV Jornadas en Tecnologia del Habla, Zaragoza*, pp. 171–176.

Brown, Cecil H. et al. (2008). "Automated classification of the world's languages: a description of the method and preliminary results." In: *STUF – Language Typology and Universals. Sprachtypologie und Universalienforschung* 61.4, pp. 285–308.

Caseiro, D., I. Trancoso, and L. Oliveira (2002). "Grapheme-to-phone using finite state transducers." In: *Proceedings of 2002 IEEE Workshop on Speech Synthesis*.

Chomsky, Noam and Morris Halle (1968). *The sound pattern of English*. Harper & Row.

Dellert, Johannes et al. (2016). "A deep-coverage lexical database of Northern Eurasia." Presentation held at PLM2016.

EVOLAEMP. Language Evolution: The Empirical Turn (2016). URL: http://www.evolaemp.uni-tuebingen.de/ (visited on 08/25/2016).

Greenhill, Simon J., Robert Blust, and Russell D. Gray (2008). "The Austronesian basic vocabulary database: from bioinformatics to lexomics." In: *Evolutionary Bioinformatics* 4, p. 271.

Helsinki Finite-State Transducer Technology (HFST) (2016). URL: http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/ (visited on 08/10/2016).

HFST: Command Line Tools (2016). URL: https://kitwiki.csc.fi/twiki/bin/view/KitWiki/HfstCommandLineTools (visited on 09/22/2016).

Indo-European Lexical Cognacy Database (2016). URL: `http://ielex.mpi.nl/` (visited on 08/10/2016).

Kaplan, Ronald M. and Martin Kay (1994). "Regular models of phonological rule systems." In: *Computational linguistics* 20.3, pp. 331–378.

Kiparsky, Paul (1968). "Linguistic Universals and Linguistic Change." In: *Universals in Linguistic Theory*. Ed. by Emmon Bach and Robert T. Harms. Holt, Rinehart & Winston, pp. 170–202.

Koskenniemi, Kimmo and Anssi Yli-Jyrä (2008). "CLARIN and Free Open Source Finite-State Tools." In: *FSMNLP*, pp. 3–13.

Toma, Ştefan-Adrian and Doru-Petru Munteanu (2009). "Rule-based automatic phonetic transcription for the Romanian language." In: *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*. IEEE, pp. 682–686.

Wells, John C. (1995). *Computer-coding the IPA: a proposed extension of SAMPA*.

9 Appendix

9.1 Transliterator.java

```
import java.io.File;
import java.io.FileNotFoundException;

public abstract class Transliterator {

  public abstract void convert(File in, File out) throws
      FileNotFoundException;
  public abstract String convert(String s);

}
```

9.2 AleutToIPATransliterator.java

```
/**
 * The specific transliterator converting from Aleut
 * Roman orthography to IPA.
 *
 * Based loosely on the language transliterators of
 * Johannes Dellert.
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class AleutToIPATransliterator extends Transliterator {

  private static final File[] FILES = new File[]{new File("lowercase")
      ,
        new File("ale2xsampa"),
        new File("ale-vowels1"),
        new File("ale-vowels2"),
        new File("xsampa2ipa")};

  private TerminalSymbolsAdder tsa;
  private FiniteStateTransliterator translit;

  public AleutToIPATransliterator() {
    tsa = new TerminalSymbolsAdder();

    File fst = new File("translit.ol");
    if (!fst.exists() || !fst.isFile()) {
      compile();
      fst = new File("translit.ol");
    }

    translit = new FiniteStateTransliterator(fst);
  }
```

```
@Override
public void convert(File in, File out) {
  try {
    System.err.print("Transliterating " + in.getName() + "... ");
    File tmp = new File("TMPX");
    tsa.convert(in, tmp);
    translit.convert(tmp, out);
    tmp.delete();
    System.err.println("Done.");
  }
  catch (FileNotFoundException e) {
    System.err.println("Cancelled. File not found.");
  }
}

@Override
public String convert(String s) {
  String res = translit.convert(tsa.convert(s));
  return res;
}

public static void compile() {
  System.err.print("Building FST... (This might take a while.)");
  for (File f : FILES)
    if (!f.isFile()) {
      System.err.println(" Cancelled. " + f.getName() + " does not
          exist or is not a file.");
      return;
    }
  FSTBuilder.build(FILES);
  System.err.println(" Done.");
}

public static void main(String[] args)
{
  if (args.length == 0) {
    AleutToIPATransliterator translit = new AleutToIPATransliterator
        ();
    System.err.println("AleutToIPATransliterator ready, waiting for
        input (Ctrl+C to quit):");
    Scanner in = new Scanner(System.in);

    while (in.hasNextLine())
      System.out.println(translit.convert(in.nextLine()));

    in.close();
  }
  else {
    if (args[0].equals("compile"))
      compile();
    else {
      if (args.length != 2)
        System.err.println("Usage: AleutToIPATransliterator [lemma
            file] [output file]\n"
```

```
            + "   or: AleutToIPATransliterator compile");
        else
          new AleutToIPATransliterator().convert(new File(args[0]),
              new File(args[1]));
      }
    }
  }
}
```

9.2.1 ale2xsampa

```
// Source: Knut Bergsland - Aleut Dictionary (1994)

aa  a:
ch  t)S
d D
ĝ R
g G
hd  T
hl  K
hm  m_0
hng N_0
hn  n_0
hw  W
hy  C
ii  i:
ng  N
oo  o:
q q_h
r r\
uu  u:
x̂ X
y j
'
```

9.2.2 ale-vowels1

```
// Source: Knut Bergsland - Aleut Dictionary (1994)

#def  uvular  [q_h X R]
#def  coronal [t d t)S s z n_0 n K l r\]

[uvular]i [.]e
[uvular]a [.]A
[uvular]u [.]o

[coronal]a  [.]E
[coronal]u  [.]y
```

9.2.3 ale-vowels2

```
// Source: Knut Bergsland - Aleut Dictionary (1994)

#def  uvular  [q_h X R]
```

```
#def  coronal [t d t)S s z n_0 n K l r\]

i[uvular] e[.]
a[uvular] A[.]
u[uvular] o[.]
i:[uvular]  e:[.]
a:[uvular]  A:[.]
u:[uvular]  o:[.]

a[coronal]  E[.]
u[coronal]  y[.]
a:[coronal] E:[.]
u:[coronal] y:[.]
```

9.3 TerminalSymbolsAdder.java

```
/**
 * This class encloses words in word boundary marks (#).
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class TerminalSymbolsAdder extends Transliterator {

  @Override
  public String convert(String str) {
    return "#" + str.replace(" ", "# #") + "#";
  }

  @Override
  public void convert(File in, File out) throws FileNotFoundException
      {
    Scanner read = new Scanner(in);
    PrintWriter writ = new PrintWriter(out);
    while (read.hasNextLine())
      writ.println(convert(read.nextLine()));
    read.close();
    writ.close();
  }
}
```

9.4 FiniteStateTransliterator.java

```
/**
 * This class feeds strings to an FST.
 */

import java.io.File;

public class FiniteStateTransliterator extends Transliterator {
```

```
  Terminal t;
  File fst;

  public FiniteStateTransliterator(File fst) {
    this.t = new Terminal();
    this.fst = fst;
  }

  public void setFST(File fst) {
    this.fst = fst;
  }

  /**
   * Feeds a whole file to the FST and writes output directly to the
       specified output file.
   * @param infile Input file
   * @param outfile Output file
   */
  public void convert(File infile, File outfile) {
    t.run("hfst-lookup -i " + fst.getAbsolutePath() + " -I " + infile.
        getAbsolutePath() + " | cut -f 2 | sed -e '/^\\s*$/d' > " +
        outfile.getAbsolutePath());
    String error = t.error();
    if (!error.equals(""))
      System.err.println("\n" + error);
  }

  /**
   * Feeds a single string to the FST and returns the result.
   * @param s String to be converted
   * @return String transformed by the FST
   */
  public String convert(String s) {
    t.run("echo '" + s + "' | hfst-lookup -i" + fst.getAbsolutePath()
        + " | cut -f 2 | sed -e '/^\\s*$/d'");
    String error = t.error();
    if (!error.equals(""))
      System.err.println("\n" + error);
    return t.output();
  }
}
```

9.5 FSTBuilder.java

```
/**
 * This class builds a minimal Helsinki Finite State Transducer in
     optimized lookup
 * format "translit.ol" from files with replacement rules.
 */

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
```

```java
import java.util.Scanner;

public class FSTBuilder {

  /**
   * Takes an array of rule files, generates terminal commands to
       build the FST
   * from them and runs the commands.
   * @param infiles An array with all rule files
   */
  public static void build(File[] infiles) {
    try {
      // Write terminal commands for FST creation to file cmd.txt
      File outfile = new File("cmd.txt");
      PrintWriter wrt = new PrintWriter(outfile);
      Scanner read;
      boolean first = true;

      for (File infile : infiles) {
        read = new Scanner(infile, "UTF-8");
        appendRules(read, wrt);
        read.close();

        if (first) {
          wrt.append("mv FSTfile FSTresult\n");
          first = false;
        }
        else {
          wrt.append("hfst-compose FSTresult FSTfile > TMPappend\n");
          wrt.append("mv TMPappend FSTresult\n");
        }
      }

      wrt.append("hfst-minimize FSTresult > TMPmin\n");
      wrt.append("hfst-fst2fst --optimized-lookup-unweighted TMPmin >
          translit.ol\n");
      wrt.close();

      // Run commands from cmd.txt in terminal
      Terminal ter = new Terminal();
      read = new Scanner(outfile);
      while (read.hasNextLine())
        ter.run(read.nextLine());
      read.close();

      // Delete created files
      ter.run("rm TMP*\n");
      ter.run("rm FST*\n");
      ter.run("rm cmd.txt");
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
    }
  }
```

```
/**
 * Takes a Scanner with a single rule file and generates commands to
     create
 * an FST from it.
 * @param read A Scanner with a rule file
 * @param wrt The PrintWriter with the file to write the commands to
 */
private static void appendRules(Scanner read, PrintWriter wrt) {
  HashMap<String, String[]> groups = new HashMap<>(); // context
      groups
  String line;
  String[] parts;
  String[] regex;
  StringBuilder brac = new StringBuilder("");
  StringBuilder trans = new StringBuilder("");

  while (read.hasNextLine()) {
    line = read.nextLine();

    if (!line.equals("") && !line.startsWith("//")) { // if line is
        neither empty nor a comment
      parts = line.split("\\t");

      if (parts[0].equals("#def")) // get context groups
        groups.put(parts[1], parts[2].substring(1, parts[2].indexOf(
            ']')).split("\\s"));

      else {
        if (!groups.isEmpty() && line.indexOf('[') >= 0) { //
            context rule
          regex = getContextRule(parts, groups);
          brac.append("[" + regex[0] + "]|");
          trans.append("[{⟦}:0 " + regex[1] + " {⟧}:0] | ");
        }
        else { // non-context rule
          brac.append("{" + escape(parts[0]) + "}|");
          if (parts.length < 2)
            trans.append("{⟦" + escape(parts[0]) + "⟧}:0 | ");
          else
            trans.append("{⟦" + escape(parts[0]) + "⟧}:{" + escape(
                parts[1]) + "} | ");
        }
      }
    }
  }

  wrt.append("echo '[" + brac.substring(0, brac.length() - 1).
      replace("{}", "") + "] @-> {⟦} ... {⟧}' | hfst-regexp2fst >
      FSTbrac\n"
  + "echo '[" + trans.toString().replace("{}", "") + "\\{⟦}]*' |
      hfst-regexp2fst > FSTtrans\n"
  + "hfst-compose FSTbrac FSTtrans > FSTfile\n");
}
```

```
/**
 * Takes a context rule and returns the corresponding identity and
     transducer regex
 * to create an FST for it.
 * @param part Array with left and right hand side of the rule
 * @param groups The context groups
 * @return Length-2 array with the identity regex at index 0 and the
     transducer regex
 * at index 1
 */
private static String[] getContextRule(String[] part, HashMap<String
    , String[]> groups) {
  // Replace [.]s in second part, insert |s before group names &
      replace brackets with tabs
  int i;
  int j;
  StringBuilder leftSide = new StringBuilder(part[0]);
  StringBuilder rightSide = new StringBuilder(part[1]);
  String cat;

  while ((i = leftSide.indexOf("[")) >= 0) {
    j = leftSide.indexOf("]");
    cat = "\t|" + leftSide.substring(i+1, j) + "\t";
    leftSide.replace(i, j+1, cat);
    j = rightSide.indexOf("[");
    rightSide.replace(j, j+3, cat);
  }

  // Split into sequences of contexts and literals
  String[] left = leftSide.toString().trim().split("\t+");
  String[] right = rightSide.toString().trim().split("\t+");

  // Align contexts and literals and create commands for FST
      building
  int l = 0; // left side index
  int r = 0; // right side index
  StringBuilder iden = new StringBuilder(""); // left side regex
      only
  StringBuilder trans = new StringBuilder(""); // regex for
      transformation from left to right
  String context;

  while (l < left.length && r < right.length) {
    // identity rule for contexts
    if (left[l].equals(right[r]) && left[l].startsWith("|")) {
      context = getContext(groups.get(left[l].substring(1)));
      iden.append(context + " ");
      trans.append(context + " ");
    }
    else
      // insertion of literal before context
      if (left[l].charAt(0) == '|') {
        trans.append("0:{" + escape(right[r]) + "} ");
```

```
        l--; // only increase r
      }
      else {
        iden.append("{" + escape(left[l]) + "} ");

        // deletion of literal before context
        if (right[r].charAt(0) == '|') {
          trans.append("{" + escape(left[l]) + "}:0 ");
          r--; // only increase l
        }
        // literal transformation
        else {
          trans.append("{" + escape(left[l]) + "}:{" + escape(right[
              r]) + "} ");
        }
      }
    l++;
    r++;
  }

  // Add final deletion or insertion
  if (l < left.length) {
    iden.append("{" + escape(left[l]) + "} ");
    trans.append("{" + escape(left[l]) + "}:0 ");
  }
  if (r < right.length) {
    trans.append("0:{" + escape(right[r]) + "} ");
  }

  // Delete final whitespace
  iden.deleteCharAt(iden.length()-1);
  trans.deleteCharAt(trans.length()-1);

  return new String[]{iden.toString(), trans.toString()};
}

/**
 * Generates the disjunct regex for a context group.
 * @param group The strings in the context group
 * @return The regex
 */
private static String getContext(String[] group) {
  String context = "";

  for (String c : group)
    context += "{" + escape(c) + "}|";

  return "[" + context.substring(0, context.length()-1) + "]";
}

/**
 * Escapes special characters in a rule. Special characters are:
 * - { and } used by HFST, escaped with HFST's escape character %
 * - ' used by echo, escaped with backslash and enclosed in 's
```

```
   * @param s String to escape
   * @return Escaped string
   */
  private static String escape(String s) {
    String r = s.replace("{", "\t1%{\t2");
    r = r.replace("}", "\t1%}\t2");
    r = r.replace("\t1", "}");
    r = r.replace("\t2", "{");
    r = r.replace("'", "'\\''");

    return r;
  }
}
```

9.6 Terminal.java

```
/**
 * This class can run commands in the bash.
 */

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Terminal {

  ProcessBuilder p;
  Process currentProcess;

  public Terminal() {
    p = new ProcessBuilder();
  }

  public void run(String cmd) {
    p.command(new String[]{"bash", "-c", cmd});
    try {
      currentProcess = p.start();
      currentProcess.waitFor();
    }
    catch (IOException | InterruptedException e) {
      System.out.println(e.getMessage());
    }
  }

  public String output() {
    String result = "";
    BufferedReader output = new BufferedReader(new InputStreamReader(
        currentProcess.getInputStream()));

    try {
      String line;
      while ((line = output.readLine()) != null)
        result += line + "\n";
    }
```

```
    catch (IOException e) {
      System.out.println(e.getMessage());
    }

    return result;
  }

  public String error() {
    String result = "";
    BufferedReader error = new BufferedReader(new InputStreamReader(
        currentProcess.getErrorStream()));

    try {
      String line;
      while ((line = error.readLine()) != null)
        result += line + "\n";
    }
    catch (IOException e) {
      System.out.println(e.getMessage());
    }

    return result;
  }
}
```