Font Layout Table: Knowlege base for Complex Text Layout

K. Handa M. Nishikimi N. Takahashi S. Tomura *

Abstract

We have designed a format called *Font layout table* that represents a script and writing-system dependent knowledge for *Complex Text Layout*.

A font layout table is used to convert a character code sequence into a glyph sequence to be displayed. It consists of rules, structured as cascaded stages, that may contain regular expressions and character categories. This combination enables flexible code sequence manipulation.

This paper describes the problems of *complex text layout* and how *font layout table* solves them.

1 introduction

Drawing text (text layouting) is one of the most important and basic facilities of text processing systems. Complex scripts, especially those used in Asia, require complicated transformations from the character sequence in the memory to the glyph string to be displayed. Characters may change their shapes according to the context or may be combined together to produce special shapes. This process is called *complex text layout* (CTL hereafter), and depends on many kinds of knowledge: scripts, languages, and orthographies affect text layout.

In existing text layouting systems, the knowledge required for CTL has been stored in programs or/and in fonts. Current tendency is to drive OpenType fonts which contain font-specific knowledge by programs in which script/language-specific knowledge is hard coded. However, the latter knowledge tends to become very complicated (especially when represented by a program), and it is not easy to maintain such programs.

In addition, as each text layouting system (e.g. Gtk+/Pango, Qt, OpenOffice/ICU) has its own programs for CTL, when the Unicode Standard is updated and new characters are added that require special treatment on rendering (actually that happens often), all of such programs must be fixed.

To solve these problems, we have designed a new resource, called the *font layout table* (FLT hereafter), to store knowledge about scripts and languages in a compact and simple form, and also developed a library[1]that performs CTL by utilising FLT.

A font layout table stipulates the rendering process in the form of rules. Rules are structured as cascaded stages, so it is possible to formulate a new CTL step by step. Rules may contain regular expressions and character categories. This combination enables flexible character sequence manipulation. With FLTs, even end users can write and add support for new scripts.

In this paper, we first summarise the task of CTL. We then explain how FLT performs that task with a few example, and conclude with a discussion of future work.

2 Complex Text Layout

The task of CTL is to generate a glyph sequence from a given character sequence and a font. Each glyph has the information about the glyph-code, the relative coordinate from the base position of the glyph, and the logical width of the glyph. We have modeled the procedure for that task by the following five steps:

- Cluster extraction

  The first step is to extract a cluster from the given character sequence. A cluster is a minimal character sequence with which we can decide their graphical shape[2]. For instance, in the case of Thai, a cluster is a base consonants and the succeeding combining vowel and/or tone-mark. The following steps can work on each cluster independently.

[*] AIST (National Institute of Advanced Industrial Science And Technology), Japan

[1] We have been developing a general purpose multilingual text processing library for Unix/Linux systems. The library, named "the m17n library", was first released in March 2004 and is now packaged in a variety of Linux distributions.

[2] This definition is a little bit different from *Grapheme Cluster* defined in the Unicode Standard as below, but is more practical in a point of CTL view:

- Character reordering

  Character reordering changes the order of characters within a cluster. Unicode basically adopts a logical (or phonetic) order in character encoding, but many of Asian scripts has a vowel that should be placed before, that is, to the left of the base consonant. In addition, a cluster may have multiple consonants and the first one is not the base consonant, in such case we must move the base consonant to the head of a cluster.

  This reordering is necessary because OTF features are usually applicable only to a glyph sequence that is already set in a visual order.

- Character to glyph mapping

  A glyph code (or Glyph ID) of a font is often different form a character code, and we must change a character code sequence to a glyph code sequence.

- Glyph substitution

  In many CTLs, this is the most complex step. This step is typically performed by applying GSUB features of OTF to a sequence of glyphs. Some of them produce ligatures, some change glyphs to a combining form, and some adjust glyph shapes for better design. We must apply different features to the same glyph depending on a context.

- Glyph positioning

  The last step typically applies GPOS features of OTF to adjust the positioning of glyphs. Some combining marks must be placed on top of the base glyph(s) relatively.

3 Font Layout Table

FLTs describe the conversion process from a character sequence to a glyph sequence as multiple cascaded stages. Each stage has a category table that classifies characters that behave similarly in layout, and a generator that consists of a set of conversion rules. Except for the first stage, a category table can be omitted if a stage uses the same one as the previous stage.

```
FLT ::= CATEGORY-TABLE GENERATOR [ CATEGORY-TABLE ? GENERATOR ] *
```

The category table maps characters into category letters (Latin alphabets) which can be used within a regular expression in a generator. The first category table also works to specify a set of characters supported by the FLT.

The generator of each stage contains one top-level rule that is applied to every sequence it receives. A top-level rule may contain sub-rules that work on subsequences. Each rule consists of a *matcher* and the following *command*s. A rule itself can be an command.

```
GENERATOR ::= RULE
RULE ::= '(' MATCHER COMMAND * ')'
COMMAND ::= [ BUILTIN-COMMAND | RULE | CONDITIONAL ]
CONDITIONAL ::= '(cond' RULE * ')'
```

A matcher is a regular expression, an index number of a segment matched with the previous regular expression or a character code range, and it specifies the part of the input sequence to which the following commands are applied.

```
MATCHER ::= REGEXP | SEGMENT-INDEX | RANGE
```

In the following part of this section, we briefly explain how FLT perform the task of CTL along with an example of Khmer word "ហើយ (already)" shown in Figure 1.

Grapheme Cluster. A maximal character sequence consisting of a grapheme base followed by zero or more grapheme extenders or, alternatively, by the sequence ¡CR, LF¿. A grapheme cluster represents a horizontally segmentable unit of text, consisting of some grapheme base (which may consist of a Korean syllable) together with any number of nonspacing marks applied to it.

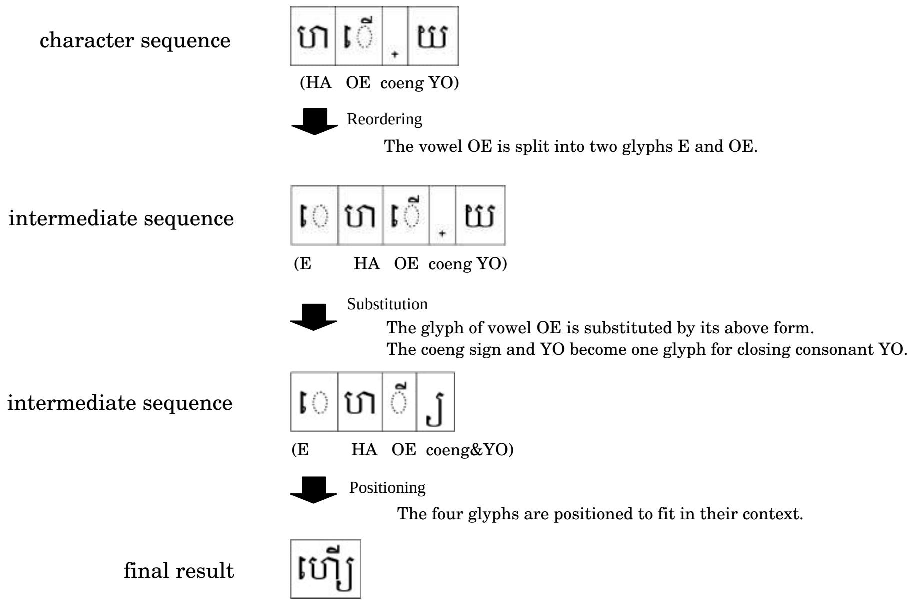

character sequence
(HA OE coeng YO)
Reordering
The vowel OE is split into two glyphs E and OE.
intermediate sequence
(E HA OE coeng YO)
Substitution
The glyph of vowel OE is substituted by its above form.
The coeng sign and YO become one glyph for closing consonant YO.
intermediate sequence
(E HA OE coeng&YO)
Positioning
The four glyphs are positioned to fit in their context.
final result

Figure 1: Drawing a Khmer word by multiple stages

Here's the first category table of FLT for Khmer OTF.

```
(category
 (0x1780 0x17FF ?E)        ; E: anything else
 (0x19E0 0x19FF ?E)
 (0x1780 0x17B3 ?C)        ; C: consonant & independent vowel
 (0x179A        ?R)        ; R: RO
 (0x17B6        ?p)        ; p: vowel sign (post)
 (0x17B7 0x17BA ?a)        ; a: vowel sign (above)
 (0x17BB 0x17BD ?b)        ; b: vowel sign (below)
 (0x17BE        ?A)        ; A: vowel sign (two-part, above)
 (0x17BF 0x17C0 ?P)        ; P: vowel sign (two-part, post)
 (0x17C1 0x17C3 ?m)        ; m: vowel sign (pre)
 (0x17C4 0x17C5 ?P)
 (0x17C6        ?s)        ; s: sign (above)
 (0x17C7 0x17C8 ?S)        ; S: sign (post)
 (0x17C9 0x17CA ?c)        ; c: consonant shifter
 (0x17CB 0x17D1 ?s)
 (0x17CC        ?r)        ; r: ROBAT
 (0x17D2        ?H)        ; H: COENG
 (0x17D3        ?s)
 (0x17DD        ?s)
 (0x200C        ?N)        ; N: ZERO WIDTH NON-JOINER
 (0x200D        ?J)        ; J: ZERO WIDTH JOINER
 )
```

In order to process the character sequence, the FLT has to know which letters are consonants or two-part vowel signs, etc. This category table assigns such information to each character. The character sequence shown in the Figure 1, "HA OE coeng YO" (the code sequence of "`0x17A0, 0x17BE, 0x17D2, 0x1799`") is thus assigned categories `CAHC` by this table.

The generator of the first stage of this FLT has the following codes (middle part is omitted), and there appear rules whose *matcher* are regular expressions matching with one of Khmer syllable patterns (alphabets are category letters defined in the above example).

```
(generator
 (0
  (cond

   ;; pre vowel sign
   ;;1    2       3       4  5     67              8   9    10
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?)(m)(s*S?(H[CR])?)"
    < | (8 =) (4 = =) (1 = *) (5 =) (6 = *) (9 = *) | > )

   ;; two-part above vowel sign
   ;;1    2       3       4  5     67              8
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?As*S?(H[CR])?)"
    < | 0x17C1 (4 = =) (1 = *) (5 =) (6 = *) | > )

   ;; post vowel sign
   ;;1    2       3       4  5     67              8   9   10 11
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?)(p)(s*)(S?(H[CR])?)"
    < | (4 = =) (1 = *) (5 =) (6 = *) (9 = *) (8 =) (10 = *) | > )

   [...]

   ("." =))
  *))
```

In the top-level rule, the matcher is `0`, which refers to the whole current run, so this rule covers every sequence regardless of its content. A syllable is defined in the first five branches of the conditional `cond` command. The last branch is a fallback case when the first character in the sequence does not constitute a syllable. This fallback case simply makes a copy of the target "`.`" (a full stop, which means any one character) and passes it to the next stage. The `*` (asterisk) command in the last line means the repetition of the preceding command, conditional command in this case. Therefore this process of clustering and reordering is repeated until every code in the input sequence is consumed.

The `cond` executes the first rule whose matcher matches with the head of the current run. Our example in Figure 1, "HA OE coeng YO" is `CAHC` in categories, so it does not match the first but the second, as shown below.

```
Segment index:      1    2       3       4  5     67                  8
Regular expression: ([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?As*S?(H[CR])?)
Matched category:    C                                      A     H C
Character:           HA                                    OE coeng YO
```

With this match, the `CAHC` is segmented into `(C()())(())(()A(HC))`. In the command part, only the first, fourth, fifth and sixth segments are mentioned (other segments are included in them), and their contents are "HA", empty, empty, and "OE coeng YO", respectively.

Now let us see the command part.

```
 < | 0x17C1 (4 = =)(1 = *)(5 =)(6 = *) | > )
```

The commands `<` and `>` specify a grapheme cluster. The command `|` produces a special glyph whose category is " " (SPACE). This special glyph works as a delimiter between clusters in the later stages. The command `=` copies the first character of the matched segment into the output sequence, and the command `*` instructs repetition, so the command `(= *)` produces the whole copy of the matched segment. In total, the command part first produces the character `0x17C1` (Khmer E), and then copies the first two characters in the fourth segment, all the codes in the first segment, the only character in the fifth, and all the characters in the sixth. Our example is now changed to "E HA OE coeng YO", or `mCAHC` in categories.

As you see, regular expressions with reference to the content of segmented groups and repetitions of patterns are very powerful.

In the second stage of the current example, the control structure of generator is the same as in the first stage, so we show below only one branch that is used to rewrite our example.

```
(generator
 (0
  (cond
   ;; no consonant shift sign
   ;; 1   2    3       45       6           7   8     9   10  11
   (" (m)?(HR)?([CR]r?)((HCr?)*)([NJ]?[ba]?)(A)?(s*p?)(P)?(S)?(H[CR])? "
    |
    (1 =)
    (2 otf:khmr=pref+)
    (3 = *)
    (4 otf:khmr=blwf,pstf+)
    (6 = *)
    (7 otf:khmr=abvf+)
    (8 = *)
    (9 otf:khmr=pstf+)
    (10 =)
    (11 otf:khmr=blwf,pstf+)
    | )
        :     :
  *))
```

The example sequence is now “E HA OE coeng YO”, or `mCAHC` in categories, so it matches this branch by assigning `m` to the first segment, `C` to the third, `A` to the seventh, and `HC` eleventh. In the command part, new commands that begin with `otf:` appear. They are instructions to the OpenType font driver. By specifying a script tag name (`khmr` in this generator) and a GSUB feature tag name (`pref`, `blwf`, etc.), the feature is applied to the target. The “OE” in our example is the seventh segment, thus the `abvf` feature is applied. As a result it changes its shape, as shown in Figure 1.

In the third stage, which is for applying the other GSUB features, and lastly GPOS features to position glyphs, the generator also uses commands to instruct the OpenType font driver, with other feature tag names for conjuncts or typographical forms, although we do not elaborate on the detail here. See the Appendix.C for the full code of the FLT for Khmer.

4 conclusion

CTL requires much knowledge of text processing, scripts, and writing systems, and we often have to update it because of newly added characters and/or rules. This is why it is so important to represent the knowledge in a flexible and easily-maintainable way. We believe FLT fulfils these requirements.

The drawback of FLT compared with hard-coded programs is its processing speed. Our experiment of using FLT driver of the m17n library in a Pango module shows that it is about 4 times slower than the original C code. The reason of the slowness is, of course, the overhead of interpreting FLT. But, as the current API of the m17n library is fairly high level, we can make the speed much faster by providing a lower level API.

In addition, the task of text layouting is not limited to CTL. It consists of many other tasks such as text formatting (line-breaking, justifying, etc), font selection, the actual rendering on the screen, etc. So, the slowness of CTL processing doesn’t directly affect the total speed of text layouting.

We are going to estimate how much CTL processing affects the whole job while tuning up the current FLT drivers.

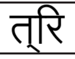
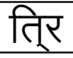
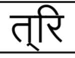
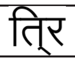
We also consider FLT as a tool for rapid prototyping to support newly added scripts.

APPENDIX

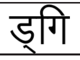
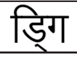
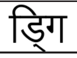
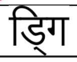
A Bugs of CTL programs (case study)

Incorrect rendering is indicated by red column title.
Questionable rendering is indicated by magenta column title.

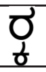
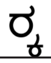
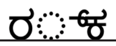
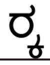
- Devanagari: Explicit halant form caused by ZWNJ (p.305 R15 Unicode 5.0)

Characters	FLT	gedit	kedit	OpenOffice
0924 094D 200C 0930 093F				

- Devanagari: Halant form caused by the absence of a conjunct form in the font (p.309 R16, Unicode 5.0).

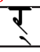
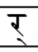
Characters	FLT	gedit	kedit	OpenOffice
0921 094D 0917 093F				

- Kannada: Forced below form caused by ZWJ (p.334, Unicode 5.0).

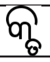
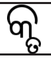
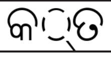
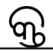
Characters	FLT	gedit	kedit	OpenOffice
0CB0 200D 0CCD 0B95				

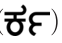
The font used here is "Lohit Kannada" which has the GPOS feature *blwm*, and that moves the below form of U+0B95 to just below the base consonant. It seems that gedit and OpenOffice don't apply that feature.

- Devanagari: Eyelash-RA; not RA_{sup} for compatibility with Unicode 2.0 (p.305 R5a, Unicode 5.0)

Characters	FLT	gedit	kedit	OpenOffice
0930 094D 200D				

- Oriya: Forced Oriya below form caused by ZWJ. (p.303 Fig.9-7, Unicode 5.0)

Characters	FLT	gedit	kedit	OpenOffice
0B15 200D 0B4D 0B24				

If there is no ZWJ, RA + HALANT is depicted with the REPH form ().

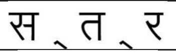
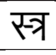
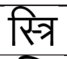
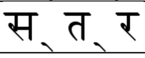
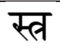
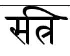
B How to solve this problem?

In the page 309 of Unicode 5.0, the paragraph for R16 starts as this:

> The presence of an explicit virama (either caused by a ZWNJ or by the absence of a conjunct in the font) blocks this reordering, and the dependent vowel I_{vs} is rendered after the rightmost such explicit virama.

To handle that rule (the red part), the FLT for Devanagari OTF checks the number of glyphs after having applied the *half* and *pres* OTF features to the pre-base and base consonants. If the font has a conjunct form, there should be only one glyph; otherwise two or more glyphs. In the latter case, FLT moves the pre-base vowel sign to the left of the final glyph (the case of "Lohit Hindi" font below).

However, we found that some fonts represent a single conjunct form with multiple glyphs. In that case, FLT fails to place the pre-base vowel modifier at the right position (the case of "Chandas" font below).

Font	Characters		Conjunct form	With Vowel I
Lohit Hindi	0938 094D 0924 094D 0930 (093F)			
Chandas	0938 094D 0924 094D 0930 (093F)			

C Full code of Khmer FLT (KHMR-OTF.flt)

```
;; KHMR-OTF.flt -- Font Layout Table for Khmer OpenType fonts
;; Copyright (C) 2005, 2007
;;   National Institute of Advanced Industrial Science and Technology (AIST)
;;   Registration Number H15PRO112

;; This file is part of the m17n database; a sub-part of the m17n
;; library.

;; The m17n library is free software; you can redistribute it and/or
;; modify it under the terms of the GNU Lesser General Public License
;; as published by the Free Software Foundation; either version 2.1 of
;; the License, or (at your option) any later version.

;; The m17n library is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
;; Lesser General Public License for more details.

;; You should have received a copy of the GNU Lesser General Public
;; License along with the m17n library; if not, write to the Free
;; Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
;; Boston, MA 02110-1301, USA.

;;; <li> KHMR-OTF.flt
;;;
;;; For Khmer OpenType fonts to draw Khmer.
;;; A Font is available from
;;; <http://www.khmeros.info/drupal/?q=en/download/fonts>.

(font layouter khmr-otf nil
      (font (nil nil unicode-bmp :otf=khmr=pres)))

(category
 (0x1780 0x17FF ?E)                    ; E: else
 (0x19E0 0x19FF ?E)
 (0x1780 0x17B3 ?C)                    ; C: consonant & independent vowel
 (0x179A        ?R)                    ; R: RO
 (0x17B6        ?p)                    ; p: vowel sign (post)
 (0x17B7 0x17BA ?a)                    ; a: vowel sign (above)
 (0x17BB 0x17BD ?b)                    ; b: vowel sign (below)
 (0x17BE        ?A)                    ; A: vowel sign (two-part, above)
 (0x17BF 0x17C0 ?P)                    ; P: vowel sign (two-part, post)
 (0x17C1 0x17C3 ?m)                    ; m: vowel sign (pre)
 (0x17C4 0x17C5 ?P)
 (0x17C6        ?s)                    ; s: sign (above)
 (0x17C7 0x17C8 ?S)                    ; S: sign (post)
 (0x17C9 0x17CA ?c)                    ; c: consonant shifter
 (0x17CB 0x17D1 ?s)
 (0x17CC        ?r)                    ; r: ROBAT
 (0x17D2        ?H)                    ; H: COENG
 (0x17D3        ?s)
 (0x17DD        ?s)
 (0x200C        ?N)                    ; N: ZERO WIDTH NON-JOINER
 (0x200D        ?J)                    ; J: ZERO WIDTH JOINER
 )

;; Generic syllable pattern is as follows.
;; [CR](r|N?c)?(HCr?)*(HR)?(HCr?)*[NJ]?(m|b|a|A|p|P)?s*S?(H[CR])?

;; Step 0.
;; Move m to the beginning.
;; Split A and P.
;; Exchange the order of pP and s.
;; Move HR before the base.

(generator
 (0
  (cond

   ;; pre vowel sign
   ;;1    2       3       4  5     67            8  9    10
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?)(m)(s*S?(H[CR])?)"
    < | (8 =) (4 = =) (1 = *) (5 =) (6 = *) (9 = *) | > )

   ;; two-part above vowel sign
   ;;1    2       3       4  5     67                8
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?As*S?(H[CR])?)"
    < | 0x17C1 (4 = =) (1 = *) (5 =) (6 = *) | > )

   ;; post vowel sign
   ;;1    2       3       4  5     67            8  9   10 11
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?)(p)(s*)(S?(H[CR])?)"
    < | (4 = =) (1 = *) (5 =) (6 = *) (9 = *) (8 =) (10 = *) | > )

   ;; two-part post vowel sign
   ;;1    2       3       4  5     67            8  9   10 11
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?)(P)(s*)(S?(H[CR])?)"
    < | 0x17C1 (4 = =) (1 = *) (5 =) (6 = *) (9 = *) (8 =) (10 = *) | > )

   ;; other vowel signs or no vowel sign
   ;;1    2       3       4  5     67                     8
   ("([CR](r|N?c)?(HCr?)*)(HR(r)?)?((HCr?)*[NJ]?[b|a]?s*S?(H[CR])?)"
    < | (4 = =) (1 = *) (5 =) (6 = *) | > )

   ("." =))
  *))

;; Now a syllable looks like below.
;; m?(HR)?[CR](r|N?c)?(HCr?)*[NJ]?(b|a|A)?s*(p|P)?S?(H[CR])?

;; Step 1.
;; Set the form of consonant shifter.

(generator
 (0
  (cond
   ;; shifter + above vowel sign without ZWNJ
   ;; Shifter take blwf.  HR takes pref.  HC's take blwf or pstf.
   ;; 1   2    3     4      5     6     7     8
   (" (m)?(HR)?([CR]c(HCr?)*(N|J)?(a|A))(s*S?)(H[CR])? "
    |
    (1 =)
    (2 otf:khmr=pref+)
    (3 otf:khmr=blwf,abvf,pstf+)
    (7 = *)
    (8 otf:khmr=blwf,pstf+)
    | )

   ;; shifter + ZWNJ + above vowel sign, or, shifter without above vowel sign
   ;; Shifter stays above.  HR takes pref.  HC's take blwf or pstf.
   ;; 1   2    3        45       6           7   8     9   10  11
   (" (m)?(HR)?([CR]N?c)((HCr?)*)([NJ]?[ba]?)(A)?(s*p?)(P)?(S)?(H[CR])? "
    |
    (1 =)
    (2 otf:khmr=pref+)
    (3 = *)
    (4 otf:khmr=blwf,pstf+)
    (6 = *)
    (7 otf:khmr=abvf+)
    (8 = *)
    (9 otf:khmr=pstf+)
    (10 =)
    (11 otf:khmr=blwf,pstf+)
    | )

   ;; no shifter
   ;; 1   2    3       45       6           7   8     9   10  11
   (" (m)?(HR)?([CR]r?)((HCr?)*)([NJ]?[ba]?)(A)?(s*p?)(P)?(S)?(H[CR])? "
    |
    (1 =)
    (2 otf:khmr=pref+)
    (3 = *)
    (4 otf:khmr=blwf,pstf+)
    (6 = *)
    (7 otf:khmr=abvf+)
    (8 = *)
    (9 otf:khmr=pstf+)
    (10 =)
    (11 otf:khmr=blwf,pstf+)
    | )

   ("." =))
  *))

;; Now a syllable looks like below.  ~ characters are OTF feature applied.
;; m?(HR)?[CR](r|c)?(HCr?)*[NJ]?(b|a|A)?s*(p|P)?S?(H[CR])?
;;    ~~   ~~    ~   ~~~~            ~       ~     ~~~~~

;; Step 2.
;; Concatenate adjacent Khmer syllables.
;; Remove J.  Retain N to prevent ligature.

(generator
 (0
  (cond
   ("  ")
   ("J")
   ("." =))
  *))

;; Step 3.
;; Apply other OTF features.

(generator
 (0
  (cond
   (" ([^ ]*) "
    (1 otf:khmr=pres,blws,abvs,psts,clig))
   ("."
    [ otf:khmr=+ ]))
  *))

;; Step 4.
;; Remove N to clean up.
(generator
 (0
  (cond
   ("N")
   ("." =))
  *))
```