Computer-Assisted Language Comparison in Practice

Tutorials on Computational Approaches to the History and Diversity of Languages

Sequence Manipulation with Orthography Profiles in JavaScript

Johann-Mattis List
DLCE / Chair of Multilingual Computational Linguistics
MPI-EVA / University of Passau

Orthography profiles allow for the explicit simultaneous segmentation and conversion of sequences from one orthography to another. They play a crucial role in the standardization workflows developed as part of the Cross-Linguistic Data Formats initiative, where they are used to convert original orthographies used for language documentation to a strict version of the International Phonetic Alphabet. Given that the basic algorithm by which orthography profiles can be used to segment and convert sequences across orthographies is very straightforward, it can be easily implemented in JavaScript.

1 Introduction

Orthography profiles (Moran and Cysouw 2018) are lookup and replacement tables that can be used to convert sequences from one alphabet to another one. The conversion is explicit and proceeds in two steps. Based on a replacement table that defines graphemes in the source orthography (a grapheme representing a distinctive unit that consists of one or more individual characters) and their counterparts in the target orthography, a sequence is first segmented into graphemes, using a greedy strategy. In a second step, the graphemes in the source orthography are replaced by their counterparts in the target orthography.

Since they are straightforward to use and easy to apply computationally, orthography profiles have come to play a major role in the standardization of multilingual wordlists to Cross-Linguistic Data Formats (Forkel et al. 2018). Their role was specifically important for the creation of the Lexibank repository (List et al. 2022), which uses the Python implementation by Moran et al. (2022) in order to represent the orthographies underlying individual wordlist collections by the standardized version of the International Phonetic Alphabet proposed in the Cross-Linguistic Transcription Systems (CLTS) reference catalog (Anderson et al. 2022, https://clts.clld.org).

Computer-Assisted Language Comparison in Practice
Volume 6, Number 2,65–72
URL: https://calc.hypotheses.org/6361/ DOI: 10.15475/calcip.2023.2.3

Published on 27/09/2023

Given that the algorithm for orthography profile creation is very straightforward, I have been experimenting for quite some time with JavaScript implementations, hoping that I could use them in interactive web applications. I have recently started to revise my old code and since the JavaScript implementation is rather short, I thought it might be useful to share the full implementation along with comments on the reasoning behind it.

2 The Algorithm for Orthography Profile Conversion

To give a simple example for the way an orthography profile segments and converts sequences, consider the orthography profile in Table 1 below, which defines a source orthography consisting of 12 graphemes in the column Grapheme along with the replacements given in the column IPA. While half of the graphemes consist of only one character (a, e, i, p, t, and k), three graphemes consist of two ASCII characters (ph, th, and kh), and three graphemes consist of two characters when applying Unicode NFD normalization (see Moran and Cysouw 2018: 17), , i.e. the graphemes ā, ē, and ī can be represented by a, e, and i combined with the the macron character ◌̄.

Grapheme	IPA
ā	aː
ē	eː
ī	iː
a	a
e	e
i	i
p	p
t	t
k	k
ph	p^h
th	t^h
kh	k^h

Table 1: A simple orthography profile for illustration purposes.

This orthography profile can be used to segment a sequence such as "khapā" into its graphemes "kh a p ā". If graphmes have been identified, it is straightforward to convert the sequence to its target sequence "k^h a p aː" in the International Phonetic Alphabet. The algorithm defined by Moran and Cysouw (2018) proceeds in a greedy fashion from the left to the right. It starts by checking the longest potential grapheme, which would be the whole word, and then checks if this segment occurs in the orthography profile. If this is not the case, the next smaller segment is tested, which would be "khapa" in this case (stripping of the macron from the sequence), followed by "khap", "kha", and finally "kh", the first segment that we can indeed find in the orthography profile. Once such a segment that occurs in the profile has been found, the algorithm takes the remaining

sequence and proceeds in the same way (via "apā", "apa", "ap", "a"), and so on, until all graphemes have been identified.

If single character units cannot be found in the orthography profile, they are treated as an unknown grapheme. Thus, "khaupa" would be parsed as "kh a u p a", but the u could not be converted to any existing character, which could, for example, be marked by a question mark in the conversion process, resulting in "kh a ? p a".

3 Implementing Orthography Profiles in JavaScript

In order to implement orthography profiles in JavaScript (or any other programming language), one should first decide on the basic format used to represent orthography profiles. Since orthography profiles are "lookup tables", that is, tables, where one needs to look up individual segments, the easiest way to represent them in a script is to use a hash table, better known as a dictionary in Python and JavaScript, as a data type. The orthography profile shown in Table 1 can be represented in the form of a dictionary in which the entries in the first column are used as a key and each row is represented itself as a dictionary with column names as keys and cell content as values.

```
var profile = {
  "ā": {"Grapheme": "ā", "IPA": "aː"},
  "ē": {"Grapheme": "ē", "IPA": "eː"},
  "ī": {"Grapheme": "ī", "IPA": "iː"},
  "a": {"Grapheme": "a", "IPA": "a"},
  "e": {"Grapheme": "e", "IPA": "e"},
  "i": {"Grapheme": "i", "IPA": "i"},
  "p": {"Grapheme": "p", "IPA": "p"},
  "t": {"Grapheme": "t", "IPA": "t"},
  "k": {"Grapheme": "k", "IPA": "k"},
  "ph": {"Grapheme": "ph", "IPA": "pʰ"},
  "th": {"Grapheme": "th", "IPA": "tʰ"},
  "kh": {"Grapheme": "kh", "IPA": "kʰ"}
};
```

One may criticize this representation for being redundant, since the content of the Grapheme column is repeated as key and as value, but the format has the advantage that we can check if a graphem occurs in the orthography profile by using JavaScripts in operator, while we can at the same time use the profile to convert from the original grapheme to the target orthography by specifying the column name.

For the implementation, we follow the two-stage procedure outlined above, which means that we define two functions, one that segments a sequence with an orthography profile and one that converts a segmented sequence into another sequence with the help of the orthography profile.

The first function, which I call segmentize takes as input the original sequence (represented as a string) and the profile (represented as shown above). Before segmenting the sequence, we make sure that the input string is normalized according to Unicode NFD normalization, and we return the empty string inside an array if the word itself is the empty string.

```
function segmentize(word, profile) {
  word = word.normalize('NFD');
  if (word.length == 0) {
    return [word];
  }
...
}
```

We now define basic variables that we will use throughout the function. First, there is a queue that contains lists of three objects, the part that has been segmented at any stage in the parsing process, the current segment that one tests against the orthography profile, and the remaining rest of the sequence. Results are stored in the list called segments, and the three elements of each item in the queue are called segmented, current, and rest.

```
function segmentize(word, profile) {
  ...
  /* define queue and output */
   var queue = [[[], word, '']];
  /* define variables */
  var segmented, current, rest;
  ...
}
```

We can now start with the main loop, which we implement as a while loop that stops when the queue is empty. Upon each loop, we retrieve the already segmented parts, the current suggestion for grapheme segmentation, and the rest of the sequence.

```
function segmentize(word, profile) {
   ...
  /* main loop */
  while (queue.length > 0) {
    [segmented, current, rest] = queue.splice(0, 1)[0];
    ...
  }
}
```

There are now four different conditions that we need to check. First, the “current” part occurs in the profile and the rest is empty. In this case, we append the segment represented by the variable current to the list of graphemes and the loop stops.

```
function segmentize(word, profile) {
  ...
  while (queue.length > 0) {
    ...
    /* A: check if the current segment occurs in the profile */
    if (current in profile &amp;amp;&amp;amp; !(rest)) {
      return segmented.concat([current]);
    }
  ...
  }
}
```

Second, the current grapheme is of length one (and can therefore no longer shortened) and does not occur in the profile. In this case, we check if the rest is not empty, and if this is the case, we proceed, taking the rest as our current segment and adding the current grapheme to the list of segmented graphemes. If the rest is empty, we add the grapheme to the list of segmented graphemes and return the list.

```
function segmentize(word, profile) {
  ...
  while (queue.length > 0) {
    ...
    /* B: check for non-specified characters in profile */
    else if (current.length == 1 &amp;amp;&amp;amp; !(current in profile)) {
      if (rest) {
        queue.push([segmented.concat([current]), rest, '']);
      }
      else {
        return segmented.concat([current]);
      }
    }
    ...
  }
}
```

Third, if the current character combination does not recur in the orthography profile, we decrease it, by dropping the last character and assigning it to the remaining characers in the queue.

```
function segmentize(word, profile) {
  ...
  while (queue.length > 0) {
    ...
    /* C: check for next smaller string */
    else if (!(current in profile)) {
      queue.push(
        [
          segmented,
          current.slice(0, current.length-1),
          current[current.length-1] + rest
        ]
      );
    }
    ...
  }
}
```

Fourth, if the current character combination is a grapheme in the profile, we add it to the segmented entries and proceed with the rest of the sequence.

```
function segmentize(word, profile) {
  ...
  while (queue.length > 0) {
    ...
    /* D: if current in profile, proceed with rest */
    else {
      queue.push(
        [
          segmented.concat([current]),
          rest,
          ''
        ]
      );
    }
    ...
  }
}
```

The output that this function yields is a list of strings which correspond to the graphemes found or to individual characters that could not be identified to recur in the orthography profile.

Once the segmentation has been carried out, the conversion is extremely easy to manage, since all we have to do is to look for each grapheme if it finds a counterpart in our orthography profile and yield the segment that is specified in the respective column in our orthography profile. If segments cannot be identified, we place them in specific quotation marks «».

```
function convert(segments, profile, column){
  var output = [];
  var i;
  for (i=0; i<segments.length; i++) {
    if (segments[i] in profile) {
      output.push(profile[segments[i]][column]);
    }
    else {
      output.push('«' + segments[i] + '»');
    }
  }
  return output;
}
```

These two small functions are all that is needed in order to carry out sequence manipulation with orthography profiles in JavaScript.

4 Demo

In order to make it easier to follow my examples, I have made a very small application that uses the scripts shown above and allows to type in a sequence and check how it is segmented and converted at the same time. This application is supplemented with this study and can also be accessed in the form of a JavaScript Fiddle (see https://jsfiddle.net/LinguList/rkqcsa36/30/). The profile used in this demonstration is identical with the one shown above, but long vowels are no longer represented with a macron, but by duplicating the respective vowel instead.

By typing into the “output” field of the demo page, you can immediately see how the string will be segmented step by step and how segments will be converted into the target orthography. The advantage of this interactive representation, for which only a small function was added to the script, is that one can see how orthography profiles work “in action”. This in turn may be helpful for illustrative purposes (when teaching how to use orthography profiles) and for the purpose of debugging existing profiles.

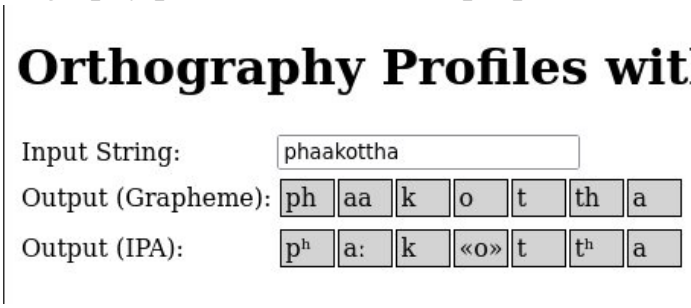

Figure 1: Orthography profile demo.

Figure 1 shows how the string "phaakottha" (that does not correspond to any word in any language but was just typed for illustrational purposes) is first segmented to "ph aa k o t th a" and then converted to "p^h a: k «o» t t^h a". Here, the grapheme o is missing from

the profile and is not converted to any target grapheme accordingly, but rather marked as something that the orthography profile does not handle.

5 Outlook

Having a JavaScript implementation of the string manipulation method using orthography profiles at one's disposal offers many additional interesting possibilities. Apart from using it for interactive software demonstrations, an extended application could be used to help in debugging orthography profiles, and the possibility to write plugins in JavaScript for Spreadsheet solutions such as Google Sheets can even make it possible to apply orthography profiles directly from spreadsheets.

The code described in this study is available online in the form of a JavaScript Fiddle that can be found under the link https://jsfiddle.net/LinguList/rkqcsa36/30/.

References

List, Johann-Mattis and Anderson, Cormac and Tresoldi, Tiago and Forkel, Robert (2021): Cross-Linguistic Transcription Systems. Version 2.1.0. Jena:Max Planck Institute for the Science of Human History. https://clts.clld.org

Forkel, Robert and List, Johann-Mattis and Greenhill, Simon J. and Rzymski, Christoph and Bank, Sebastian and Cysouw, Michael and Hammarstrom, Harald and Haspelmath, Martin and Kaiping, Gereon A. and Gray, Russell D. (2018): Cross-Linguistic Data Formats, advancing data sharing and re-use in comparative linguistics. Scientific Data 5.180205. 1-10. https://doi.org/10.1038/sdata.2018.205

List, Johann-Mattis and Forkel, Robert and Greenhill, Simon J. and Rzymski, Christoph and Englisch, Johannes and Gray, Russell D. (2022): Lexibank, A public repository of standardized wordlists with computed phonological and lexical features. Scientific Data 9.316. 1-31. https://doi.org/10.1038/s41597-022-01432-0

Moran, Steven and Cysouw, Michael (2018): The Unicode Cookbook for Linguists: Managing writing systems using orthography profiles. Berlin:Language Science Press.

Steven Moran, Robert Forkel, Christoph Rzymski, and Johann-Mattis List (2022): Segments. Unicode Standard tokenization routines and orthography profile segmentation [Software Library, Version 2.2.1]. Leipzig: Max Planck Institute for Evolutionary Anthropology. https://pypi.org/project/segments