Syllabic Form Folding

Daniel Yacob, Ge'ez Frontier Foundation

Basic Folding

Rows and Columns (or "Enter the Matrix")

In an open syllabary each letter will represent a consonant followed by a vowel (a "CV" pattern). For education and pedagogical purposes, the syllabary will most typically be organized into a two-dimensional matrix. Each row presents all forms of a particular consonant base. Graphically the symbols of the consonant base will appear very similar across the row with only a minor difference (often a graphical appendage) to indicate which vowel is present. The Cherokee syllabary is an exception in this case where a consonant group has little visual similarity.

The symbols added to the graphical base that denotes the vowel component are approximately the same for every consonant group. The symbol will sometimes have to be adjusted to attach to a more complex graphical base. To exploit regularity, the elements in the consonant group will be ordered in the same way across the row. The vowel sequence across a row is then identical over all rows. Consistent columns then emerge for each vowel inflection. These columns, which represent vowel forms of the syllabary matrix, may also be named.

Since the consonant and graphical base remain constant across a row (or more precisely the "read direction" of the matrix) the association of the consonant to the syllabic series is stronger than that of the vowel. The consonant is the primary component of the syllable and the vowel the secondary component. Changing the vowel component would be a column-wise move within a syllabic series. Changing the consonant component is a row-wise move and necessarily changes to a new syllabic series altogether. These consonant based groupings also define an equivalence class. Figure one presents an example syllabary that we will utilize in our discussion.

Figure 1: A Syllabary with Six Forms

	a	**e**	**i**	**o**	**u**	**(none)**	*Each row is a "syllabic series" named after its "base".*
c	○ (ca)	◐ (ce)	◑ (ci)	◓ (co)	◒ (cu)	● (c)	This is the "ca series", also called the "ca family", its base is "○".
d	□ (da)	◧ (de)	◨ (di)	⬒ (do)	⬓ (du)	■ (d)	This is the "da series", also called the "da family", its base is "□".
n	◇ (na)	⬖ (ne)	⬗ (ni)	◆ (no)	⬙ (nu)	◆ (n)	This is the "na series", also called the "na family", its base is "◇".
p	△ (pa)	◭ (pe)	◮ (pi)	◬ (po)	◬ (pu)	▲ (p)	This is the "pa series", also called the "pa family", its base is "△".
r	▷ (ra)	◧ (re)	◨ (ri)	⊳ (ro)	⊳ (ru)	▶ (r)	This is the "ra series", also called the "ra family", its base is "▷".

s	◁ (sa)	◀ (se)	◀ (si)	◀ (so)	◀ (su)	◀ (s)	This is the “sa series”, also called the “sa family”, its base is “◁”.
t	▽ (ta)	▼ (te)	▼ (ti)	▼ (to)	▼ (tu)	▼ (t)	This is the “ta series”, also called the “ta family”, its base is “▽”.

Note that in our simple syllabary the 7th column (or 6th form) is the vowel-less form. It may seem like a contradiction in terms to have a syllable with a consonant and no vowel. Most (all?) syllabaries do however have a vowel-less form that, depending on the word, will have a very weakly pronounced vowel (dotless-i, “ɪ”, in IPA) present or absolutely none at all (in consonant clusters for example).

Forms are the Cases

It is sometimes useful when considering the folding of syllables to think of each syllabic vowel form as a different “case”. The analogy is useful when first considering the issue under the familiar “case-folding” idiom. The objective is the same, we want to convert all manifestations of a letter into a single easily comparable state. After a certain point however the case context becomes only weakly applicable to the vowel states and the notion becomes misleading. To avoid confusion, we will avoid the use of the “case” analogy and refer to the vowel states of syllabic series as a “form”.

As alluded to in the preceding paragraph syllabic folding converts a given syllable into a representative form. The representative form is generally considered to be the first form which is also the graphic base of the series. While any form will do for the purpose of comparison, the first form is the most intuitive because it is also considered the representative in other contexts.

However, as a matter of practicality the “reading quality” of a string of syllables all in the first form is found to be cognitively awkward and tedious. Using instead the vowel-less form as the representative is found to require less cognitive energy to work with and fits in naturally with the folding being against the vowel component of the syllable. Syllabic folding then converts the syllables into their vowel-less state.

Syllabic folding, unlike case folding, is grammatically destructive, the meaning of the word is altered not only its appearance. Hence folding should only be used for matching and never for formatting.

Matching Problems

Substrings & Form Insensitivity

The crux of the syllabic pattern matching problem comes from the fusion of consonants and vowels into a single symbol. The character boundary encompasses two phoneme boundaries. The consonant and vowel are both properties of the character, but we have no means by which to specify or mask one or the other in regular expressions languages.

To illustrate the problem, consider an example from an alphabetic system such as English, where consonants and vowels are always separate symbols. The word “can” is a simple substring of the words “canada”, “canister”, “canopy” and an imaginary word “stocanura”. Turned into syllables and adjusting some phonetics along the way (ignoring that “ca” is better as “ka” but converting “py” into “pi”) the words become “(ca)(na)(da)”, “(ca)(ni)(s)(te)(r)”, “(ca)(no)(pi)” and “(s)(to)(ca)(nu)(ra)”. None of which could be matched by “(ca)(n)” because “(n)” is a different symbol (and codepoint) than “(na)”, “(ni)”, “(no)” and

"(nu)". This problem never emerges in alphabets as the built-in separation of vowels and consonants makes it impossible (there is no overlap in character-phoneme boundaries).

Figure 2: A Syllabary Search for ○◆

Original	Folded	Match on Fold
○◆ (ca)(n)	●◆ (c)(n)	●◆ (c)(n)
○◇□ (ca)(na)(da)	●◆■ (c)(n)(d)	●◆■ (c)(n)(d)
○⬗◀▽▶ (ca)(ni)(s)(te)(r)	●◆◀▼▶ (c)(n)(s)(t)(r)	●◆◀▼▶ (c)(n)(s)(t)(r)
○◈◭ (ca)(no)(pi)	●◆▲ (c)(n)(p)	●◆▲ (c)(n)(p)
◀▽○◈▷ (s)(to)(ca)(nu)(ra)	◀▼●◆▶ (s)(t)(c)(n)(r)	◀▼●◆▶ (s)(t)(c)(n)(r)

In a Perl example:

```
my $haystack = join("",<FILE>);
my $needle   = "○◆";

if ( $haystack =~ /$needle/i ) {  # folding here should be applied as above
    :
    :
}
```

where `$needle` folds to "●◆". Comparison to words in the `$haystack` would likewise be made within the folded form. Note that the case insensitivity here is equivalent to expanding each symbol into all of its forms. That is, the English expression /can/i is equivalent to `/[Cc][Aa][Nn]/` (which in turn is short for `/(C|c)(A|a)(N|n)/`). The case insensitivity flag, i, is then a tool of convenience that relieves the developer of the tedium of composing the expanded form. The flag also aids in keeping the expression easier to read, understand and maintain.

The equivalent expansion for a syllabary would be: /[○◐◑◓◒●][◇⬖⬗◈◆◆]/ . Clearly, this is a considerably greater burden for the developer working with syllabaries. This is also representative of the present status of RE languages and their support for syllabic character proprties. For want of syllabic folding, this is how developers working with syllabic scripts must construct form insensitive expressions.

Back to Class(es)

The example with `/○◆/i`. helps illustrate another point. While the expansion shown is accurate for a form insensitive match of the two syllables, we did not truly desire that the entire sequence be matched form insensitively. A form insensitive match for "(ca)(n)", would match all forms of "(n)" as desired, but we were not also interested in all forms of "(ca)" in this case (we did not want "(ce)", "(ci)", etc.). We wanted

to restrict our match to "○" specifically followed by any form of the "◇" family. What we were really after was: /○[◇◧◨◆◆◆]/.

To be certain, form insensitive matches *are desirable* and do save the burden of work illustrated in the last example. The longer the syllabic sequence, the more labour will be saved with a form insensitive match feature.

The /○[◇◧◨◆◆◆]/ expression is still a little awkward and laborius to use. The range equivalent /○[◇-◆]/ is better but hazardous since the range operator, -, is likely to be applied over the initial and final character code address range. There is no certainty that syllables in a family are in contiguous address space or even ordered as expected.

The ideal form is then to apply syllabic character classes giving us the concise expression: /○[#◇#]/. A discussion on syllabic character classes can be found in the article An Introduction to Syllabaries and Syllabic Pattern Matching.

Advanced Folding

Thinking again in terms of the "case" idiom a new circumstance will arise in a multi-case writing system. When you have more than two cases to contend with, you may not want to fold all cases all the time. This is an issue that never arises with Roman (aka "Latin") alphabet since cases are in a binary system -when you fold one case, you fold all cases. Before long a developer will desire a means by which to restrict which cases get folded.

A solution is to make use of the occurrence limiter notation in regular expressions languages under a syllabic context. The syllabic context can be set with the % operator. For example, the expression: /◇{%3-5}/ would indicate that only forms 3-5 should be folded together. The expansion here would be: /[◨◆◆]/. However, the expression is not an option when the expression is not constant. For instance, if the folding limiter was applied against a variable: `/$x{%3-5}/`.

The folding limiter may grow be applied in a negation context whereby folding occurs for all forms except for those specified: `/$x{^%3-5}/`.

Likewise, the limiter may be applied to ranges: `/[○□]{%1,4-6}/` and `/[▷-▽]{^%4-6}/`.

Form conversions are often desirable. Conversion may be applied in Perl transliterations and substitutions for example:

```
tr/[#1-3#]/[#4-6#]/;
```

and

```
$word =~ s/[#7#]([#□#])/[#2#]$1/g;
```