FLT: Font Layout Table

Kenichi Handa, Mikiko Nishikimi, Naoto Takahashi and Satoru Tomura*

Abstract

Rendering a complex text such as one written in Indic scripts, or Complex Text Layout requires many kinds of knowledge for scripts and writing system. To date, those kinds of knowledge are scattered in fonts or text renderers, which makes it difficult to support new scripts or new fonts.

We propose a new system for Complex Text Layout: Font Layout Table. Font Layout Table is a resource that provides knowledge about writing systems of various scripts and languages, and it bridges a text renderer and fonts.

Our library for multilingualization, *the m17n library* utilizes FLT and succeeded in rendering such scripts as Devanagari, Tibetan, Thai, Khmer, and etc.

1 Introduction

We have been developing a general purpose multilingual library called "the m17n library." One of the most important facilities of the library is displaying, or rendering multilingual text properly. Texts in some scripts can be rendered by just lining glyphs up, but others require complex processing called Complex Text Layout.

Complex Text Layout requires much knowledge for processing text, scripts and writing systems. Such knowledge, however, is often distributed in text renderers and fonts inconsistently, which makes it hard to add a support for new scripts or fonts, or to use fonts of different formats.

We propose a new system for storing the knowledge: Font Layout Table. Font Layout Table (FLT for short hereafter) is a resource that provides knowledge about writing systems of different scripts and languages, and it bridges the text renderer of the m17n library and various kinds of fonts.

In section 2, we briefly explain how a complex text is rendered. Section 3 overviews the several ideas proposed so far to perform Complex Text Layout and then section 4 describes FLT. Section 5 shows how each task in Complex Text Layout is realized with FLT.

2 Complex Text Layout

Rendering a text is easy for easy kinds of scripts. A text renderer converts character codes into glyph codes one by one, consulting the encoding of the font selected, and then puts the glyphs side by side. That is all. There are many scripts, however, that requires more complex processing for rendering. Asian scripts such as Thai and Devanagari are good examples. In such scripts, a sequence of characters may have to be drawn by a single ligature glyph, or glyphs may have to be drawn at 2-dimensionally shifted positions. Furthermore, glyphs may require reordering. The technology for such a rendering is known by the name "complex text layout" (CTL for short hereafter).

Complex text layout is typically carried out with the following steps:

1. Clustering

   The CTL processor has to partition a character sequence to be rendered into processing units. The unit is characters of a minimum length that can decide how a renderer should work with the characters. For example, the unit is a syllable in Indic scripts, or it is a sequence of a base character and the following combining characters in Thai scripts. We call the unit as a "cluster".

Keywords: Complex Text Layout, Font Layout Table, the m17n library

*National Institute of Advanced Industrial Science and Technology (AIST)

2. Reordering

   The order of the glyphs is not always the order of characters. A text in the computer memory has characters aligned in one order, but sometimes the glyphs that correspond to characters (or character sequences) of the text should be arrayed in other order. In such cases, The CTL processor has to reorder characters in the text before further handling.

3. Character to glyph mapping

   A character code sequence is converted into a glyph code sequence. Usually this conversion is a simple one to one mapping. In CTL, however, a single character may correspond to a sequence of glyphs, or a sequence of characters should be converted into to a single glyph. That is, in this step, CTL requires a context dependent flexible mapping of one to one, one to many, many to one, and many to many.

4. Glyph substitution

   A glyph may have multiple variants but the context decides which is the right one. For instance, in the case of Arabic, we should select the right glyph from initial, medial, and final forms. In the case of Devanagari, the width of upper horn of some vowels must be adjusted according to the consonant(s).

   A sequence of glyphs may form a ligature, in which case a ligature glyph is substituted for the sequence. In this step also, a flexible mapping is indispensable and actually in some cases this step can be combined with the previous step for efficiency.

5. Glyph positioning

   Now that the CTL processor has the correct sequence of glyphs, it should place those glyphs next to next on the baseline. Some of them, however, have to be positioned shifted horizontally and/or vertically. The position is decided relatively to the previous glyphs.

In every step of CTL, knowledge specific to each script is required. The problem is where such knowledge should be kept.

3 Where to keep knowledge?

To date, CTL is executed in several different fashions that we review below.

3.1 Legacy Systems

In a legacy system, an application program can handle single script only, with a font specific to the program. Such a font just provides glyphs and the knowledge for CTL is embedded in the programs. The defect of this method is its extreme inflexibility. The renderer of such a program must be modified just to support a new font. In order to support a new script, the renderer must be re-written from scratch, as shown in Fig. 1.

3.2 OpenType Font

An OpenType Font (or OTF in short) has some of the knowledge for CTL. It contains glyph data plus knowledge enough for the last three tasks: Character to glyph mapping, glyph substitution, and glyph positioning. As the other two tasks (clustering and reordering) must be performed by a renderer, the renderer has to keep the knowledge about writing systems, which makes it difficult to add a support for a new script or to fix a bug of the knowledge.

Fig. 2 shows how the knowledge is divided in OpenType font and its renderer: Uniscribe.

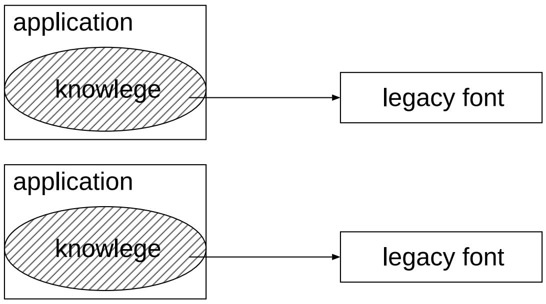

Figure 1: Legacy system and CTL knowledge

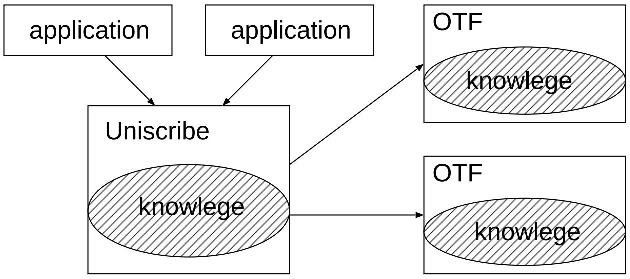

Figure 2: OpenType font and CTL knowledge

3.3 Graphite System

A Graphite system constitutes of Graphite renderer and Graphite fonts. Graphite fonts are fully intelligent and contain all the knowledge for CTL. A Graphite font is realized as a TrueType font compiled with one extensional table containing “Graphite Description Language” (GDL for short). GDL represents knowledge for CTL as a powerful rule-base description.

In this system, every font has to carry the whole knowledge about a writing system, even though multiple fonts use one same knowledge, because there is no place to keep it. Thus, in order to fix a bug of the same embedded knowledge, all the fonts that have the knowledge must be re-built.

Moreover, GDL does not have a primitive to utilize information of OpenType layout tables embedded in an OpenType font. An OpenType font can be a Graphite font only by duplicating that information by GDL.

Fig. 3 illustrates the Graphite system.

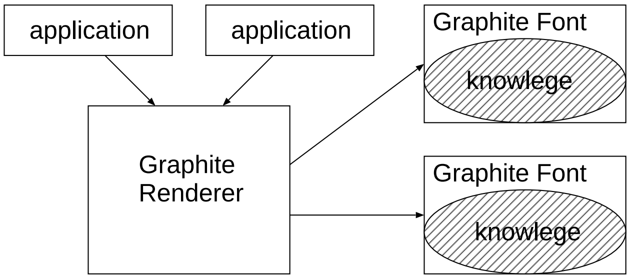

Figure 3: Graphite System and CTL knowledge

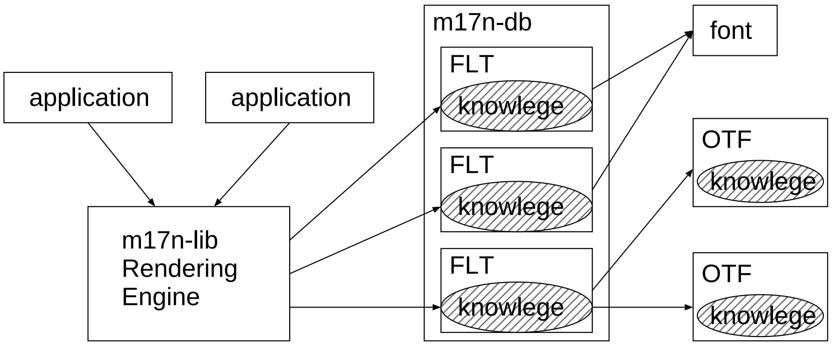

Figure 4: Font Layout Table and CTL knowledge

3.4 Font Layout Table

As we have seen above, embedding the knowledge in a renderer leads to inflexibility, and embedding it in a font leads to scattering of the common knowledge.

We thus propose a Font Layout Table. Font Layout Table is a resource added to a font to provide full intelligence for CTL. It represents the knowledge by a rule-base description similar to that of Graphite fonts, although the knowledge exists independent from fonts. FLT is not a font, nor a renderer, it bridges between them. That is, the single FLT can be used for driving multiple fonts, and multiple FLTs can drive the same font for multiple ways of writing text (e.g. with ancient style ligatures or by modern style ligatures).

In addition, Font Layout Table can be used with any kind of fonts including OTF. When FLT is used with OTF, FLT describes only the knowledge common to a script, and the knowledge specific to a font resides in the font as OpenType layout table. Fig. 4 illustrates the rendering system using Font Layout Table.

4 Font Layout Table

4.1 Overview

The m17n library has a database called "the m17n database" and it stores character/script/writing system specific knowledge. Font Layout Tables are also kept in the m17n database.

The renderer of the m17n library is equipped with the FLT driver. The driver reads a FLT from the m17n database and converts a character sequence to a glyph sequence with 2-dimensional positioning information according to the FLT.

The conversion process that the FLT specifies can be multiple cascaded stages. In each stage, a sequence of characters or glyphs is converted into another sequence using rules defined for the stage, and the new sequence is passed to the next stage. The length of sequences may differ from stage to stage, because a rule may replace multiple elements with one element, or may do the opposite. Each stage can have a "category table" that assigns a category code to each element of the sequence. The "category table" is mandatory in the first stage.

In this section, we briefly describe how this conversion is performed. The full syntax and semantics of FLT is described in the appendix.

4.2 Conversion process

When drawing a text, the renderer first determines one font and one FLT for each character in the text. For each subsequence of characters that share the same font and FLT, the renderer invokes the FLT driver to generate a glyph sequence corresponding to the subsequence.

The FLT driver at first generates an intermediate glyph sequence from the character subsequence. Each element in the glyph sequence contains the following information.

- code

  In the first stage of the conversion, "code" is the character code in the original character sequence. In the last stage, it is the glyph code passed to the font driver. In other stages, "code" is an arbitrary intermediate code.

- category

  "category" is a Latin alphabet `A` to `Z` and `a to` \z+ given by the "category table" of the current stage. "category" is used in a regular expression matching as described in the next subsection.

- combining-spec

  This is an integer value specifying how to combine the glyph with the previous one. Initially the value is zero, which means not to combine the glyph.

The FLT driver then executes a conversion rule of each stage as a cascaded manner. If a stage has its own "category table", "category" of each element is updated. The glyph sequence generated by the last stage is given back to the renderer.

4.3 Conversion rules

Conversion rules in Font Layout Table must be powerful and flexible enough to write all the conversion required. They should also be written with ease and efficiency.

A rule is applied to the "current run" that is the whole glyph sequence in the top-level rule of a stage.

FLT rules are defined as below:

```
RULE ::= REGEXP-BLOCK | MATCH-BLOCK | SUBSEQ-BLOCK | RANGE-BLOCK
       | COND-BLOCK
       | DIRECT-CODE | COMBINING-SPEC | OTF-SPEC | PREDEFINED-RULE
       | MACRO-NAME
```

The first four rules have the similar structure:

```
'(' MATCHER RULE * ')'
```

`MATCHER` specifies a matching part of the current run. If a matching part is found, the current run is restricted to the matching part, and the sub-rules are executed one by one regardless of they succeed or not. At last, the matching part is excluded from the current run.

In `REGEXP-BLOCK`, the matcher is a regular expression. The regular expression should match with the beginning or the string generated by concatenating "categories" of the current run. If it matches, the corresponding subsequence is the matching part. Otherwise, the rule fails.

In `MATCH-BLOCK`, the matcher is an integer specifying a parenthesized subexpression recorded by the previous `REGEXP-BLOCK`. If such a subexpression has been found by the previous regular expression matching, the corresponding subsequence is the matching part. Otherwise, the rule fails. If there is no previous regular expression matching (e.g. in such case that this is the top-most rule), the matcher must be 0 that matches the whole current run.

In `SUBSEQ-BLOCK`, the matcher is a sequence of "code". If the "codes" are the same as the beginning of the current run, the corresponding part is the matching part. Otherwise, the rule fails.

In `RANGE-BLOCK`, the matcher has the following form:

```
'(' 'range' FROM-CODE TO-CODE ')'
```

It specifies the range of the first "code" of the current run. If the first "code" is in the specified range, the first glyph is the matching part. Otherwise, the rule fails. While executing the sub rules, the difference of the first "code" and `FROM-CODE` is remembered as `CODE-OFFSET` that affects the behavior of the rule `DIRECT-CODE`.

COND-BLOCK provides conditional application of the rule. It has one or more rules and sequentially executes them one by one until one of them succeeds. If no rule succeeds, the rule fails.

DIRECT-CODE is an integer, and generates a new glyph whose "code" is DIRECT-CODE. If the rule is a sub-rule of RANGE-BLOCK, "code" is DIRECT-CODE plus CODE-OFFSET.

COMBINING-SPEC is an integer specifying how to combine the next glyph with the previous one. The combination is defined with five values, VPOS, HOPS, OFFSET, VPOS2, and HPOS2.

The combination of VPOS and HPOS specifies the vertical and horizontal reference points of a glyph. Reference points is one of the twelve points of a glyph, shown in Fig. 6. The VPOS and HPOS specifies the reference point of the previous glyph, and VPOS2 and HPOS2 specify that of the next glyph. The next glyph is drawn so that these two reference points align. OFFSET specifies the way of alignment in detail. If it is '.', the reference points are on the same position. Once the next glyph is combined with the previous one, they are treated as a single combined glyph.

OTF-SPEC specifies an instruction to the OTF driver.

PREDEFINED-RULE performs basic functions such as producing the same glyph as in the source, repeating rules, or setting padding flags.

MACRO-NAME specifies a macro defined in the stage.

5 CTL by FLT

In this section we describe how FLT typically performs each task in Complex text layout.

1. Clustering

   A cluster is extracted by specifying a regular expression in a pattern part of a rule. For instance, in the case of Thai script, by assigning category C to consonants, V to upper or lower vowels, and T to tone marks, a cluster can be extracted by the conditional shown in the following rule.

```
(cond
  ("C(VT?|T)"
   ... rules for the cluster containing combining characters ...)
  ("."
   ... rules for the single character cluster ...))
```

2. Reordering

   Source sequence can be reordered using the rule REGEXP-BLOCK that contains MATCH-BLOCK sub-rules. For instance, in the case of Devanagari script, the vowel I must be placed before the preceding consonant(s) in the cluster as shown in Fig. 5. By assigning category C to consonant, H to Halant, I to the vowel I, V to the other dependent vowels and M to the other combining marks (e.g. vowel modifiers), a Devanagari syllable ("cluster") is represented by the regular expression (CH)*C[IV]M* [1]. The following rule can move the vowel I to the head, once the match is found.

```
("((CN?H)*CN?)(I)(M*)"
 (3 =)
 (1 = *)
 (4 =))
```

[1] This is a simplified explanation to give a rough image. In a practical code, we have to consider Nukta sign, Reph, etc. For more detail, see the page: http://www.microsoft.com/typography/otfntdev/indicot/shaping.htm

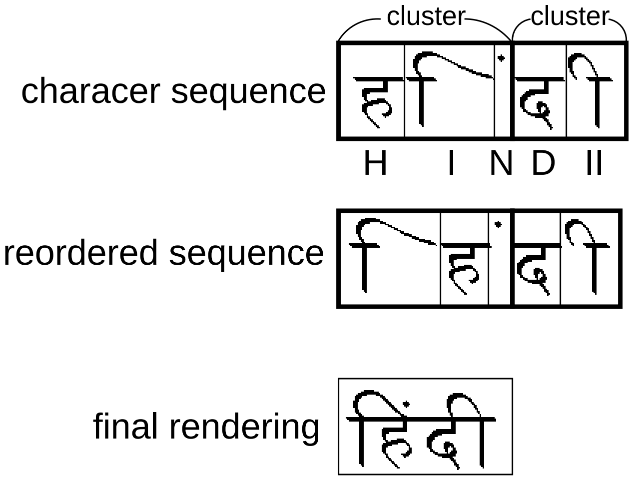

Figure 5: Example of Devanagari script "HINDI"

3. Glyph Mapping

   Character code to glyph code mapping can be performed by a sequence of `SUBSEQ-BLOCK` rules. For instance, if an input sequence is a Thai text of Unicode character codes, and the font encoding is TIS620, the following rule does the mapping.

```
(cond
 ((0xE01) 0xA1)
 ((0xE02) 0xA2)
 ...)
```

   In this case, we can also use `RANGE-BLOCK` rules as below, which is more efficient.

```
((range 0xE01 0xE5F) 0xA1))
```

4. Glyph Substitution

   Substitution of glyphs can be realized by various ways utilizing various rules. Here we show an example of using a `SUBSEQ-BLOCK` rule for the first part of Tibetan scripts སྐད་ ("kää" which means "language").

```
(cond
  ...
  ((0xF66 0xF90) 0xC7)
  ...)
```

   This rule converts the sequence ས྄ཀ into the single precomposed glyph སྐ.

5. Glyph Positioning

   2-dimensional positioning of a glyph is realized by `COMBINING-SPEC`. A glyph is assigned twelve reference points as shown in Fig. 6. A `COMBINING-SPEC` rule instructs how to combine the next glyph with the

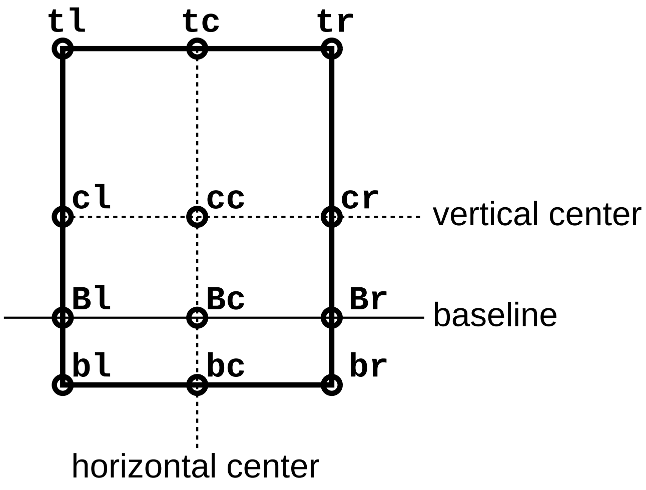

Figure 6: Reference points of a glyph

previous one.

For instance, the combining-spec `tc.bc` means to combine two glyphs so that the reference point `tc` of the previous glyph and the reference point `bc` of the following glyph align exactly.

6 Conclusion

Complex Text Layout requires much knowledge for processing text, scripts and writing systems. Such knowledge, however, is often distributed in text renderers and fonts inconsistently, which makes it hard to add a support for new scripts or fonts, or to use fonts of different formats.

We have proposed and implemented a new system for storing the knowledge: Font Layout Table. Font Layout Table provides knowledge about writing systems of different scripts and languages, and bridges the text renderer of the m17n library and various kinds of fonts.

In the m17n library[2], we have succeeded in making FLTs for Thai, Lao, Devanagari, Malayalam, Tamil, Tibetan, Khmer, Arabic, and Hebrew.

The future works on FLT are as below:

- Control from the render

  Currently, we don't have a mechanism to control the behaviour of FLT from the render. So, if the renderer requires a different layouting, the different FLT must be created and used from the renderer. It may be convenient if a rule of FLT can be run differently depending on some flag set by the renderer.

- More powerful primitives?

  FLT doesn't have such primitives that are usually required for a programming language, for instance, variables, arithmetic calculation, and etc. That's is to make the FLT driver simpler, but, we may face with a case that requires more powerful primitives. We must investigate more scripts to answer the question.

[2] The m17n library will be soon released as open source software at: `http:/www.m17n.org/m17n-lib/`

Appendix A

The full syntax of FLT is as below.

```
FONT-LAYOUT-TABLE ::= STAGE0 STAGE *

STAGE0 ::= CATEGORY-TABLE GENERATOR

STAGE ::= CATEGORY-TABLE ? GENERATOR

CATEGORY-TABLE ::= '(' 'category' CATEGORY-SPEC + ')'

CATEGORY-SPEC ::= '(' CODE CATEGORY ')' | '(' CODE CODE CATEGORY ')'

CODE ::= INTEGER

CATEGORY ::= INTEGER
```

In the definition of `CATEGORY-SPEC`, `CODE` is a glyph code, and `CATEGORY` is an ASCII code of an upper or lower letter, i.e. one of `A`, ... `Z`, `a`, .. `z`.

The first form of `CATEGORY-SPEC` assigns `CATEGORY` to a glyph whose code `CODE`. The second form assigns `CATEGORY` to glyphs whose code falls between the two `CODE`s.

```
GENERATOR ::= '(' 'generator' RULE MACRO-DEF * ')'

RULE ::= REGEXP-BLOCK | MATCH-BLOCK | SUBSEQ-BLOCK | RANGE-BLOCK
         | COND-BLOCK
         | DIRECT-CODE | COMBINING-SPEC | PREDEFINED-RULE | OTF-SPEC
         | MACRO-NAME
```

Each `RULE` specifies glyphs to be consumed and glyphs to be produced. When some glyphs are consumed, they are taken away from the current run. A rule may fail in some condition. If not described explicitly to fail, it should be regarded that the rule succeeds.

```
REGEXP-BLOCK ::= '(' REGEXP RULE * ')'
```

`REGEXP` is a regular expression that should match the sequence of categories of the current run. If a match is found, this rule executes `RULE`s while temporarily limiting the current run to the matched part. The matched part is consumed by this rule.

Parenthesized subexpressions, if any, are recorded to be used in `MATCH-BLOCK` that may appear in one of `RULE`s.

If no match is found, this rule fails.

```
MATCH-BLOCK ::= '(' MATCH-INDEX RULE * ')'
```

MATCH-INDEX is an integer specifying a parenthesized subexpression recoded by the previous `REGEXP-BLOCK`. If such a subexpression was found by the previous regular expression matching, this rule executes `RULE`s temporarily limiting the current run to the matched part of the subexpression. The matched part is not consumed by this rule.

If no match was found, this rule fails.

If this is the first rule of the stage, `MATCH-INDEX` must be 0, and it matches the whole current run.

```
SUBSEQ-BLOCK ::= '(' '(' CODE + ')' RULE * ')'
```

If CODEs matches the sequence of codes of the current run, this rule executes RULEs while temporarily limiting the current run to the matched part. The matched part is consumed.

If they don't match, this rule fails.

```
RANGE-BLOCK ::= '(' '(' 'range' FROM-CODE TO-CODE ')' RULE * ')'
```

If the first glyph code of the current run is in the range FROM-CODE to TO-CODE, this rule sets the default code-offset to the first glyph code minus FROM-CODE, and executes RULESs while temporarily limiting the current run to the first glyph. The first glyph is consumed.

If the first glyph is not in the specified range, this rule fails.

```
COND-BLOCK ::= '(' 'cond' RULE + ')'
```

This rule sequentially executes RULEs until one succeeds. If no rule succeeds, this rule fails.

```
DIRECT-CODE ::= INTEGER
```

This rule consumes no glyph and produces a glyph that has the following attributes:

- code : INTEGER plus the default code-offset
- combining-spec : default value
- left-padding-flag : default value
- right-padding-flag : zero

After having produced the glyph, the default code-offset, combining-spec, and left-padding-flag are all reset to zero.

```
COMBINING-SPEC ::= SYMBOL
```

COMBINING-SPEC is a symbol whose name specifies how to combine the next glyph with the previous one. This rule sets the default combining-spec to an integer code that is unique to the symbol name. The name has the following syntax.

```
COMBINING-NAME ::= VPOS HPOS OFFSET VPOS HPOS

VPOS ::= 't' | 'c' | 'b' | 'B'

HPOS ::= 'l' | 'c' | 'r'

OFFSET :: = '.' | XOFF | YOFF XOFF ?

XOFF ::= ('<' | '>') INTEGER ?

YOFF ::= ('+' | '-') INTEGER ?
```

The combination of VPOS and HPOS specifies the vertical and horizontal reference points of a glyph as shown in the Fig. 6.

The first VPOS and HPOS in the definition of COMBINING-NAME specifies the reference point of the previous glyph, and the second VPOS and HPOS specify that of the next glyph. The next glyph is drawn so that these two reference points align.

OFFSET specifies the way of alignment in detail. If it is '.', the reference points are on the same position.

XOFF specifies how much the X position of the next reference point should be shifted to the right (<) or left (>) from the previous reference point.

YOFF specifies how much the Y position of the next reference point should be shifted upward (+) or downward (-) from the previous reference point.

In both cases, INTEGER is the amount of shift expressed as a percentage of the font size, i.e., if INTEGER is 10, it means 10% (1/10) of the font size. If INTEGER is omitted, it is assumed that 5 is specified.

Once the next glyph is combined with the previous one, they are treated as a single combined glyph.

```
PREDEFINED-RULE ::= '=' | '*' | '<' | '>' | '|' | '[' | ']'
```

Each rule performs an action described below.

- `=`

  This rule consumes the first glyph in the current run and produces the same glyph. It fails if the current run is empty.

- `*`

  This rule repeatedly executes the previous rule. If the previous rule fails, this rule does nothing and fails.

- `<`

  This rule specifies the start of a grapheme cluster. No glyph is consumed, no glyph is produced.

- `>`

  This rule specifies the end of a grapheme cluster. No glyph is consumed, no glyph is produced.

- `[`

  This rule sets the default left-padding-flag to 1. No glyph is consumed, no glyph is produced.

- `]`

  This rule changes the right-padding-flag of the lastly generated glyph to 1. No glyph is consumed, no glyph is produced.

- `|`

  This rule consumes no glyph and produces a special glyph whose category is ' ' and other attributes are zero. This is the only rule that produces that special glyph.

```
OTF-SPEC ::= SYMBOL
```

OTF-SPEC is a symbol whose name specifies an instruction to the OTF driver. The name has the following syntax.

```
OTF-SPEC-NAME ::= 'otf:' SCRIPT LANGSYS ? GSUB-FEATURES ? GPOS-FEATURES ?

SCRIPT ::= SYMBOL

LANGSYS ::= '/' SYMBOL

GSUB-FEATURES ::= '=' FEATURE-LIST ?

GPOS-FEATURES ::= '+' FEATURE-LIST ?

FEATURE-LIST ::= ( SYMBOL ',' ) * [ SYMBOL | '*' ]
```

Each SYMBOL specifies a tag name defined in the OpenType specification.

For SCRIPT, SYMBOL specifies a Script tag name (e.g. “deva” for Devanagari).

For LANGSYS, SYMBOL specifies a Language System tag name. If LANGSYS is omitted, the Default Language System table is used.

For GSUB-FEATURES, each SYMBOL specifies a GSUB Feature tag name to apply. ’*’ is allowed as the last item to specify all remaining features. If SYMBOL is preceded by ’ ’ and the last item is ’*’, SYMBOL is excluded from the features to apply. If no SYMBOL is specified, no GSUB feature is applied. If GSUB-FEATURES itself is omitted, all GSUB features are applied.

The specification of GPOS-FEATURES is analogous to that of GSUB-FEATURES.

See <http://www.microsoft.com/typography/otspec/default.htm> for the OpenType specification.

Appendix B

This is a Font Layout Table for Devanagari OTF fonts.

```
(category
 ;; C: consonant (except for R)
 ;; R: consonant RA
 ;; n: NUKTA
 ;; H: HALANT
 ;; m: MATRA (pre)
 ;; u: MATRA (above)
 ;; b: MATRA (below)
 ;; p: MATRA (post)
 ;; A: vowel modifier (above)
 ;; a: vowel modifier (post)
 ;; S: stress sign (above)
 ;; s: stress sign (below)
 ;; V: independent vowel
 ;; N: ZWNJ (ZERO WIDTH NON-JOINER)
 ;; J: ZWJ (ZERO WIDTH JOINER)
 ;; E: ELSE
 ;;
 (0x0900 0x097F ?E)                    ; ELSE
 (0x0901        ?A)                    ; SIGN CANDRABINDU (above)
 (0x0902        ?A)                    ; SIGN ANUSVARA (above)
 (0x0903        ?a)                    ; SIGN VISARGA (post)
 (0x0905 0x0914 ?V)                    ; LETTER A .. LETTER AU
 (0x0915 0x0939 ?C)                    ; LETTER KA .. LETTER HA
 (0x0930        ?R)                    ; LETTER RA
 (0x093C        ?n)                    ; SIGN NUKTA
 (0x093D        ?E)                    ; SIGN AVAGRAHA
 (0x093E 0x094C ?p)                    ; VOWEL SIGN (post)
 (0x093F        ?m)                    ; VOWEL SIGN I (pre)
 (0x0941 0x0944 ?b)                    ; VOWEL SIGN (below)
 (0x0945 0x0948 ?u)                    ; VOWEL SIGN (above)
 (0x094D        ?H)                    ; SIGN VIRAMA (HARANT)
 (0x0950        ?E)                    ; OM
 (0x0951 0x0954 ?S)                ; STRESS SIGN or TONE MARK (above)
 (0x0952        ?s)                ; STRESS SIGN or TONE MARK (below)
 (0x0958 0x095E ?C)                    ; LETTER QA .. LETTER YYA
 (0x0960        ?V)                    ; LETTER VOCALIC RR
 (0x0961        ?V)                    ; LETTER VOCALIC LL
 (0x0962 0x0963 ?b)                    ; VOWEL SIGN (below)
 (0x0964 0x0970 ?E)                    ; DANDA .. ABBREVIATION SIGN
 (0x200C        ?N)                    ; ZWNJ
 (0x200D        ?J)                    ; ZWJ
 (0x097E        ?x)            ; internally used tag to indicate Reph
 (0x097F        ?y)            ; internally used tag to indicate Mpost
```

```
 )

;; The first stage is to extract a syllable and re-order characters in
;; it.
(generator
 (0
  (cond
   ;; If [CR]H is followed by ZWNJ/ZWJ, move ZWNJ/ZWJ) to the head so
   ;; that the later stages find it quickly.
   ("([CR]n?H)([NJ])"
    < | (2 =) (1 = =) | > )

   ;; A syllable starting with RH (Reph) and ending with a vowel
   ;; and/or a vowel modifier.
   ("(RH)(([CR]n?H)*[CR]n?)([mubp][Aa]?[Ss]?)"
    < | (1 0x097E = =) (2 move-base-Halant) (4 reorder-post-base) | >)

   ;; A syllable starting with RH (Reph) and ending without a vowel
   ;; nor a vowel modifier.
   ("(RH)(([CR]n?H)*[CR]n?)(H)?"
    < | (1 0x097E = =) (2 move-base-Halant) (4 =) | >)

   ;; A syllable starting with the other consonant and ending with a
   ;; vowel and/or a vowel modifier.
   ("(([CR]n?H)*[CR]n?)([mubp][Aa]?[Ss]?|[Aa][Ss]?|[Ss])"
    < | (1 move-base-Halant) (3 reorder-post-base) | >)

   ;; A syllable starting with the other consonant and ending without
   ;; a vowel nor a vowel modifier.
   ("(([CR]n?H)*[CR]n?)(H)?"
    < | (1 move-base-Halant) (3 =) | >)

   ;; A syllable starting with an independent vowel.
   ("V[Aa]?[Ss]?"
    < | = * | > )

   ("." =))
  *)

 ;; Move Halant on a base consonant to the tail.
 (move-base-Halant
  (cond
   ("(([CR]n?H)*[CR]n?)(H)(R)"
    (1 = *)
    (4 =)
    (3 =))
   (".*"
    = *)))
```

```
 ;; Re-order post modifiers.
 (reorder-post-base
  ;; 12   3   4   5    67   8     9   10
  ("((m)|(u)|(b)|(p))?((A)|(a))?((S)|(s))?$"
   (2 =) (4 =) (10 =) (3 =) (5 =)
   0x097F
   (7 =) (9 =) (8))))

;; The second stage is to reorder Reph and Mpre.
(generator
 (0
  (cond
   (" [NJ]([^ ]*) "
    = *)
   (" (x(..))([CRnH]*) "
    | (3 = *) (2 otf:deva=rphf) |)
   (" (x(..))([CRnH]*)(m?)([^y]*)y([^ ]*) "
    | (4 =) (3 = *) (5 = *) (2 otf:deva=rphf) (6 = *) |)
   (" ([CRnH][CRnH]*) "
    = *)
   (" ([CRnH][CRnH]*)(m?)([^y]*)y([^ ]*) "
    | (2 =) (1 = *) (3 = *) (4 = *) |)
   ("." =))
  *))

;; The third stage is to drive OTF tables.  For the moment, we use
;; the default LangSys, and try all GSUB/GPOS features except for the
;; sequence followed by ZWNJ in which case try "nukt" and "haln"
;; features only.
(generator
 (0
  (cond
   (" N([^ ]*) "
    (1 otf:deva=nukt,haln))

   (" J([^ ]*) "
    (1 otf:deva))

   (" ([^ ]*) "
    (1 otf:deva=~rphf,*))

   ("."
    [ otf:deva=+ ] ))
  *))
```