Finite-state script normalization and processing utilities: The Nisaba Brahmic library

Cibu Johny[†] **Lawrence Wolf-Sonkin**[‡] **Alexander Gutkin**[†] **Brian Roark**[‡]
Google Research
[†]United Kingdom and [‡]United States
`{cibu,wolfsonkin,agutkin,roark}@google.com`

Abstract

This paper presents an open-source library for efficient low-level processing of ten major South Asian Brahmic scripts. The library provides a flexible and extensible framework for supporting crucial operations on Brahmic scripts, such as NFC, visual normalization, reversible transliteration, and validity checks, implemented in Python within a finite-state transducer formalism. We survey some common Brahmic script issues that may adversely affect the performance of downstream NLP tasks, and provide the rationale for finite-state design and system implementation details.

1 Introduction

The Unicode Standard separates the representation of text from its specific graphical rendering: text is encoded as a sequence of characters, which, at presentation time are then collectively rendered into the appropriate sequence of glyphs for display. This can occasionally result in many-to-one mappings, where several distinctly-encoded strings can result in identical display. For example, Latin script letters with diacritics such as "é" can generally be encoded as either: (a) a pair of the base letter (e.g., "e" which is `U+0065` from Unicode's Basic Latin block, corresponding to ASCII) and a diacritic (in this case `U+0301` from the Combining Diacritical Marks block); or (b) a single character that represents the grapheme directly (`U+00E9` from the Latin-1 Supplement Unicode block). Both encodings yield visually identical text, hence text is often normalized to a conventionalized normal form, such as the well-known Normalization Form C (NFC), so that visually identical words are mapped to a conventionalized representative of their equivalence class for downstream processing. Critically, NFC normalization falls far short of a complete handling of such many-to-one phenomena in Unicode.

In addition to such normalization issues, some scripts also have well-formedness constraints, i.e., not all strings of Unicode characters from a single script correspond to a valid (i.e., legible) grapheme sequence in the script. Such constraints do not apply in the basic Latin alphabet, where any permutation of letters can be rendered as a valid string (e.g., for use as an acronym). The Brahmic family of scripts, however, including the Devanagari script used to write Hindi, Marathi and many other South Asian languages, do have such constraints. These scripts are alphasyllabaries, meaning that they are structured around orthographic syllables (akṣara) as the basic unit.[1] One or more Unicode characters combine when rendering one of thousands of legible akṣara, but many combinations do not correspond to any akṣara. Given a token in these scripts, one may want to (a) normalize it to a canonical form; and (b) check whether it is a well-formed sequence of akṣara.

Brahmic scripts are heavily used across South Asia and have official status in India, Bangladesh, Nepal, Sri Lanka and beyond (Cardona and Jain, 2007; Steever, 2019). Despite evident progress in localization standards (Unicode Consortium, 2019) and improvements in associated technologies such as input methods (Hinkle et al., 2013) and character recognition (Pal et al., 2012), Brahmic script processing still poses important challenges due to the inherent differences between these writing systems and those which historically have been more dominant in information technology (Sinha, 2009; Bhattacharyya et al., 2019).

In this paper, we present Nisaba, an open-source software library,[2] which provides processing utilities for ten major Brahmic scripts of South Asia: Bengali, Devanagari, Gujarati, Gurmukhi, Kannada, Malayalam, Oriya (Odia), Sinhala, Tamil,

[1]See §3 for details on the scripts.
[2]`https://github.com/google-research/nisaba/`

and Telugu. In addition to string normalization and well-formedness processing, the library also includes utilities for the deterministic and reversible romanization of these scripts, i.e., transliteration from each script to and from the Latin script (Wellisch, 1978). While the resulting romanizations are standardized in a way that may or may not correspond to how native speakers tend to romanize the text in informal communication (see, e.g., Roark et al., 2020), such a default romanization can permit easy inspection of an approximate version of the linguistic strings for those who read the Latin script but not the specific Brahmic script being examined.

As a whole, the library provides important utilities for language processing applications of South Asian languages using Brahmic scripts. The design is based on the observation that, while there are considerable superficial differences between these scripts, they follow the same encoding model in Unicode, and maintain a very similar character repertoire having evolved from the same source — the Brāhmī script (Salomon, 1996; Fedorova, 2012). This observation lends itself to the script-agnostic design (outlined in §4) that, unlike other approaches reviewed in §2, is based on the weighted finite state transducer (WFST) formalism (Mohri, 2004). The details of our system are provided in §5.

2 Related Work

The computational processing of Brahmic scripts is not a new topic, with the first applications dating back to the early formal syntactic work by Datta (1984). With an increased focus on the South Asian languages within the NLP community, facilitated by advances in machine learning and the increased availability of relevant corpora, multiple script processing solutions have emerged. Some of these toolkits, such as statistical machine translation-based Brahmi-Net (Kunchukuttan et al., 2015), are model-based, while others, such as URoman (Hermjakob et al., 2018), IndicNLP (Kunchukuttan, 2020), and Aksharmukha (Rajan, 2020), employ rules. The main focus of these libraries is script conversion and romanization. In this capacity they were successfully employed in diverse downstream multilingual NLP tasks such as neural machine translation (Zhang et al., 2020; Amrhein and Sennrich, 2020), morphological analysis (Hauer et al., 2019; Murikinati et al., 2020), named entity recognition (Huang et al., 2019) and part-of-speech tagging (Cardenas et al., 2019).

Similar to the software mentioned above, our library does provide romanization, but unlike some of the packages, such as URoman, we guarantee reversibility from Latin back to the native script. Similar to others we do not focus on faithful invertible transliteration of named entities which typically requires model-based approaches (Sequiera et al., 2014). Unlike the IndicNLP package, our software does not provide morphological analysis, but instead offers significantly richer script normalization capabilities than other packages. These capabilities are functionally separated into normalization to Normalization Form C (NFC) and visual normalization. Additionally, our library provides extensive script-specific well-formedness grammars. Finally, in contrast to these other approaches, grammars in our library are maintained separately from the code for compilation and application, allowing for maintenance of existing scripts and languages plus extension to new ones without having to modify any code. This is particularly important given that Unicode standards do change over time and there remain many languages left to cover.

To the best of our knowledge this is the first publicly available general finite-state grammar approach for low-level processing of multiple Brahmic scripts since the early formal syntactic work by Datta (1984) and is the first such library designed based on an observation by Sproat (2003) that the fundamental organizing principles of the Brahmic scripts can be algebraically formalized. In particular, all the core components of our library (inverse romanization, normalization and well-formedness) are compactly and efficiently represented as finite state transducers. Such formalization lends itself particularly well to run-time or offline integration with any finite state processing pipeline, such as decoder components of input methods (Ouyang et al., 2017; Hellsten et al., 2017), text normalization for automatic speech recognition and text-to-speech synthesis (Zhang et al., 2019), among other natural language and speech applications.

3 Brahmic Scripts: An Overview

The scripts of interest have evolved from the ancient Brāhmī writing system that was recorded

Name	Id	IV	DV	C	CO
Bengali	`BENG`	16	13	43	5
Devanagari	`DEVA`	19	17	45	4
Gujarati	`GUJR`	16	15	39	5
Gurmukhi	`GURU`	12	9	39	8
Kannada	`KNDA`	17	15	39	3
Malayalam	`MLYM`	16	16	38	10
Oriya	`ORYA`	14	13	38	5
Sinhala	`SINH`	18	17	41	2
Tamil	`TAML`	12	11	27	1
Telugu	`TELU`	16	15	38	5

Table 1: Sizes of core graphemic classes: Independent vowels (IV), dependent vowel diacritics (DV), consonants (C), coda symbols (CO).

from the 3rd century BCE and fell out of use by the 5th century CE (Salomon, 1996; Strauch, 2012; Fedorova, 2012). The main unit of linear graphemic representation in Brahmic scripts is known by its traditional Sanskrit-derived name *akṣara*. As Bright (1999) notes, it is often translated as "syllable" although it does not bear direct correspondence to a syllable of speech, but rather to an orthographic syllable. The structure, or "grammar" of an akṣara is based on the following common principles: an akṣara often consists of a consonant symbol C, by default bearing an unmarked *inherent* vowel or attached diacritic (*dependent*) vowel sign v (C^v); but it may also be an *independent* vowel symbol V, or a consonant symbol with its inherent vowel "muted" by a special *virama* diacritic $\emptyset$ ($C^\emptyset$). In any of these preceding scenarios, the base consonant C can be replaced by a consonant cluster where all but the last consonant lose their inherent vowel. When the individual component consonants of the cluster combine to form a composite form, precluding the use of an overt *virama* diacritic, this is known as a "consonant conjunct" (e.g., $C_i^\emptyset C_j^\emptyset C_k$ vs $[C_iC_jC_k]$[3]) (Fedorova, 2013; Bright, 1999; Coulmas, 1999; Share and Daniels, 2016).

The elements of the akṣara grammar described above can be grouped into several natural classes. The sizes of the core classes are shown in Table 1 for each writing system and its corresponding ISO 15924 identifier in uppercase format (ISO, 2004). The major classes are the independent vowels (e.g., the Devanagari diphthong औ), the dependent vowel diacritics (e.g., the Gujarati ◌ી), and the consonants (e.g., the Gurmukhi ਞ). Another important class consists of the coda consonant symbols, like *anusvara*, *chandrabindu*, and *visarga*, which modify the akṣara as a whole (and follow and vowel signs in the memory representation). Finally, there is a class of special characters, such as the religious symbol *Om* ॐ, that behave like independent akṣara.[4]

Visual	Legacy sequence	NFC normalized
ऩ	`NA NUKTA (U+0928 U+093C)`	`NNNA (U+0929)`
क़	`QA (U+0958)`	`KA NUKTA (U+0915 U+093C)`

Table 2: NFC examples for Devanagari.

Unicode Normalization Unicode defines several *normalization forms* which are used for checking whether the two Unicode strings are equivalent to each other (Unicode Consortium, 2019). In our library we support Normalization Form C (NFC) which is well suited for comparing visually identical strings. This normalization generally converts strings to the equivalent form that uses composite characters. Table 2 shows two examples of legacy sequences corresponding canonically equivalent forms for Devanagari.

Visual Normalization As was mentioned above, an akṣara may be represented by multiple Unicode character sequences and the goal of NFC normalization is to convert them to their unique canonical form. However, there are many Unicode character sequences that fall outside the scope of NFC algorithm. We provide visual normalization that, in addition to providing the NFC functionality, also supports transforming such legacy sequences. Some of the rules are provided as "Do Not Use" tables by the Unicode Consortium (2019) that recommends transformations from legacy sequences to their corresponding canonical form, such as Devanagari { अ (`U+0905`), ॅ (`U+0945`) } $\rightarrow$ ॲ (`U+0972`). We also included transformations for visually identical sequences (under many implementations) which are commonly found on the Web, such as Devanagari { ऐ (`U+0910`), े (`U+0947`) } $\rightarrow$ ऐ (`U+0910`).[5]

Well-formedness Check A well-formedness acceptor verifies whether the given text is readable in a particular script or not. It would be hard for the native reader to visually parse the text if the script rules are not followed. For example, the reader

[3]Here, surrounding the consonants in square brackets will serve to indicate that the enclosed consonants form a conjunct together.

[4]These classes are documented in `https://github.com/google-research/nisaba/blob/main/nisaba/brahmic/mappings.md`.

[5]Here the combining vowel sign `U+0947` does not affect the compound glyph's visual appearance hence is removed.

Script ID	Visual	Character(s)	Translit.
BENG	ৎ	KHANDA TA	⟨tʾ⟩
DEVA	इ	Non-word initial VOWEL I	⟨.i⟩
GUJR	ૐ	Religious sign OM	⟨ōṁ̐⟩
GURU	ੱ	ADDAK	⟨˖⟩
MLYM	ൻ	CHILLU N	⟨nʾ⟩
SINH	ඥ	JNYA	⟨ñǰ⟩
TAML	ஃப	VISARGA, PA	⟨f⟩

Table 3: Examples for additions to ISO 15919.

does not expect two vowels signs on a single consonant and such a thing may not even be possible to reasonably draw. Furthermore, unlike the Latin script, acronyms are not written using arbitrary letter sequences, they are formed only as a sequence of akṣara. Our approach verifies whether the text is a sequence of well-formed akṣara using the grammar described above.

Reversible ISO Transliteration ISO 15919 represents a unified 8-bit Latin transliteration scheme for major South Asian Brahmic scripts (ISO, 2001). Since it has not been updated with the characters that were introduced to the Unicode standard after 2001, we have added additional mappings, with some examples shown in Table 3. These additions are crucial because they allow us to reverse the romanizations to get the original Brahmic strings back reliably. This property allows various data processing pipelines to use the romanized text as an internal representation and convert it back to the original native script at the output stage.

Language-specific Logic Several South Asian languages often share the same script with some, often minor, language-specific differences. Our library supports language-specific customizations that can be combined with language-agnostic script logic. For example, the modern Bengali–Assamese script (Beng) is shared by both Bengali and Assamese languages, among others (Brandt and Sohoni, 2018). For both of these languages our library provides customizations,[6] such as the transformations required for visual normalization of Assamese that transform Bengali letter *ra* into its Assamese equivalent when it participates in a consonant conjunct (which generally occurs when following or preceding *virama*, e.g., { র (U+09B0), ্ (U+09CD) } $\rightarrow$ { ৰ (U+09F0), ্ (U+09CD) }).

[6] https://github.com/google-research/nisaba/tree/main/nisaba/brahmic/data/lang

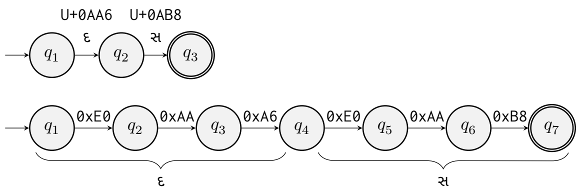

Figure 1: String acceptors for Gujarati word દસ (⟨dasa⟩, “ten”) over an alphabet of Unicode code points (top) and bytes (bottom).

```
Require: FSAs: consonant, vowel, vowel_sign, coda, standalone, virama,
    dead_consonant, accept.
1: function $\mathcal{W}$(consonant, vowel, vowel_sign, coda, standalone, virama,
       dead_consonant, accept)
2:     $cluster \leftarrow (consonant + virama)^* + consonant$
3:     $codable \leftarrow (vowel \cup (cluster + vowel\_sign^?) \cup accept) \cup coda^?$
4:     $akshara \leftarrow codable \cup (cluster + virama + dead\_consonant^?)$
5:     $T \leftarrow akshara \cup standalone$
6:     return $T^+$                                        ▷ Kleene plus
```

Figure 2: Simplified construction of the well-formed automaton $\mathcal{W}$.

4 The Finite-State Approach

The Brahmic script manipulation operations described above have a natural intepretation grounded in formal language theory. We treat the text corpus in a given script as a set of strings over some finite alphabet Σ that defines a set of admissable script symbols. The set of zero or more strings is known as *language* which, in its simplest (regular) form, can be succintly described (or *recognized*) by a finite state automaton (FSA) or acceptor (Yu, 1997). Two simple FSAs that represent the Gujarati word દસ are shown in Figure 1, where the top automaton represents the word over an alphabet of Unicode code points for Gujarati, while the bottom one represents the same string over the corresponding byte symbols in UTF-8 encoding (Unicode Consortium, 2019). Our library supports both representations.

The akṣara grammar outlined in the previous section can be expressed via elementary formal operations on the FSAs that describe grammar constituents. Such set-theoretic operations include union ($\cup$), concatenation ($+$) and closure, where closure is defined as an arbitrary natural number of concatenations of a language L over Σ with itself, either accepting an empty input $\{\epsilon\}$ or not, denoted L^* (Kleene star) and L^+ (Kleene plus), respectively (Kuich and Salomaa, 1986). These operations represent non-trivial automata which are compiled offline resulting in compact and efficient representations. A simplified process for constructing the automaton $\mathcal{W}$ to perform the well-

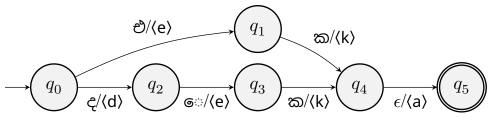

Figure 3: Romanization of Sinhala words එක (“one”) and දෙක (“two”) into ⟨eka⟩ and ⟨deka⟩, respectively.

formed check from the previous section is shown in Figure 2. In this simplified example, the paths through the automaton that define a legal consonant cluster (line 2 of the algorithm) are represented by a sub-automaton that recognizes the language that consists of strings formed from the consonant and virama symbols only, where each consonant, apart from the last one, must be followed by the virama that removes an inherent vowel.

The rest of the operations on the Brahmic scripts, namely the normalization and transliteration, involve modifications of the Brahmic script inputs. Such operations are naturally expressed by finite state transducers (FSTs), which are a generalization of the FSA concept used to encode string-string *relations* (or transductions), by modifying the automata arcs to have pairs of labels from input and output alphabets, instead of single labels. A trivial romanization in our representation of the two Sinhala words එක (⟨eka⟩, “one”) and දෙක (⟨deka⟩, “two”) is shown in Figure 3. Note the “vocalization” of the final consonant by insertion of a schwa via an input ϵ-transition. Also note that the path accepting the second word is longer. The word දෙක consists of three akṣara and requires modification of the inherent vowel by the dependent vowel in order to produce ⟨de⟩.

The basic operations on the FSAs outlined above also extend to the FST case and allow for similarly succinct final compiled representations (Mohri, 2000), such as the simplified construction of the ISO romanization transducer $\mathcal{I}$ for converting from Brahmic scripts to Latin alphabet, shown in Figure 4. An important extension of FSAs and FSTs are the weighted finite state automata (WFSAs) and transducers (WFSTs) (Mohri, 2004, 2009) that equip each arc in the automaton or transducer with a weight, thus allowing optimization and search algorithms to compute the costs of distinct paths, which can be used to determine their relative importance. We use weights in some of our grammars to indicate the relative priority of a particular akṣara modification. For example, in Figure 4, the paths corresponding to consonants followed by dependent vowels (line 6) have priority

```
Require: FSTs: consonant, vowel, vowel_sign, coda, standalone, virama.

 1: function $\mathcal{I}$(consonant, vowel, vowel_sign, coda, standalone, virama)
 2:     del_virama ← virama × Ø                                   ▷ Delete virama
 3:     ins_schwa ← Ø × {⟨a⟩}                                     ▷ Insert inherent vowel
 4:     deweight ← ($\epsilon, \epsilon, w \downarrow$)           ▷ De-prioritize the path
 5:     T ← (
 6:         (consonant + vowel_sign) ∪      ▷ (ස,⟨sa⟩) + (◌ු,⟨u⟩) → (සු,⟨su⟩)
 7:         (consonant + del_virama + deweight) ∪
 8:         (consonant + ins_schwa + deweight) ∪
 9:         (vowel + deweight) ∪ coda ∪ standalone ∪
10:         ... )                                                 ▷ Further logic
11:     return $T^*$                                              ▷ Kleene star
```

Figure 4: Simplified construction of the transliteration transducer $\mathcal{I}$.

over the akṣara-initial independent vowels (line 9).

The two remaining operations on akṣara, namely NFC and visual normalization, are represented in our library using the context-dependent rewrite rules from the formal approach popularized by Chomsky and Halle (1968). The normalization rules are represented as a sequence $\{\phi \rightarrow \psi / \lambda \_\_ \rho\}$, where the source ϕ is rewritten as ψ if its left and right contexts are λ and ρ. For an earlier example from §3, a single NFC normalization rule rewrites the Devanagari string ϕ = “न” (*na*, `U+0928`) + “़” (*nukta* sign, `U+093C`) into its canonical composition ψ = “ऩ” (*nnna*, `U+0929`). Kaplan and Kay (1994) proposed an algorithm for compiling such sequences into an FST. This approach was further improved and extended to WFSTs by Mohri and Sproat (1996), whose algorithm we use to compile sequences of NFC and visual normalization rules into transducers denoted $\mathcal{N}$ and $\mathcal{V}$.

Finally, the transducers representing language-specific customizations of a particular script operation are compiled by composing the generic language-agnostic transducer, such as the Devanagari visual normalizer, with the transducer representing transformations that capture language-specific use of the script, e.g., Devanagari for Nepali.

5 System Details and Demo

The core of the Nisaba Brahmic script manipulation library resides under the `brahmic` directory of the distribution. In this section we provide details for how to build and use the library and also explore its application to visual normalization of Wikipedia-based text in 9 of these scripts.

Prerequisites We use Bazel (Google, 2020) as a primary build environment. For compiling the

Op.	Symb.	Prop.	Script									
			BENG	DEVA	GUJR	GURU	KNDA	MLYM	ORYA	SINH	TAML	TELU
$\mathcal{I}$	Unicode	N_s	127	130	113	93	119	122	105	122	75	112
		N_a	475	546	476	418	487	522	452	513	326	485
	Byte	N_s	248	235	195	171	210	201	178	192	126	181
		N_a	384	399	334	288	350	345	305	339	229	318
$\mathcal{N}$	Unicode	N_s	9	17	1	8	21	8	9	17	11	4
		N_a	158	248	75	78	349	261	160	352	228	163
	Byte	N_s	31	55	1	28	70	27	31	55	37	14
		N_a	1,812	1,841	255	1,047	2,884	2,322	1,813	2,611	3,098	1,543
$\mathcal{V}$	Unicode	N_s	103	51,710	98	119	1764	287	60	182	209	57
		N_a	2,423	121,157	2,234	2,322	6,136	3,021	1,732	2,129	1,280	2,249
	Byte	N_s	369	165,168	356	425	5,611	965	232	624	703	225
		N_a	18,896	266,441	18,684	20,733	30,422	18,598	16,146	15,363	11,830	18,717
$\mathcal{W}$	Unicode	N_s	11	7	7	7	10	10	7	7	4	6
		N_a	427	446	388	341	465	485	380	361	158	335
	Byte	N_s	38	23	21	23	33	33	22	22	11	19
		N_a	297	321	284	257	309	297	279	195	130	239

Table 4: Properties of script FSTs arranged by operation and symbol types (Unicode code points and UTF-8 bytes), where $\mathcal{I}$ denotes the ISO transliteration operation, $\mathcal{N}$ is the NFC normalization, $\mathcal{V}$ denotes visual normalization, and $\mathcal{W}$ is the well-formed check. The numbers of states and arcs are denoted by N_s and N_a, respectively.

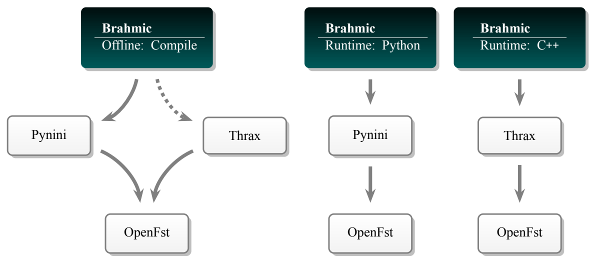

Figure 5: Software dependency diagrams for the three modes of operation: compile stage (left), Python run-time (center) and C++ run-time (right).

automata and transducers we employ Pynini[7], a Python library for constructing finite-state grammars and for performing operations on WFSTs (Gorman, 2016; Gorman and Sproat, in press). In addition, the library depends on Thrax[8], an older relative of Pynini, that provides a custom grammar manipulation language for WFSTs (Tai et al., 2011; Roark et al., 2012). Although Thrax has been mostly superseded by Pynini, we still rely on some of its utilities for unit testing and its C++ run-time components. At their core, both Pynini and Thrax depend on the OpenFst library[9] for the implementation of most WFST algorithms (Allauzen et al., 2007; Riley et al., 2009). The overall dependency diagram is shown on the left-hand side of Figure 5 (the minimal dependency on Thrax is indicated by a dotted arrow). At build time, Bazel pulls in these dependencies remotely from their respective repositories.

[7]http://pynini.opengrm.org/
[8]http://thrax.opengrm.org
[9]http://www.openfst.org

Compiling the Transducers Figure 6 presents the sequence of steps to compile the transducers, including downloading the repository (line 2), compiling the library and its artifacts (line 5) and running the unit tests (line 7). The artifacts are compiled by Bazel using Pynini and consist of the finite state archive (FAR) files that contain collections of WFSTs (Roark et al., 2012). For each of the four Brahmic script operations we generate two FAR files: one for WFSTs over the byte alphabet, and another over the Unicode code point alphabet.[10] Each FAR file contains ten script-specific transducers whose names correspond to the upper-case ISO 15924 script codes. Since the transliteration operation is bidirectional, the name of each script-specific transliteration transducer has the prefix `FROM_` for the native-to-Latin direction, and `TO_` for the inverse. The numbers of states (N_s) and arcs (N_a) of the resulting transliteration ($\mathcal{I}$), NFC ($\mathcal{N}$), visual normalization ($\mathcal{V}$) transducers and well-formedness acceptors ($\mathcal{W}$) for each script and alphabet type are shown in Table 4.

Offline and Online Usage Once the transducers are compiled, they can be applied offline to the input files using the `rewrite-tester` tool provided by Thrax, as shown in lines 8–13 of the example in Figure 6, where the visual normalization transducer $\mathcal{V}$ for Kannada that resides in the `visual_norm.far` archive is applied to words in input file `words.txt`.

We provide lightweight run-time interfaces for

[10]The Unicode code point FARs rather misleadingly have the suffix `utf8` in their name for historical reasons.

```
# Download Nisaba repository.
git clone https://github.com/google-research/nisaba.git
cd nisaba
# Compile the transducers and tests.
bazel build -c opt //nisaba/brahmic/...
# Run the unit tests.
bazel test -c opt //nisaba/brahmic/...
# Compile Thrax rewrite helper tool.
bazel build -c opt @org_opengrm_thrax//:rewrite-tester
# Run visual normalization for Kannada.
bazel-bin/external/org_opengrm_thrax/rewrite-tester \
   --far=bazel-bin/nisaba/brahmic/visual_norm.far \
   --rules=KNDA < words.txt
```

Figure 6: Compiling the transducers.

```python
import unittest
from nisaba import brahmic

class BrahmicTest(unittest.TestCase):
  def testBasicOperations(self):
    # Check romanization.
    iso_to_deva = brahmic.IsoTo('Deva')
    self.assertEqual('क्लब',
       iso_to_deva.ApplyOnText('⟨k·laba⟩'))
    # Check valid inputs.
    wellformed_mlym = brahmic.WellFormed('Mlym')
    self.assertTrue(wellformed_mlym.AcceptText('സ്വരം'))
    # Visual normalizer.
    visual_norm_deva = brahmic.VisualNorm('Deva')
    self.assertEqual('औ', visual_norm_deva.ApplyOnText('अौ'))
```

Figure 7: Run-time Python interface example.

both Python and C++, their dependencies shown in the center and the right-hand side of Figure 5, respectively. The Python interface is provided via several wrappers around the `pynini.Fst` abstraction, with a simple example shown in Figure 7. In addition to performing simple operations on individual strings, more WFST-specific operations, such as transducer composition, are provided by Pynini. The C++ interface is provided by the `Grammar` helper class, shown in Figure 8, that includes the necessary methods for initializing the WFSTs and performing rewrites (for transducers) and acceptance tests (for acceptors). In addition, many more operations on WFSTs are available through the OpenFst library, if required.

Prevalence of Normalization To demonstrate the prevalence of text requiring normalization in these scripts, we normalized publicly available corpora and measured how frequently words in the samples were modified. The Dakshina dataset (Roark et al., 2020) includes (among other things) collections of monolingual Wikipedia sentences in 12 South Asian languages, 10 of which use Brahmic scripts. We applied visual normalization to the training partitions of the collections in these 10 languages, and Table 5 presents the percentage of both types and tokens that were changed by the normalization.[11] Malayalam is the language with the highest percentage of both types and tokens changed by visual normalization, largely due to frequent conversion to chillu letters from alternative encodings. For example, the relatively frequent word തന്റെ ("yours") is normalized to the encoding with the chillu letter ൻ instead of ന.

```cpp
#include <string>

// Generic wrapper around FST archive with Brahmic transducers.
class Grammar {
  public:
  // Constructs given the FAR path, its name and the name of WFST.
  Grammar(const std::string& far_path, const std::string& far_name,
          const std::string& fst_name);
  // Initializes the transducer.
  bool Load();
  // Rewrites <input> into <output>.
  bool Rewrite(const std::string& input, std::string *output) const;
  // Checks whether the grammar accepts <input>.
  bool Accept(const std::string& input) const;
};
```

Figure 8: Run-time C++ interface.

Language	Script	% Changed Types	% Changed Tokens
Bengali	BENG	0.53	0.06
Gujarati	GUJR	0.46	0.09
Hindi	DEVA	1.41	0.18
Kannada	KNDA	4.19	1.66
Malayalam	MLYM	6.33	4.19
Marathi	DEVA	1.51	0.40
Punjabi	GURU	1.67	0.33
Sinhala	SINH	3.55	0.71
Tamil	TAML	0.59	0.17
Telugu	TELU	1.97	0.63

Table 5: Percentage of types and tokens changed by visual normalization from native script Wikipedia training partitions of the Dakshina dataset.

6 Conclusion and Future Work

We presented finite-state automata-based utilities for processing the major Brahmic scripts. The finite state transducer formalism provides an efficient and scalable framework for expressing Brahmic script operations and is suitable for many NLP applications, such as those reported in Kumar et al. (2020) and Kakwani et al. (2020), which may benefit from the reduction in "noise" present in unnormalized text. In the future, we will continue to improve the support for existing scripts and extend our work to other Brahmic scripts.

[11]Tokenization was simply based on whitespace, with no other processing such as punctuation separation, so the total number of distinct types is accordingly relatively high. The texts from that dataset were already NFC normalized.

Acknowledgments

The authors would like to thank Işın Demirşahin for valuable discussion on this project.

References

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. OpenFst: A general and efficient weighted finite-state transducer library. In *International Conference on Implementation and Application of Automata*, pages 11–23. Springer.

Chantal Amrhein and Rico Sennrich. 2020. On Romanization for model transfer between scripts in neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2461–2469, Online. Association for Computational Linguistics.

Pushpak Bhattacharyya, Hema Murthy, Surangika Ranathunga, and Ranjiva Munasinghe. 2019. Indic language computing. *Communications of the ACM*, 62(11):70–75.

Carmen Brandt and Pushkar Sohoni. 2018. Script and identity – the politics of writing in South Asia: an introduction. *South Asian History and Culture*, 9(1):1–15.

William Bright. 1999. A matter of typology: Alphasyllabaries and abugidas. *Written Language & Literacy*, 2(1):45–55.

Ronald Cardenas, Ying Lin, Heng Ji, and Jonathan May. 2019. A grounded unsupervised universal part-of-speech tagger for low-resource languages. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 2428–2439, Minneapolis, Minnesota. Association for Computational Linguistics.

George Cardona and Danesh Jain. 2007. *The Indo-Aryan Languages*. Routledge Language Family Series. Routledge, New York.

Noam Chomsky and Morris Halle. 1968. *The Sound Pattern of English*. Harper & Row, New York.

Florian Coulmas. 1999. *The Blackwell Encyclopedia of Writing Systems*. John Wiley & Sons, Oxford.

A. K. Datta. 1984. A generalized formal approach for description and analysis of major Indian scripts. *IETE Journal of Research*, 30(6):155–161.

Liudmila L Fedorova. 2012. The development of structural characteristics of Brahmi script in derivative writing systems. *Written Language & Literacy*, 15(1):1–25.

Liudmila L. Fedorova. 2013. The development of graphic representation in abugida writing: The akshara's grammar. *Lingua Posnaniensis*, 55(2):49–66.

Google. 2020. Bazel. `http://bazel.build`. [Online], Accessed: 2020-12-10.

Kyle Gorman. 2016. Pynini: A Python library for weighted finite-state grammar compilation. In *Proceedings of the SIGFSM Workshop on Statistical NLP and Weighted Automata*, pages 75–80, Berlin, Germany. Association for Computational Linguistics.

Kyle Gorman and Richard Sproat. in press. *Finite-State Text Processing*. Human Language Technologies. Morgan & Claypool, Williston, VT.

Bradley Hauer, Amir Ahmad Habibi, Yixing Luan, Rashed Rubby Riyadh, and Grzegorz Kondrak. 2019. Cognate projection for low-resource inflection generation. In *Proceedings of the 16th Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 6–11, Florence, Italy. Association for Computational Linguistics.

Lars Hellsten, Brian Roark, Prasoon Goyal, Cyril Allauzen, Françoise Beaufays, Tom Ouyang, Michael Riley, and David Rybach. 2017. Transliterated mobile keyboard input via weighted finite-state transducers. In *Proceedings of the 13th International Conference on Finite State Methods and Natural Language Processing (FSMNLP 2017)*, pages 10–19, Umeå, Sweden. Association for Computational Linguistics.

Ulf Hermjakob, Jonathan May, and Kevin Knight. 2018. Out-of-the-box universal Romanization tool uroman. In *Proceedings of ACL 2018, System Demonstrations*, pages 13–18, Melbourne, Australia. Association for Computational Linguistics.

Lauren Hinkle, Albert Brouillette, Sujay Jayakar, Leigh Gathings, Miguel Lezcano, and Jugal Kalita. 2013. Design and evaluation of soft keyboards for Brahmic scripts. *ACM Transactions on Asian Language Information Processing (TALIP)*, 12(2):1–37.

Xiaolei Huang, Jonathan May, and Nanyun Peng. 2019. What matters for neural cross-lingual named entity recognition: An empirical analysis. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 6395–6401, Hong Kong, China. Association for Computational Linguistics.

ISO. 2001. ISO 15919: Transliteration of Devanagari and related Indic scripts into Latin characters. `https://www.iso.org/standard/28333.html`. International Organization for Standardization.

ISO. 2004. ISO 15924: Codes for the representation of names of scripts. `https://www.iso.org/obp/ui/#iso:std:iso:15924:ed-1:v1:en`. International Organization for Standardization.

Divyanshu Kakwani, Anoop Kunchukuttan, Satish Golla, Gokul N. C., Avik Bhattacharyya, Mitesh M. Khapra, and Pratyush Kumar. 2020. iNLPSuite: Monolingual corpora, evaluation benchmarks and pre-trained multilingual language models for Indian languages. In *Proc. of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020*, pages 4948–4961, Online Event. Association for Computational Linguistics.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378.

Werner Kuich and Arto Salomaa. 1986. *Semirings, Automata, Languages*, volume 5 of *Monographs in Theoretical Computer Science*. Springer, Berlin.

Saurav Kumar, Saunack Kumar, Diptesh Kanojia, and Pushpak Bhattacharyya. 2020. "A passage to India": Pre-trained word embeddings for Indian languages. In *Proc. of the 1st Joint Workshop on Spoken Language Technologies for Under-resourced languages (SLTU) and Collaboration and Computing for Under-Resourced Languages (CCURL)*, pages 352–357, Marseille, France. European Language Resources association.

Anoop Kunchukuttan. 2020. The IndicNLP Library. `https://github.com/anoopkunchukuttan/indic_nlp_library`.

Anoop Kunchukuttan, Ratish Puduppully, and Pushpak Bhattacharyya. 2015. Brahmi-net: A transliteration and script conversion system for languages of the Indian subcontinent. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Demonstrations*, pages 81–85, Denver, Colorado. Association for Computational Linguistics.

Mehryar Mohri. 2000. Minimization algorithms for sequential transducers. *Theoretical Computer Science*, 234(1-2):177–201.

Mehryar Mohri. 2004. Weighted finite-state transducer algorithms. An overview. In Carlos Martín-Vide, Victor Mitrana, and Gheorghe Păun, editors, *Formal Languages and Applications*, pages 551–563. Springer, Berlin; Heidelberg.

Mehryar Mohri. 2009. Weighted automata algorithms. In Manfred Droste, Werner Kuich, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Monographs in Theoretical Computer Science, pages 213–254. Springer.

Mehryar Mohri and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238, Santa Cruz, California, USA. Association for Computational Linguistics.

Nikitha Murikinati, Antonios Anastasopoulos, and Graham Neubig. 2020. Transliteration for cross-lingual morphological inflection. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 189–197, Online. Association for Computational Linguistics.

Tom Ouyang, David Rybach, Françoise Beaufays, and Michael Riley. 2017. Mobile keyboard input decoding with finite-state transducers.

Umapada Pal, Ramachandran Jayadevan, and Nabin Sharma. 2012. Handwriting recognition in Indian regional scripts: A survey of offline techniques. *ACM Transactions on Asian Language Information Processing (TALIP)*, 11(1):1–35.

Vinodh Rajan. 2020. Aksharamukha. `https://github.com/virtualvinodh/aksharamukha`.

Michael Riley, Cyril Allauzen, and Martin Jansche. 2009. OpenFst: An open-source, weighted finite-state transducer library and its applications to speech and language. In *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*, pages 9–10, Boulder, Colorado. Association for Computational Linguistics.

Brian Roark, Richard Sproat, Cyril Allauzen, Michael Riley, Jeffrey Sorensen, and Terry Tai. 2012. The OpenGrm open-source finite-state grammar software libraries. In *Proceedings of the ACL 2012 System Demonstrations*, pages 61–66, Jeju Island, Korea. Association for Computational Linguistics.

Brian Roark, Lawrence Wolf-Sonkin, Christo Kirov, Sabrina J. Mielke, Cibu Johny, Isin Demirsahin, and Keith Hall. 2020. Processing South Asian languages written in the Latin script: the Dakshina dataset. In *Proc. of 12th Language Resources and Evaluation Conference (LREC)*, pages 2413–2423, Marseille, France.

Richard G. Salomon. 1996. Brahmi and Kharoshthi. In Peter T. Daniels and William Bright, editors, *The World's Writing Systems*, pages 373–383. Oxford University Press, New York, NY.

Royal Denzil Sequiera, Shashank S. Rao, and B. R. Shambavi. 2014. Word-level language identification and back transliteration of romanized text. In *Proceedings of the Forum for Information Retrieval Evaluation*, pages 70–73, Bangalore, India.

David L. Share and Peter T. Daniels. 2016. Aksharas, alphasyllabaries, abugidas, alphabets and orthographic depth: Reflections on Rimzhim, Katz and Fowler (2014). *Writing systems research*, 8(1):17–31.

R. Mahesh K. Sinha. 2009. A journey from Indian scripts processing to Indian language processing. *IEEE Annals of the History of Computing*, 31(1):8–31.

Richard Sproat. 2003. A formal computational analysis of Indic scripts. In *In International Symposium on Indic Scripts: Past and Future*, Tokyo, Japan.

Sanford B. Steever. 2019. *The Dravidian Languages*, 2nd edition. Routledge Language Family Series. Routledge, New York.

Ingo Strauch. 2012. The character of the Indian Kharoṣṭhī script and the "Sanskrit Revolution": A writing system between identity and assimilation. In Alexander J. de Voogt and Joachim Friedrich Quack, editors, *The Idea of Writing: Writing Across Borders*, pages 131–168. Brill, Leiden; Boston.

Terry Tai, Wojciech Skut, and Richard Sproat. 2011. Thrax: An open source grammar compiler built on OpenFst. In *Proc. of IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, volume 12, Hawaii, USA.

Unicode Consortium. 2019. The Unicode Standard. Online: `http://www.unicode.org/versions/Unicode12.1.0/`. Version 12.1.0, Mountain View, CA.

Hans H. Wellisch. 1978. *The Conversion of Scripts: Its Nature, History, and Utilization*. Information sciences series. John Wiley & Sons, New York.

Sheng Yu. 1997. Regular languages. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1: Word, Language, Grammar, pages 41–110. Springer, Berlin.

Hao Zhang, Richard Sproat, Axel H Ng, Felix Stahlberg, Xiaochang Peng, Kyle Gorman, and Brian Roark. 2019. Neural models of text normalization for speech applications. *Computational Linguistics*, 45(2):293–337.

Yuhao Zhang, Ziyang Wang, Runzhe Cao, Binghao Wei, Weiqiao Shan, Shuhan Zhou, Abudurexiti Reheman, Tao Zhou, Xin Zeng, Laohu Wang, Yongyu Mu, Jingnan Zhang, Xiaoqian Liu, Xuanjun Zhou, Yinqiao Li, Bei Li, Tong Xiao, and Jingbo Zhu. 2020. The NiuTrans machine translation systems for WMT20. In *Proceedings of the Fifth Conference on Machine Translation*, pages 338–345, Online. Association for Computational Linguistics.