m17n ライブラリにおける表示機能の実現
The display engine of the m17n library

半田剣一、錦見美貴子、高橋直人、戸村哲
産業技術総合研究所 情報処理研究部門

1 はじめに

ソフトウェアの国際化および多言語化は、世界中の人たちがネットワークを経由して情報発信するようになっている現在、避けて通ることのできない課題である。人間とのインターフェースを必要とするアプリケーションソフトウェアのほとんどで、自然言語、それも文字が用いられている。しかし、多言語化部分をまとめて処理するライブラリがないために、個々のソフトウェアで個別に言語やスクリプト処理の部分を作り、個別に多言語化機能を開発するという非効率な事態が起こっている。Emacs, Mozilla, Perl などもその例である。

一方、ソフトウェア内部で多言語対応を必要とする部分は、他の部分から比較的容易に切り出せる場合が多い。また、どのようなアプリケーションであっても、文字を用いたインタフェースの部分において必要とされる機能はほぼ共通している。入力や表示などはその典型である。そして、多くのソフトウェアが、そのような機能の多言語化を必要としている。そこで、多言語化機能を集約して共通に利用できるようにした多言語化ライブラリを開発すれば、多くのソフトウェアを効率的に開発するために役に立つと考えられる。

本稿では、このような考えにそって開発中である多言語化ライブラリ m17n [1] ライブラリについて述べる。m17n ライブラリの機能は多岐に渡るが、特にその中の表示機能に重点をおく。まず m17n ライブラリ全体の概要とついで多言語の表示にはどのような問題があるか述べる。その上で、m17n ライブラリではそれらをどのように解決しようとしているかを、実例を含めて説明する。

[1] m17n=multilingualization

2 m17n ライブラリ

m17n ライブラリは Unix/Linux 上の一般のアプリケーションから使用できる、汎用の多言語ライブラリであり、C ライブラリと X ライブラリのテキスト処理部分と差し換えて利用されることによって、多言語化機能を提供することを目指している。

m17n ライブラリはライブラリ本体とは別に言語情報ベースと呼ばれるデータベースを持ち、言語やスクリプトに固有の情報はここに隔離して保持することを想定している。ライブラリの本体は言語情報ベースの情報を参照しながら動作するが、個別の情報とは独立に設計と開発を行なっている。

たとえば

- サポートする文字セットやコード系の定義
- どの言語のためにどの入力メソッドをサポートするか
- どのスクリプトはどのフォントで表示するか
- どのフォントはどうエンコードされ、どう表示するべきか

といった情報はすべて言語情報ベースに収められる。

m17n ライブラリの利用者、すなわちアプリケーションの開発者が m17n ライブラリの機能を拡張したい場合は、このデータベースに情報を追加することになる。

m17n ライブラリの特徴は、操作対象となる多言語のテキストを表わす構造体として M-text と呼ぶ属性付文字列を用いる点にある。M-text は、その全体ないし任意の部分に対して、テキストプロパティと呼ぶ任意の属性-値ペアを付加することができる。表示において必要となる書字方向やタイプフェースなどの情報も、テキストプロパティとして表現される。テキス

トプロパティと Pango のテキスト属性は、表示に関する点ではほぼ同じように利用することができる。

m17n ライブラリは平成 13 年度には「情報処理開発支援事業」として開発し、また平成 14 年度は「情報技術継続事業」として情報処理振興事業協会 (IPA) からの支援を受ける予定であり、オープンソースソフトウェアとしての公開を前提として開発している。

3 多言語の表示における問題

テキスト処理において、表示は入力と並ぶ重要な要素である。多言語対応を考える場合には特に、以下のような問題を解決する必要がある。

3.1 フォントの選択

世界のすべてのスクリプトを表示できるフォントはいまだ存在していない。そのため、多言語対応を考える場合には、システムには複数のフォントが準備されており、その中から個々のスクリプトあるいはスクリプトのグループ毎に、それを表示することができるもの、可能ならば高品質に表示できるものを選択していくというモデルを取らざるを得ない。

Unicode フォント (registry: iso10646-1) と呼ばれるフォントが準備されていたとしても、それらのフォントは多くの場合 Unicode の一部のみしかサポートしていない。そこで、フォント自身の中身を調べることによって、利用したいスクリプトが表示できるかを調べ、表示できるものを選択していく必要がある。つまり「Unicode フォント」によってもフォント選択の問題は解決していない。

3.2 複雑な表示を必要とするスクリプトへの対応

非ラテン系のスクリプトでは特に、フォントからキャラクタコードに対応するグリフを選んで順番に並べるというだけのやり方では、正しい表示にはならないものが多い。受容できるレベルの表示を行なうためには、

- 代替グリフ
- リガチャ（ラテン系の場合にはおおむね高品質な組版を得るためのオプションであるのに対し、これが必須である言語も多い。）
- メモリ内の論理的な文字の順序からディスプレイ上の視覚的なグリフの順序への変更

などのさまざまな処理が要求される。

このような特別な処理の具体的な内容は、言語単位、スクリプト単位で異なっている。そこで、多言語の表示機能を設計する際には、言語やスクリプトに独立なコアの部分と各言語／スクリプト固有のモジュールを分離するというやり方が採用される。この方針のもとで問題となるのは、

> どこまでを「言語やスクリプトに固有」とするか

という点である。この問題は、表示に必要な情報を表示システムのどの部分が持つのか、つまりフォントなのか、表示システムのモジュールなのか、また必要な情報は表示機能の設計者がすべて提供するのか、利用者が追加できるのかに影響される。

モジュールに多くの情報を抱え、モジュールの設計に自由度を残した方法では、新たなスクリプトをサポートする際にはそのスクリプト専用のモジュールを最初から作る必要がある。したがって、新たなスクリプトやフォントを追加する際の手数が多く、敷居が高い。

しかし非ラテン系スクリプトへの対応では、この個々のスクリプトごとにハードコードされたモジュールを利用するやり方が多い。そして、これらのモジュールは特定のエンコーディングを持ったフォントのみを対象としているため、異なるエンコーディングのフォントを使うには、別のモジュールが必要となる。

たとえば Pango ではスクリプトごと、フォントのエンコーディングごとに表示ルーティンがハードコードされている。 Thai スクリプトの場合を見てみよう。Pango はこのスクリプトを `thai-x.c` というプログラムで処理している。このプログラムは TIS, TIS_WIN, TIS_MAC, XTIS, ISO10646 の５種類のエンコーディングをサポートし、それぞれのために必要なコードが直接書かれている。

また、インド系の Devanagari スクリプトに関しては特定のエンコーディングのフォント (registry: iso10646-dev) のみをサポートするコードを `devanagari-x.c` にハードコードし、このエンコーディングにしたがった X フォントを同時に配布して

いる。この X フォントは pango の表示エンジンが特別に解釈する以下のようなプロパティ

```
PANGO_LIGATURE_HACK
   ":*:PANGO_LIG1 :*:PANGO_LIG2"
PANGO_LIG1 "F915+0937=D000..."
PANGO_LIG2 "0930+0941=D01E..."
```

を持っており、この情報を使ってリガチャを行う。したがって現状では Pango で Devanagari スクリプトを表示するためには、このフォントしか許されないことになる。

このように、モジュール側に多くの自由度を持つ設計のもとで、各スクリプトの各フォントエンコーディング用のモジュールを書くためには、そのスクリプトでのリガチャや代替グリフに関する知識だけではなく、表示の詳細にわたる知識が要求される。もちろん Pango のようにモジュールを含めた表示システムを提供しようとする場合にはそれでかまわない。個々のスクリプトに関する知識を持つ開発者がそのスクリプトに関するモジュールを作ることで、高品質なモジュールが平行して作られ、開発のスピードアップが期待できる。しかし、アプリケーションの開発者が、自分の必要な言語やスクリプトを自力で追加しようとする場合には困難が予想される。

4 m17n ライブラリの表示機能

4.1 表示機能の概要

m17n ライブラリでは、新たなスクリプトやフォントを、ライブラリの本体に変更を加えることなく、簡単に追加して利用できる枠組みを提供しようとしている。このため、まずフォント選択は、言語情報ベースに登録されたフォントセットに従って柔軟になされるように設計した。また、テキスト表示機能は、前述のようにスクリプトやフォント毎の処理をハードコーディングすることは避け、単一の汎用レイアウト・エンジンと、言語情報ベースからロードするスクリプト毎の表示ルールを組み合わせて実現している。このフォント／スクリプト毎の表示ルールをフォントコンポーザと呼び、レイアウト・エンジンはフォントコンポーザのインタプリタとして働く。

この設計により、新たなスクリプトやフォントをサポートする際にも、必要な情報を言語情報ベースのフォントセットやフォントコンポーザに登録するだけですむ。

この節ではこれらの処理の詳細を述べる。

4.2 フォント選択機能

m17n ライブラリでは、言語情報ベースに定義されるフォントセットにしたがってフォントを選択する。フォントセットには各スクリプト毎に使用するフォントのスペック(`family`, `adstyle`, `registry`)を複数登録する。

例えば Hangul スクリプトを表示する際には、以下のようになる。

```
(hangul
 (nil
  (nil nil "ksc5601.1987-0")
  ("*-baekmuk batang" nil "iso10646-1")))
```

この例では３、４行目に２種類のフォントが登録されている。２行目の `nil` は設定が言語に関係なく有効であることを示す。m17n ライブラリのフォントセレクタはこれらのスペックにマッチするすべてのフォントのうち、現在表示しようとしている文字のグリフを持ち、現在のタイプフェース（M-text の ‘face’ 属性であらわされるサイズ、スタイル等）に最も適合するものを選ぶ。

あるフォントがどれだけの文字を表示できるかは、やはり言語情報ベースに登録されたフォントエンコーディングによって決定される。たとえば Han スクリプトで用いられるフォント用の情報は以下のようになる。

```
(nil nil "jisx0208.1983-0" jisx0208.1983 t)
(nil nil "jisx0212.1990-0" jisx0212 t)
(nil nil "gb2312.1980-0"   gb2312.1980 t)
(nil nil "ksc5601.1987-0"  ksc5601.1987 t)
(nil nil "big5.eten-0"     big5 t)
```

最初の３つの要素がフォントのスペック(`family`, `adstyle`, `registry`)を、４番目の要素がそのフォント内でのグリフのエンコーディングに対応する文字セットを、５番目の要素がそのフォントがサポートする文字のレパートリに対応する文字セットを示す。５番目の要素が `t` なら４番目の要素と同じであり、`nil` ならフォントをオープンして調べることを意味している。

たとえば Unicode フォント用には以下のように指定しておくことにより、本当に必要なスクリプトが含まれているかどうかをチェックさせることができる。

```
(nil nil "iso10646-1"  unicode nil)
```

こうしておくことによって、 Greek スクリプト用のフォントセットが次のように設定されていたとしても、

```
(greek
  (nil
   (nil nil "iso10646-1")
   (nil nil "iso8859-7")))
```

本当に Greek スクリプトを含んだ Unicode フォントがみつからない限り、iso8859-7 フォントが選択される。

また、スクリプトは同じであっても言語によってフォント選択を変えなければならない場合もある。漢字の表示はその典型例である。そのためには、以下のように Han スクリプト用には言語毎に優先するフォントを登録すればよい。

```
(han
 (ja
  (nil nil "jisx0208.1983-0")
  (nil nil "jisx0212.1990-0"))
 (zh
  (nil nil "gb2312.1980-0")
  (nil nil "big5.eten-0"))
 (ko
  (nil nil "ksc5601.1987-0")))
```

フォント選択機能は表示しようとする M-text の持つ ‘language’ 属性を調べ、その値が ‘ja’ であれば以下のどれかのスペックにマッチするフォントを試す。

```
(nil nil "jisx0208.1983-0")
(nil nil "jisx0212.1990-0")
```

もし、上記のスペックのフォントがどれも現在の文字を表示できない場合に、残りのいずれか、つまり

```
(nil nil "gb2312.1980-0")
(nil nil "big5.eten-0")
(nil nil "ksc5601.1987-0")
```

が試されることになる。

4.3 テキスト表示機能

レイアウト・エンジンは、前述の機能によりフォントを決定した上で、文字コード列をフォントのグリフコード列に変換する。その際、通常は単にフォントエンコーディングに指定された文字セットで各文字をエンコードするだけであるが、複雑な表示を必要とするスクリプトの場合はそれだけでは済まない。フォントセットの各エントリは４番目の要素として、先に述べたフォントコンポーザを指定することができる。フォントコンポーザが指定されている場合には、レイアウト・エンジンはそのフォントコンポーザをロードして、解釈し実行する。

たとえば前述の Thai スクリプトの表示も、５種類のフォント TIS, TIS_WIN, TIS_MAC, XTIS, ISO10646 のそれぞれのフォントコンポーザを記述することによって実現できる。各フォントコンポーザを言語情報ベースに登録し、さらにフォントセット定義のなかでどのフォントはどのフォントコンポーザの情報に基づいてドライブするかを記述する。Thai スクリプト用のフォントセット定義は例えば以下のようになる。

```
((thai
  (nil
   (nil nil "tis620-2" tis620-win)
   (nil nil "tis620-1" tis620-mac)
   (nil nil "tis620.2533-0" tis620)
   (nil nil "tis620.2529-1" tis620)
   (nil nil "iso8859-11" tis620)
   (nil nil "iso10646-1" thai-unicode)
   (nil nil "xtis620.2529-1" xtis620)
   (nil nil "xtis-0" xtis620))))
```

4.4 フォントコンポーザ

m17n ライブラリの表示機能は、単一の汎用レイアウト・エンジンが、各種フォントおよびスクリプトに対応するフォントコンポーザにしたがって、文字コード列をグリフコード列に変換し表示するという方法を採用している。この方法であらゆるスクリプトやフォントに対応するために、フォントコンポーザは柔軟性と高い記述能力を持つ必要がある。

一般に、文字コード列からグリフコード列を得るには、以下のような処理が必要となり、しかもそれぞれはコンテクストに依存した処理を必要とする。

1. 文字コード列を書記素 (grapheme) のクラスタに分割
2. クラスタ内の文字の並べ替え
3. 各文字を対応するグリフコードに変換
4. 必要なリガチャの処理
5. グリフの２次元的合成

フォントコンポーザでは、これらすべての処理を多段パスによって記述することができるようにした。すなわち、最初の文字コード列から中間的なコード列（文字コード、グリフコード、あるいは何らかの記号の列）を作り出し、それを繰り返すことによって最終的なグリフコード列を生成するのである。

この方法をとることによって、レイアウトに必要な変換ルールの記述がしばしば容易になる。たとえばアラビア語のスクリプトのように、前後に他の文字があるかどうかによってグリフが変わり、かつリガチャが存在するものを考えてみよう。このようなスクリプトに対しては、まず第一段階としてリガチャを処理し、ついで前後の文字による影響を処理するというルールを書くことができる。このルールは、キャラクタから最終的なグリフ列への変換を一度に行なうルールを与えるのに比べ、明確で簡単なものになる。

以下では、デバナガリスクリプト用の ISFOC[2] にしたがったフォントを例に取りながら、どのようにフォントコンポーザを記述するかを説明する。

4.4.1　ルール記述の対象：コード単位とカテゴリ単位

入力シークエンスはコード列であるので、ルールの一般的な形はコード（列）からコード（列）を生成するものとなる。しかし文字／グリフには、類似した特徴を持つものがあり、類似した文字／グリフのコードをまとめて一つのカテゴリとすることができる。そしてカテゴリに対するルールを許すことによって、必要なルールの数を大幅に減らすことが可能である。たとえば上付きの diacritical marks 一つ一つに対してルールを記述する代わりに、全部に共通のルールを一個だけ記述すれば、手間を大きく省くことができる。そこで、シークエンス生成のルールは、カテゴリとして抽象化されたレベルと、個々の文字またはグリフコードのレベルの両方で書くことができることとした。

[2] Indian Standard Font Code. インドにおける標準的なフォントエンコーディング。 `http://tdil.mit.gov.in/faq.htm`

カテゴリは変換の各パス（ステージと呼ぶ）ごとに定義し直すことができる。ステージごとに異なるカテゴリを設定し、異なるカテゴリを対象にして書かれたルールを用いることで、きめ細かなグリフ生成が容易に実装できる。

つまり、特定のフォントでグリフ生成を行なわせるためには、以下の２つのデータを与えればよいことになる。

1. コードとカテゴリの対応表（必要があればステージごとに異なっていて良い）
2. 各ステージの変換ルール（カテゴリレベル and/or コードレベル）

このデータは、フォントとスクリプトの種類ごとに以下の形式で記述される。

```
（ステージ１のカテゴリの定義
 ステージ１の生成ルール
 ステージ２のカテゴリの定義（必要ならば）
 ステージ２の生成ルール
 ステージ３のカテゴリの定義（必要ならば）
 ステージ３の生成ルール
 ...）
```

また、カテゴリは、各ステージにおいて入力のコードとして受けつけるの範囲の指定にも用いられる。したがって、ステージ１でカテゴリを定義された文字コードが、このドライバで表示できる文字のレパートリを示すことになる。

4.4.2　カテゴリの定義

カテゴリはグリフコードに対して定義される。ショートハンドとして、グリフコードの代わりにコードを二つ並べて記述することができ、この場合はそれらに挟まれた（両端を含む）範囲のすべてのグリフがそのカテゴリに属すことを示す。カテゴリは `A` から`Z`、`a` から`z` のうちの一文字でなければならない。これにより、入力シークエンスをカテゴリを使った正規表現でスキャンすることができるようになる。

図１にステージ１のカテゴリ定義を示す。

一つのグリフコードは一つのカテゴリにしか属さない。また、定義は後に現われたものによって上書きされるので、ある範囲に対して一つのカテゴリを定義し、

```
(category
 ;; A: ANUSVARA or CANDRABINDU
 ;; C: CONSONANT (except for R)
 ;; R: LETTER RA
 ;; V: INDEPENDENT VOWEL
 ;; N: NUKTA
 ;; H: HALANT
 ;; M: MATRA (DEPENDENT VOWEL, except for I)
 ;; I: VOWEL SIGN I
 ;; S: STRESS or TONE
 ;; E: ELSE
 (0x0901 0x0902  ?A) ; CANDRABINDU, ANUSVARA
 (0x0903         ?E) ; SIGN VISARGA
 (0x0905 0x0914  ?V) ; A .. AU
 (0x0915 0x0939  ?C) ; KA .. HA
   (0x0930       ?R) ; RA
 (0x093C         ?N) ; NUKTA
 (0x093D         ?E) ; AVAGRAHA
 (0x093E 0x094C  ?M) ; VOWEL SIGN AA .. AU
   (0x093F       ?I) ; VOWEL SIGN I
 (0x094D         ?H) ; SIGN VIRAMA (HARANT)
 (0x0950         ?E) ; OM
 (0x0951 0x0954  ?S) ; UDATTA .. ACUTE
 (0x0958 0x095E  ?C) ; LETTER QA .. YYA
 (0x0960 0x0961  ?V) ; LETTER VOCALIC RR, LL
 (0x0962 0x0963  ?M) ; VOWEL VOCALIC L, LL
 (0x0964 0x0970  ?E)); DANDA...
```

図 1: デバナガリスクリプトのカテゴリ定義

その範囲のグリフコードのうち例外的なものだけに対して別のカテゴリを再定義することができる。

たとえば図 1 の 15 から 16 行目で、0x0915 から0x0939 までのコードを持つキャラクタのカテゴリは ?C （普通の子音）であること、ただし 0x0930 は ?R（RA 文字）に定義しなおされていることがわかる。

4.4.3 ルール記述の方法：正規表現、マッチインデックス、サブルーティン

シークエンスの生成ルールは(pattern command) という形式をとり、pattern に合致するソースシークエンスを受け取った場合に、command を実行してターゲットシークエンスを生成する。

pattern は、

1. カテゴリに基づく正規表現
2. 入力コードの範囲あるいは入力コードの列
3. 正規表現中でのマッチした部分を表すインデックス

のいずれかである。

正規表現中には、そのステージまでに定義されているカテゴリを使うことができる。例えば第 1 ステージでの "VA?S?" という正規表現は、独立母音、もしあればそれに続く CANDRABINDU もしくは ANUSVARA、さらにもしあればそれに続く STRESS 記号、というコード列を表す。

入力コードまたは入力コードの列はかっこでくくって表わし、たとえば生成ルール

```
((0x0905) 0x2B)
```

は「0x0905 の入力コードを受け取った際には 0x2B を生成する。」ことを、また、

```
((0x090D 0x0901) 0x42 0xC4)
```

は、「0x090D 0x0901 という入力コード列を受け取った際には0x42 0xC4 というコード列を生成する。」ことを意味する。

正規表現中でマッチした部分を表すマッチインデックスも、pattern として使うこともできる。マッチインデックスは、入力コード列をいくつかの部分に分割し、それぞれの部分に対して異なる生成ルールを適用する必要がある場合、たとえばコンテクストに依存する変換を記述する場合などに便利である。

次の例を見てみよう。

```
("(RH)([^I]*)(I)(A?S?)"
 (3 vowel-sign-I)
 (2 consonant *)
 (1 preceding-r)
 (4 post-modifier))
```

この例は、"(RH)([^I]*)(I)(A?S?)" という正規表現でpattern が記述されている。コマンド部 (2–5 行目) では、それぞれ再帰的に生成ルールが現われている。この再帰的に現われるルールのpattern として、1..4 のマッチインデックスが使われている。このルール全体は、入力コード列を (RH)、([^I]*)、(I)、(A?S?) にそれぞれマッチする 4 つのグループに分解し、3,2,1,4

という順序で各グループを処理することを示している。そして各グループ内の処理は、`vowel-sign-I`、`consonant` 等のcommand（実際には後述するコマンドマクロ）に任される。

このようなルールにより、

क ि◌ → ि◌क → कि

や

र ◌् ग ◌् घ → ग ◌् घ ◌र्‍ → र्ग्घ

のような、コンテキストに依存したグリフの並べ替えの処理が可能となる。

一方、command は

1. 生成ルール（再帰的に）
2. 条件節
3. ターゲットシークエンス
4. コマンドマクロの名前
5. 特別なコマンド

が必要なだけ現われることができる。複数のコマンドがある場合には、先頭のものから順に実行し、入力コード列のうちコマンド実行の対象になった部分を取り除いてから次のコマンドを適用する。

条件節は、入力コード列が特別な条件に合致した場合に実行されるコマンドを指定する仕組みであり、cond のあとに各条件とコマンドをかっこでくくったものを並べて記述する。また、コマンドとして出力コード列そのものを指定することもできる。

「コマンドマクロ」は、一種のサブルーチンである。複数回用いられる一まとまりのコマンド群に名前をつけておき、一つのコマンドとして呼び出すことができる。コマンドマクロは

```
(マクロの名前　コマンド*)
```

という形式で記述して定義する。

たとえば次の例でのコマンドは条件節一つであり、条件節内の最初の行は、「`0x0901` というコードが入力コード列として現われたら、`0xC4` というコードを出力コード列として生成すること」を表している（最後の行の `bc-tc` の意味に関しては後述）。

```
(post-modifier
 (cond
  ((0x0901)  0xC4)
  ((0x0902)  0xC6)
  ((0x0951)  0x27)
  ((0x0952)  bc-tc 0x2D)))
```

この例のコマンドマクロは `post-modifier` という名前なので、生成ルールのコマンド部分に `post-modifier` と指定することによって呼び出すことができる。

「特別なコマンド」としては、以下の６種類をサポートしている。

`=` 入力コード列の最初の一つのコードを消費し、それをそのままターゲットシークエンスに書き出す。

`*` 直前の処理が一つ以上のコードを消費していたら、それを繰り返す。

`<` 書記素 (grapheme) クラスタの開始を指定。

`>` 書記素 (grapheme) クラスタの終了を指定。

`|` カテゴリとして空白を持つ特別なグリフを挿入する。このグリフは、変換を繰り返して一連の生成を行なう際に、シークエンスのデリミタとして用いられることを想定している。

グリフ合成規則 直後に生成されるコードのグリフを直前のグリフと２次元的にどのように合成するかを指定する。例えば、 `bc-tc 0x2D` というコマンド列は、「bc-tc という合成規則を用いて、0x2D というコードを持つグリフを直前のグリフに合成する」ことを意味している。

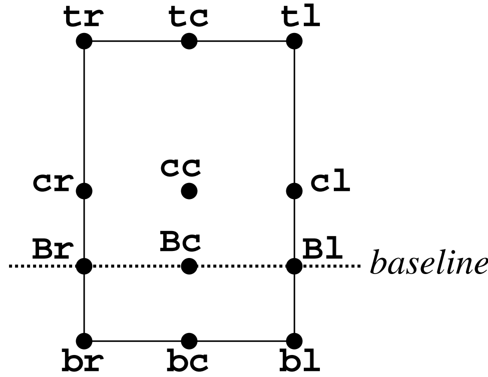
グリフ合成規則は

```
R1.R2
```

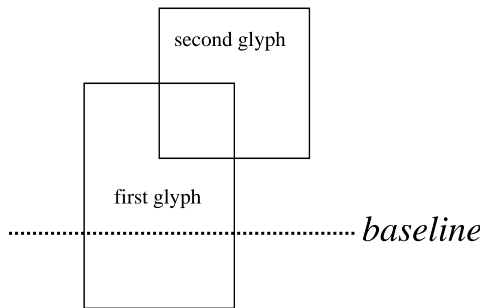
もしくは

```
R1 ( V | H | VH ) DIFF  R2
V :: = '+' | '-'
H :: = '<' | '>'
```

という形式を取る。ここで`R1` および `R2` はそれぞれのグリフの参照点であり、これらの点がが重なるように合成する。可能な参照点を以下にしめす。



たとえば tl.cc という合成規則は２つのグリフを以下のように合成する。



２番目の形式の合成規則は２番目のグリフの参照点を１番目のグリフの参照点から上下左右にずらして合成する場合に使う。+と-はそれぞれ上下に、<と>はそれぞれ左右にずらすことを示す。DIFF は 0 から 127 までの値をとり、フォントのサイズを 100 とした場合のずれの大きさを示す。

4.5 実例

pattern の記述方法として正規表現やマッチインデックスを許していること、またcommand として充分な制御構造を持たせていることによって、m17n ライブラリのフォントコンポーザでは、柔軟で記述能力の高い複雑なルールも比較的容易に書くことができる。この枠組みを用いて、種々のスクリプト用の表示ドライバを試作し、検証を行なっている。図 2 にそれらの表示の例を示す。

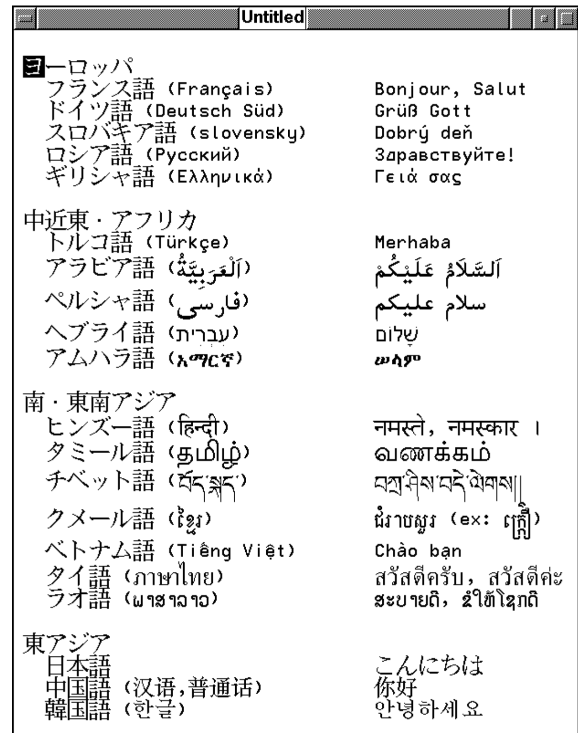


図 2: 種々のスクリプトの表示例

5 今後の課題

5.1 OpenType フォント

OpenType フォントの登場により、複雑な処理を必要とするスクリプトの表示は改善されつつある。OpenType フォントは少なくとも Unicode エンコーディングをサポートしている。また組版用の情報として、ベースライン、グリフの定義、グリフ位置、グリフ置換、行詰めのテーブルを持つことができるため、代替グリフ、リガチャなどをどう処理すれば良いかの情報をフォント自身から得ることができるからである。しかし、スクリプト毎に、さらには言語毎に OpenType フォントをどうドライブするかという処理は、相変わらず表示エンジンに任されている。

たとえば Windows 上の表示エンジン Uniscribe がインド系の各スクリプトを表示する際は、それぞれのスクリプト用のモジュールによって、シラブルの切り出し、文字の並べ替えを行なった後に、OTLS (OpenType Library Service) を呼出して、グリフの選択、リガチャ、合成のためのグリフの２次元配置などの OpenType フォントが提供する *feature* の実行を行う。

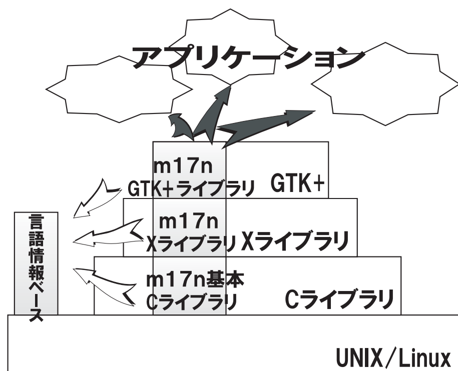


図 3: m17n ライブラリの構成

m17n ライブラリでは、FreeType ライブライが提供する OTLS 相当の機能を利用して、OpenType を利用する予定である。しかしその際も、フォントコンポーザで、コード列のどの部分に対してどの *feature* を実行するかを記述できるようにすることによって、レイアウト・エンジンの汎用性を保つことができる。

5.2 GNU Emacs 等との関連

我々はこれまで Mule, GNU Emacs ver.20 以降などでエディタの多言語化を行なってきている。m17n ライブラリはこれらの開発の経験に基づいてはいるものの、独立のシステムであり、本稿で述べる表示機能はこれら過去に開発したソフトウェアの表示機能とは無関係である。

なお現在、Unicode をベースとした Emacs を開発中であり、この Emacs では m17n ライブラリと同様の表示ルーティンを採用する。m17n ライブラリ自体を Emacs から直接利用することはないが、データベースである言語情報ベースは共有できるようにする予定である。

5.3 GNOME

現在、m17n ライブラリは Xlib レベルの API しか提供していないが、実際アプリケーションプログラマに使ってもらうには、ツールキットレベルの API が必要であろう。現在のところ、GNOME の GTK+ および Pango に m17n ライブラリを組込むことを計画している。これによって GNOME 上のアプリケーションプログラムは、図 3 のような階層構造になる。

6 結論

Unix/Linux 汎用多言語処理機能ライブラリ m17n-lib の表示機能について述べた。このライブラリの表示機能は、ライブラリ利用者であるアプリケーション開発者が、必要な言語／スクリプト／フォントを必要に応じて、容易に追加できることを目標としている。このため、汎用のレイアウトエンジンとフォントコンポーザと呼ばれるフォント毎のドライバに分離した設計を採用した。

フォントコンポーザの記述能力を確認するため、デバナガリ、チベット、クメールなどの、表示の際に複雑な処理を要するスクリプトについてドライバを試作し、その機能を確認している。

今後、m17n ライブラリの表示機能に関しては、OpenType フォントを利用する枠組の開発や、フォントコンポーザの XML 化などを行っていく予定である。