

Lua \TeX -ja パッケージ

Lua \TeX -ja プロジェクトチーム

2014 年 3 月 22 日

目次

第 I 部	ユーザーズマニュアル	4
1	はじめに	4
1.1	背景	4
1.2	pTeX からの主な変更点	4
1.3	用語と記法	5
1.4	プロジェクトについて	5
2	使い方	6
2.1	インストール	6
2.2	注意点	7
2.3	plain TeX で使う	7
2.4	LaTeX で使う	8
3	フォントの変更	8
3.1	plain TeX and LaTeX 2 _ε	8
3.2	fontspec	10
3.3	プリセット設定	10
3.4	\CID, \UTF と <code>otf</code> パッケージのマクロ	13
4	パラメータの変更	14
4.1	JAchar の範囲の設定	14
4.2	kanjiskip と xkanjiskip	15
4.3	xkanjiskip の挿入設定	16
4.4	ベースラインの移動	17
第 II 部	リファレンス	17
5	LuaTeX-ja における \catcode	18
5.1	予備知識：pTeX と upTeX における \kcatcode	18
5.2	LuaTeX-ja の場合	18
5.3	制御綴中に使用出来る JIS 非漢字の違い	19
6	フォントメトリックと和文フォント	20
6.1	\jfont 命令	20
6.2	psft プリフィックス	22
6.3	JFM ファイルの構造	23
6.4	数式フォントファミリ	27
6.5	コールバック	27
7	パラメータ	29
7.1	\ltjsetparameter	29
7.2	\ltjgetparameter	31

8	plain でも IAT_EX でも利用可能なその他の命令	32
8.1	p _T E _X 互換用命令	32
8.2	\inhibitglue	32
8.3	\ltjdeclarealtfont	32
9	IAT_EX 2_ε 用の命令	33
9.1	NFSS2 へのパッチ	33
10	拡張	36
10.1	luatexja-fontspec.sty	36
10.2	luatexja-otf.sty	36
10.3	luatexja-adjust.sty	37
第 III 部 実装		38
11	パラメータの保持	38
11.1	Lua _T E _X -ja で用いられるレジスタと whatsit ノード	38
11.2	Lua _T E _X -ja のスタックシステム	40
11.3	スタックシステムで使用される関数	41
11.4	パラメータの拡張	42
12	和文文字直後の改行	42
12.1	参考：p _T E _X の動作	42
12.2	Lua _T E _X -ja の動作	43
13	JFM グルーの挿入, kanjiskip と xkanjiskip	44
13.1	概要	44
13.2	「クラスタ」の定義	45
13.3	段落 / hbox の先頭や末尾	47
13.4	概観と典型例：2つの「和文 A」の場合	47
13.5	その他の場合	49
14	<u>listings</u> パッケージへの対応	53
14.1	注意	53
14.2	文字種	54
15	和文の行長補正方法	55
15.1	行末文字の位置調整	56
15.2	グルーの調整	56
16	IVS 対応 (未完)	57
17	複数フォントの「合成」 (未完)	58
18	Lua_TE_X-ja におけるキャッシュ	58
18.1	キャッシュの使用箇所	58
18.2	内部命令	59

参考文献	60
付録 A Package versions used in this document	61

本ドキュメントはまだまだ未完成です.

第 I 部

ユーザーズマニュアル

1 はじめに

LuaTeX-ja パッケージは、次世代標準 TeX である LuaTeX の上で、pTeX と同等／それ以上の品質の日本語組版を実現させようとするマクロパッケージである。

1.1 背景

従来、「TeX を用いて日本語組版を行う」といったとき、エンジンとしては ASCII pTeX やその拡張物が用いられることが一般的であった。pTeX は TeX のエンジン拡張であり、(少々仕様上不便な点はあるものの) 商業印刷の分野にも用いられるほどの高品質な日本語組版を可能としている。だが、それは弱点にもなってしまった:pTeX という(組版的に)満足なものがあつたため、海外で行われている数々の TeX の拡張——例えば ϵ -TeX や pdfTeX——や、TrueType, OpenType, Unicode といった計算機で日本語を扱う際の状況の変化に追従することを怠ってしまったのだ。

ここ数年、若干状況は改善されてきた。現在手に入る大半の pTeX バイナリでは外部 UTF-8 入力を利用可能となり、さらに Unicode 化を推進し、pTeX の内部処理まで Unicode 化した upTeX も開発されている。また、pTeX に ϵ -TeX 拡張をマージした ϵ -pTeX も登場し、TeX Live 2011 では pL^ATeX が ϵ -pTeX の上で動作するようになった。だが、pdfTeX 拡張 (PDF 直接出力や micro-typesetting) を pTeX に対応させようという動きはなく、海外との gap は未だにあるのが現状である。

しかし、LuaTeX の登場で、状況は大きく変わることになった。Lua コードで “callback” を書くことにより、LuaTeX の内部処理に割り込みをかけることが可能となった。これは、エンジン拡張という真似をしなくても、Lua コードとそれに関する TeX マクロを書けば、エンジン拡張とほぼ同程度のことができるようになったということを意味する。LuaTeX-ja は、このアプローチによって Lua コード・TeX マクロによって日本語組版を LuaTeX の上で実現させようという目的で開発が始まったパッケージである。

1.2 pTeX からの主な変更点

LuaTeX-ja は、pTeX に多大な影響を受けている。初期の開発目標は、pTeX の機能を Lua コードにより実装することであった。しかし、開発が進むにつれ、pTeX の完全な移植は不可能であり、また pTeX における実装がいささか不可解になっているような状況も発見された。そのため、**LuaTeX-ja は、もはや pTeX の完全な移植は目標とはしない。pTeX における不自然な仕様・挙動があれば、そこは積極的に改める。**

以下は pTeX からの主な変更点である。

- 和文フォントは (小塚明朝, IPA 明朝などの) 実際のフォント, 和文フォントメトリック (JFM と呼ぶ^{*1}) の組である。
- 日本語の文書中では改行はほとんどどこでも許されるので、pTeX では和文文字直後の改行は無視される (スペースが入らない) ようになっていた。しかし、LuaTeX-ja では LuaTeX の仕様のためにこの機能は完全には実装されていない。
- 2つの和文文字の間や、和文文字と欧文文字の間に入るグルー／カーン(両者をあわせて **JAg**lue と呼ぶ) の挿入処理が 0 から書き直されている。

^{*1} 混乱を防ぐため、pTeX の意味での JFM (min10.tfm) などは本ドキュメントでは**和文用 TFM** とよぶことにする。

- LuaTeX の内部での合字の扱いは「ノード」を単位として行われるようになっている（例えば、`of{f}ice` で合字は抑制されない）。それに合わせ、**JAgglue** の挿入処理もノード単位で実行される。
- さらに、2つの文字の間にある行末では効果を持たないノード（例えば `\special` ノード）や、イタリック補正に伴い挿入されるカーンは挿入処理中では無視される。
- **注意：上の 2 つの変更により、従来 JAgglue の挿入処理を分断するのに使われていたいくつかの方法は用いることができない。具体的には、次の方法はもはや無効である：**
`ちょ{つと}` `ちょ\つと`
もし同じことをやりたければ、空の水平ボックス (hbox) を間に挟めばよい：
`ちょ\hbox{つと}`
- 処理中では、2つの和文フォントは、実フォントが異なるだけの場合には同一視される。
- LuaTeX-ja では、pTeX と同様に漢字・仮名を制御綴内に用いることができ、`\西暦` などが正しく動作するようにしている。但し、制御綴中に使える和文文字が pTeX・upTeX と全く同じではないことに注意すること。
- 現時点では、縦書きは LuaTeX-ja ではサポートされていない。

詳細については第 III 部を参照。

1.3 用語と記法

本ドキュメントでは、以下の用語と記法を用いる：

- 文字は次の 2 種類に分けられる。この類別はユーザが後から変更可能である（4.1 節を参照）。
 - **JAchar**: ひらがな、カタカナ、漢字、和文用の約物といった日本語組版に使われる文字のことを指す。
 - **ALchar**: ラテンアルファベットを始めとする、その他全ての文字を指す。
そして、**ALchar** の出力に用いられるフォントを**欧文フォント**と呼び、**JAchar** の出力に用いられるフォントを**和文フォント**と呼ぶ。
- サンセリフ体で書かれた語（例：`prebreakpenalty`）は日本語組版用のパラメータを表し、これらは `\ltjsetparameter` コマンドのキーとして用いられる。
- 下線付きタイプライタ体の語（例：`fontspec`）は LaTeX のパッケージやクラスを表す。
- 本ドキュメントでは、自然数は 0 から始まる。自然数全体の集合は ω と表記する。

1.4 プロジェクトについて

■**プロジェクト Wiki** プロジェクト Wiki は構築中である。

- <http://sourceforge.jp/projects/luatex-ja/wiki/FrontPage> (日本語)
- <http://sourceforge.jp/projects/luatex-ja/wiki/FrontPage%28en%29> (英語)
- <http://sourceforge.jp/projects/luatex-ja/wiki/FrontPage%28zh%29> (中国語)

本プロジェクトは SourceForge.JP のサービスを用いて運営されている。

■**開発メンバー**

- | | | |
|---------|----------|---------|
| • 北川 弘典 | • 前田 一貴 | • 八登 崇之 |
| • 黒木 裕介 | • 阿部 紀行 | • 山本 宗宏 |
| • 本田 知亮 | • 齋藤 修三郎 | • 馬 起園 |

2 使い方

2.1 インストール

LuaTeX-ja パッケージのインストールには、次のものが必要である。

- LuaTeX beta-0.74.0 (or later)
- [luaotfload](#) v2.2 (or later)
- [luatexbase](#) v0.6
- [xunicode](#) v0.981 (2011/09/09)
- [adobemapping](#) (Adobe cmap and pdfmapping files)

本バージョンの LuaTeX-ja は TeX Live 2012 以前では動作しない。これは、LuaTeX と [luaotfload](#) が TeX Live 2013 において更新されたことによる。

現在、LuaTeX-ja は以下のアーカイブ、およびディストリビューションに収録されている：

- CTAN ([macros/luatex/generic/luatexja](#))
- MiKTeX ([luatexja.tar.lzma](#))
- TeX Live ([texmf-dist/tex/luatex/luatexja](#))
- W32TeX ([luatexja.tar.xz](#))

例えば TeX Live 2013 を利用しているなら、LuaTeX-ja は TeX Live manager (tlmgr) を使ってインストールすることができる。

```
$ tlmgr install luatexja
```

■手動インストール方法

1. ソースアーカイブを以下のいずれかの方法で取得する。現在公開されているのはあくまでも開発版であって、安定版でないことに注意。

- Git リポジトリの内容をコピーする：

```
$ git clone git://git.sourceforge.jp/gitroot/luatex-ja/luatexja.git
```

- master ブランチのスナップショット (tar.gz 形式) をダウンロードする。

```
http://git.sourceforge.jp/view?p=luatex-ja/luatexja.git;a=snapshot;h=HEAD;sf=tgz.
```

master ブランチ (従って、CTAN 内のアーカイブも) はたまにしか更新されないことに注意。主な開発は master の外で行われ、比較的まとまってきたらそれを master に反映させることにしている。

2. 「Git リポジトリをコピー」以外の方法でアーカイブを取得したならば、それを展開する。src/ をはじめとしたいくつかのディレクトリができるが、動作には src/以下の内容だけで十分。
3. もし CTAN から本パッケージを取得したのであれば、日本語用クラスファイルや標準の禁則処理用パラメータを格納した `ltj-kinsoku.lua` を生成するために、以下を実行する必要がある：

```
$ cd src
$ lualatex ltjclasses.ins
$ lualatex ltjsclasses.ins
$ lualatex ltjltxdoc.ins
```

```
$ luatex ltj-kinsoku_make.tex
```

ここで使用した `*.{dtx,ins}` と `ltj-kinsoku_make.tex` は通常の使用にあたっては必要ない。

4. `src` の中身を自分の `TEXMF` ツリーにコピーする。場所の例としては、例えば `TEXMF/tex/luatex/luatexja/` がある。シンボリックリンクが利用できる環境で、かつリポジトリを直接取得したのであれば、(更新を容易にするために) コピーではなくリンクを貼ることを勧める。
5. 必要があれば、`mktexlsr` を実行する。

2.2 注意点

- 原稿のソースファイルの文字コードは UTF-8 固定である。従来日本語の文字コードとして用いられてきた EUC-JP や Shift-JIS は使用できない。
- LuaTeX-ja は動作が pTeX に比べて非常に遅い。コードを調整して徐々に速くしているが、まだ満足できる速度ではない。LuaJITTeX を用いると LuaTeX のだいたい 1.3 倍の速度で動くようであるが、IPA mj 明朝などの大きいフォントを用いた場合には LuaTeX よりも遅くなることもある。
- LuaTeX-ja が動作するためには、**導入・更新後の初回起動時に UniJIS2004-UTF32-H, Adobe-Japan1-UCS2 という 2 つの CMap が LuaTeX によって見つけられることが必要である。**しかし MiKTeX ではそのようなになっていないので、次のエラーが発生するだろう：

```
! LuaTeX error ...iles (x86)/MiKTeX 2.9/tex/luatex/luatexja/ltj-rmlgbm.lua
bad argument #1 to 'open' (string expected, got nil)
```

そのような場合には、[プロジェクト Wiki 英語版トップページ](#)中に書かれているバッチファイルを実行して欲しい。このバッチファイルは、作業用ディレクトリに CMap 達をコピーし、その中で LuaTeX-ja の初回起動を行い、作業用ディレクトリを消す作業をしている。

2.3 plain TeX で使う

LuaTeX-ja を plain TeX で使うためには、単に次の行をソースファイルの冒頭に追加すればよい：

```
\input luatexja.sty
```

これで (`ptex.tex` のように) 日本語組版のための最低限の設定がなされる：

- 以下の 6 つの和文フォントが定義される：

字体	フォント名	'10 pt'	'7 pt'	'5 pt'
明朝体	Ryumin-Light	<code>\tenmin</code>	<code>\sevenmin</code>	<code>\fivemin</code>
ゴシック体	GothicBBB-Medium	<code>\tengt</code>	<code>\seventgt</code>	<code>\fivegt</code>

- “Ryumin-Light” と “GothicBBB-Medium” は PDF ファイルに埋め込まずに名前参照のみで用いることが広く受け入れられており、この場合 PDF リーダーが適切な外部フォントで代用する (例えば、Adobe Reader では Ryumin-Light は小塚明朝で代替される)。そこで、これらを引き続きデフォルトのフォントとして採用する。
- 欧文フォントの文字は和文フォントの文字よりも、同じ文字サイズでも一般に小さくデザインされている。そこで、標準ではこれらの和文フォントの実際のサイズは指定された値よりも小さくなるように設定されており、具体的には指定の 0.962216 倍にスケールされる。この 0.962216 という数値も、pTeX におけるスケーリングを踏襲した値である。

- **J**Achar と **A**Lchar の間に入るグルー (`xkanjiskip`) の量は次のように設定されている：

$$(0.25 \cdot 0.962216 \cdot 10 \text{ pt})_{-1 \text{ pt}}^{+1 \text{ pt}} = 2.40554 \text{ pt}_{-1 \text{ pt}}^{+1 \text{ pt}}$$

2.4 L^AT_EX で使う

■**L^AT_EX 2_ε** L^AT_EX 2_ε を用いる場合も基本的には同じである。日本語組版のための最低限の環境を設定するためには、`luatexja.sty` を読み込むだけでよい：

```
\usepackage{luatexja}
```

これで pL^AT_EX の `plfonts.dtx` と `pldefs.ltx` に相当する最低限の設定がなされる：

- JY3 は和文フォント用のフォントエンコーディングである（横書き用）。将来的に、`LuaTEX-ja` で縦書きがサポートされる際には、JT3 を縦書き用として用いる予定である。
- pL^AT_EX と同様に、標準では「明朝体」「ゴシック体」の 2 種類を用いる：

字体	ファミリ名		
明朝体	<code>\textmc{...}</code>	<code>{\mcfamily ...}</code>	<code>\mcdefault</code>
ゴシック体	<code>\textgt{...}</code>	<code>{\gtfamily ...}</code>	<code>\gtdefault</code>

- 標準では、次のフォントファミリが用いられる：

字体	ファミリ	<code>\mdseries</code>	<code>\bfseries</code>	スケール
明朝体	<code>mc</code>	Ryumin-Light	GothicBBB-Medium	0.962216
ゴシック体	<code>gt</code>	GothicBBB-Medium	GothicBBB-Medium	0.962216

どちらのファミリにおいても、その bold シリーズで使われるフォントはゴシック体の medium シリーズで使われるフォントと同じあることに注意。これは初期の DTP において和文フォントが 2 つ（それがちょうど Ryumin-Light, GothicBBB-Medium だった）しか利用できなかった時の名残であり、pL^AT_EX での標準設定とも同じである。

- 数式モード中の和文文字は明朝体 (`mc`) で出力される。

しかしながら、上記の設定は日本語の文書にとって十分とは言えない。日本語文書を組版するためには、`article.cls`, `book.cls` といった欧文用のクラスファイルではなく、和文用のクラスファイルを用いた方がよい。現時点では、`jclasses` (pL^AT_EX の標準クラス) と `jsclasses` (奥村晴彦氏によるクラスファイル) に対応するものとして、`ltjclasses`, `ltjsclasses` がそれぞれ用意されている。

3 フォントの変更

3.1 plain T_EX and L^AT_EX 2_ε

■**plain T_EX** plain T_EX で和文フォントを変更するためには、pT_EX のように `\jfont` 命令を直接用いる。6.1 節を参照。

■**L^AT_EX 2_ε (NFSS2)** L^AT_EX 2_ε については、`LuaTEX-ja` ではフォント選択システムを pL^AT_EX 2_ε (`plfonts.dtx`) の大部分をそのまま採用している。

- `\fontfamily`, `\fontseries`, `\fontshape`, そして `\selectfont` が和文フォントの属性を変更

するために使用できる.

	エンコーディング	ファミリ	シリーズ	シェーブ	選択
欧文	<code>\romanencoding</code>	<code>\romanfamily</code>	<code>\romanseries</code>	<code>\romanshape</code>	<code>\useroman</code>
和文	<code>\kanjiencoding</code>	<code>\kanjifamily</code>	<code>\kanjiseries</code>	<code>\kanjishape</code>	<code>\usekanji</code>
両方	—	—	<code>\fontseries</code>	<code>\fontshape</code>	—
自動選択	<code>\fontencoding</code>	<code>\fontfamily</code>	—	—	<code>\usefont</code>

ここで、`\fontencoding{encoding}` は、引数により和文側か欧文側かのどちらかのエンコーディングを変更する。例えば、`\fontencoding{JY3}` は和文フォントのエンコーディングを JY3 に変更し、`\fontencoding{T1}` は欧文フォント側を T1 へと変更する。`\fontfamily` も引数により和文側、欧文側、あるいは両方のフォントファミリを変更する。詳細は 9.1 節を参照すること。

- 和文フォントファミリの定義には `\DeclareFontFamily` の代わりに `\DeclareKanjiFamily` を用いる。しかし、現在の実装では `\DeclareFontFamily` を用いても問題は生じない。
- 和文フォントのシェイプを定義するには、通常の `\DeclareFontShape` を使えば良い：

```
\DeclareFontShape{JY3}{mc}{bx}{n}{<-> s*KozMinPr6N-Bold:jfm=ujis;-kern}{
  % Kozuka Mincho Pr6N Bold
```

仮名書体を使う場合など、複数の和文フォントを組み合わせたい場合は 8.3 節の `\ltxdeclarealtfont` と、その L^AT_EX 版の `\DeclareAlternateKanjiFont` (9.1 節) を参照せよ。

■注意：数式モード中の和文文字 p_TE_X では、特に何もしないでも数式中に和文文字を記述することができた。そのため、以下のようなソースが見られた：

```
1 $f_{高温}$~{($f_{\text{high temperature}})}$       $f_{\text{高温}} (f_{\text{high temperature}}).$ 
   ).
2 \[ y=(x-1)^2+2\quad \text{よって}\quad y>0 \]       $y = (x - 1)^2 + 2 \quad \text{よって} \quad y > 0$ 
3 $5\in \text{素}:=\{\,p\in\mathbb{N}:\text{素 } p \text{ is a prime}\,\}\$       $5 \in \text{素} := \{p \in \mathbb{N} : p \text{ is a prime}\}.$ 
```

Lua_TE_X-ja プロジェクトでは、数式モード中での和文文字はそれらが識別子として用いられるときのみ許されると考えている。この観点から、

- 上記数式のうち 1, 2 行目は正しくない。なぜならば「高温」が意味のあるラベルとして、「よって」が接続詞として用いられているからである。
- しかしながら、3 行目は「素」が単なる識別子として用いられているので正しい。

したがって、Lua_TE_X-ja プロジェクトの意見としては、上記の入力は次のように直されるべきである：

```
1 $f_{\text{高温}}$~%
2 ($f_{\text{high temperature}})$$.       $f_{\text{高温}} (f_{\text{high temperature}}).$ 
3 \[ y=(x-1)^2+2\quad
4 \quad \text{よって}\quad y>0 \]       $y = (x - 1)^2 + 2 \quad \text{よって} \quad y > 0$ 
5 $5\in \text{素}:=\{\,p\in\mathbb{N}:\text{素 } p \text{ is a prime}\,\}\$       $5 \in \text{素} := \{p \in \mathbb{N} : p \text{ is a prime}\}.$ 
```

また Lua_TE_X-ja プロジェクトでは、和文文字が識別子として用いられることはほとんどないと考えており、したがってこの節では数式モード中の和文フォントを変更する方法については記述しない。

この方法については 6.4 節を参照のこと。

3.2 fontspec

`fontspec` パッケージと同様の機能を和文フォントに対しても用いるためには、`luatexja-fontspec` パッケージをプリアンブルで読み込む必要がある。このパッケージは必要ならば自動で `luatexja` パッケージと `fontspec` パッケージを読み込む。

`luatexja-fontspec` パッケージでは、以下の 7 つのコマンドを `fontspec` パッケージの元のコマンドに対応するものとして定義している：

和文	<code>\jfontspec</code>	<code>\setmainjfont</code>	<code>\setsansjfont</code>
欧文	<code>\fontspec</code>	<code>\setmainfont</code>	<code>\setsansfont</code>
和文	<code>\newfontfamily</code>	<code>\newfontface</code>	<code>\defaultjfontfeatures</code>
欧文	<code>\newfontfamily</code>	<code>\newfontface</code>	<code>\defaultfontfeatures</code>
和文	<code>\addfontfeatures</code>		
欧文	<code>\addfontfeatures</code>		

- 1 `\fontspec[Numbers=OldStyle]{LMSans10-Regular}`
- 2 `\jfontspec[CJKShape=NLC]{KozMinPr6N-Regular}`
- 3 `JIS-X-0213:2004→辻` JIS X 0213:2004 →辻
- 4 JIS X 0208:1990 →辻
- 5 `\jfontspec[CJKShape=JIS1990]{KozMinPr6N-Regular}`
- 6 `JIS-X-0208:1990→辻`

和文フォントについては全ての和文文字のグリフがほぼ等幅であるのが普通であるため、`\setmonojfont` コマンドは存在しないことに注意。また、これらの和文用の 7 つのコマンドではフォント内のペアカーニング情報は標準では使用されない、言い換えれば kern feature は標準では無効化となっている。これは以前のバージョンの Lua_TE_X-ja との互換性のためである (6.1 節を参照)。

3.3 プリセット設定

よく使われている和文フォント設定を一行で指定できるようにしたのが `luatexja-preset` パッケージである。このパッケージは、`otf` パッケージの一部機能と八登崇之氏による `PXchfon` パッケージの一部機能とを合わせたような格好をしており、内部で `luatexja-fontspec`、従って `fontspec` を読み込んでいる。

もし `fontspec` パッケージに何らかのオプションを渡す必要がある^{*2}場合は、次のように `luatexja-preset` の前に `fontspec` を手動で読みこめば良い：

```
\usepackage[no-math]{fontspec}
\usepackage[...]{luatexja-preset}
```

■一般的なオプション

`nodeluxe` L_AT_EX 2_ε 環境下での標準設定のように、明朝体・ゴシック体を各 1 ウェイトで使用する。より具体的に言うと、この設定の下では `\mcfamily\bfseries`、`\gtfamily\bfseries`、`\gtfamily\mdseries` はみな同じフォントとなる。このオプションは標準で有効になっている。

^{*2} 例えば、数式フォントまで置換されてしまい、`\mathit` によってギリシャ文字の斜体大文字が出なくなる、など。

`deluxe` 明朝体 2 ウェイト・ゴシック体 3 ウェイトと、丸ゴシック体 (`\mgfamily`, `\textmg{...}`) を使用可能とする。ゴシック体は細字・太字・極太の 3 ウェイトがあるが、極太ゴシック体はファミリの切り替え (`\gtebfamily`, `\textgteb{...}`) で実現している。`fontspec` では通常 (`\mdseries`) と太字 (`\bfseries`) しか扱えないためにこのような中途半端な実装になっている。

`expert` 横組専用仮名を用いる。また、`\rubyfamily` でルビ用仮名が使用可能となる。

`bold` 「明朝の太字」をゴシック体の太字によって代替する。

`90jis` 出来る限り 90JIS の字形を使う。

`jis2004` 出来る限り JIS2004 の字形を使う。

`jis` 用いる JFM を (JIS フォントメトリック類似の) `jfm-jis.lua` にする。このオプションがない時は Lua_{TEX}-ja 標準の `jfm-ujis.lua` が用いられる。

`90jis` と `jis2004` については本パッケージで定義された明朝体・ゴシック体 (・丸ゴシック体) のみ有効である。両オプションが同時に指定された場合の動作については全く考慮していない。

■多ウェイト用プリセットの一覧 `morisawa-pro`, `morisawa-pr6n` 以外はフォントの指定は (ファイル名でなく) フォント名で行われる。

`kozuka-pro` Kozuka Pro (Adobe-Japan1-4) fonts.

`kozuka-pr6` Kozuka Pr6 (Adobe-Japan1-6) fonts.

`kozuka-pr6n` Kozuka Pr6N (Adobe-Japan1-6, JIS04-savvy) fonts.

小塚 Pro 書体・Pr6N 書体は Adobe InDesign 等の Adobe 製品にバンドルされている。「小塚丸ゴシック」は存在しないので、便宜的に小塚ゴシック H によって代用している。

family	series	kozuka-pro	kozuka-pr6	kozuka-pr6n
明朝	medium	KozMinPro-Regular	KozMinProVI-Regular	KozMinPr6N-Regular
	bold	KozMinPro-Bold	KozMinProVI-Bold	KozMinPr6N-Bold
ゴシック	medium	KozGoPro-Regular*	KozGoProVI-Regular*	KozGoPr6N-Regular*
		KozGoPro-Medium	KozGoProVI-Medium	KozGoPr6N-Medium
	bold	KozGoPro-Bold	KozGoProVI-Bold	KozGoPr6N-Bold
	heavy	KozGoPro-Heavy	KozGoProVI-Heavy	KozGoPr6N-Heavy
丸ゴシック		KozGoPro-Heavy	KozGoProVI-Heavy	KozGoPr6N-Heavy

上の表において、*つきのフォント (KozGo...-Regular) は、`deluxe` オプション非指定時にゴシック体細字として用いられる。

`hiragino-pro` Hiragino Pro (Adobe-Japan1-5) fonts.

`hiragino-pron` Hiragino ProN (Adobe-Japan1-5, JIS04-savvy) fonts.

ヒラギノフォントは、Mac OS X 以外にも、一太郎 2012 の上位エディションにもバンドルされている。極太ゴシックとして用いるヒラギノ角ゴ W8 は、Adobe-Japan1-3 の範囲しかカバーしていない Std/StdN フォントであり、その他は Adobe-Japan1-5 対応である。

family	series	hiragino-pro	hiragino-pron
明朝	medium	Hiragino Mincho Pro W3	Hiragino Mincho ProN W3
	bold	Hiragino Mincho Pro W6	Hiragino Mincho ProN W6
ゴシック	medium	Hiragino Kaku Gothic Pro W3*	Hiragino Kaku Gothic ProN W3*
		Hiragino Kaku Gothic Pro W6	Hiragino Kaku Gothic ProN W6
	bold	Hiragino Kaku Gothic Pro W6	Hiragino Kaku Gothic ProN W6
	heavy	Hiragino Kaku Gothic Std W8	Hiragino Kaku Gothic StdN W8
丸ゴシック		Hiragino Maru Gothic ProN W4	Hiragino Maru Gothic ProN W4

morisawa-pro Morisawa Pro (Adobe-Japan1-4) fonts.

morisawa-pr6n Morisawa Pr6N (Adobe-Japan1-6, JIS04-savvy) fonts.

family	series	morisawa-pro	morisawa-pr6n
明朝	medium	A-OTF-RyuminPro-Light.otf	A-OTF-RyuminPr6N-Light.otf
	bold	A-OTF-FutoMinA101Pro-Bold.otf	A-OTF-FutoMinA101Pr6N-Bold.otf
ゴシック	medium	A-OTF-GothicBBBPro-Medium.otf	A-OTF-GothicBBBPr6N-Medium.otf
	bold	A-OTF-FutoGoB101Pro-Bold.otf	A-OTF-FutoGoB101Pr6N-Bold.otf
	heavy	A-OTF-MidashiGoPro-MB31.otf	A-OTF-MidashiGoPr6N-MB31.otf
丸ゴシック		A-OTF-Jun101Pro-Light.otf	A-OTF-ShinMGoPr6N-Light.otf

yu-win Yu fonts bundled with Windows 8.1.

yu-osx Yu fonts bundled with OSX Mavericks.

family	series	yu-win	yu-osx
明朝	medium	YuMincho-Regular	YuMincho Medium
	bold	YuMincho-Demibold	YuMincho Demibold
ゴシック	medium	YuGothic-Regular*	YuGothic Medium*
		YuGothic-Bold	YuGothic Bold
	bold	YuGothic-Bold	YuGothic Bold
	heavy	YuGothic-Bold	YuGothic Bold
丸ゴシック		YuGothic-Bold	YuGothic Bold

■単ウェイト用プリセット一覧 次に、単ウェイト用の設定を述べる。この4設定では「細字」「太字」の区別はない。また、丸ゴシック体はゴシック体と同じフォントを用いる。

	noembed	ipa	ipaex	ms
明朝体	Ryumin-Light (非埋込)	IPA 明朝	IPAex 明朝	MS 明朝
ゴシック体	GothicBBB-Medium (非埋込)	IPA ゴシック	IPAex ゴシック	MS ゴシック

■HG フォントの利用 すぐ前に書いた単ウェイト用設定を、Microsoft Office 等に付属する HG フォントを使って多ウェイト化した設定もある。

	ipa-hg	ipaex-hg	ms-hg
明朝体細字	IPA 明朝	IPAex 明朝	MS 明朝
明朝体太字	HG 明朝 E		
ゴシック体細字			
単ウェイト時	IPA ゴシック	IPAex ゴシック	MS ゴシック
jis2004 指定時	IPA ゴシック	IPAex ゴシック	MS ゴシック
それ以外の時	HG ゴシック M		
ゴシック体太字	HG ゴシック E		
ゴシック体極太	HG 創英角ゴシック UB		
丸ゴシック体	HG 丸ゴシック体 PRO		

なお、HG 明朝 E・HG ゴシック E・HG 創英角ゴシック UB・HG 丸ゴシック体 PRO の 4 つについては、内部で

標準 フォント名 (HGMinchoE など)

90jis 指定時 ファイル名 (hgrme.ttc, hgrge.ttc, hgrsgu.ttc, hgrsmp.ttf)

jis2004 指定時 ファイル名 (hgrme04.ttc, hgrge04.ttc, hgrsgu04.ttc, hgrsmp04.ttf)

として指定を行っているので注意すること。

3.4 \CID, \UTF と *otf* パッケージのマクロ

pLATEX では、JIS X 0208 にない Adobe-Japan1-6 の文字を出力するために、齋藤修三郎氏による *otf* パッケージが用いられていた。このパッケージは広く用いられているため、LuaTeX-ja においても *otf* パッケージの機能の一部をサポートしている。これらの機能を用いるためには [luatexja-otf](#) パッケージを読み込めばよい。

```

1 \jfontspec{KozMinPr6N-Regular.otf}
2 森\UTF{9DD7}外と内田百\UTF{9592}とが\UTF{9
   AD9}島屋に行く。
3
4 \CID{7652}飾区の\CID{13706}野家、
5 \CID{1481}城市、葛西駅、
6 高崎と\CID{8705}\UTF{FA11}
7
8 \aj半角{はんかくカタカナ}

```

```

森鷗外と内田百閒とが高島屋に行く。
葛飾区の吉野家、葛城市、葛西駅、高崎と高崎
はんかくカタ

```

otf パッケージでは、それぞれ次のようなオプションが存在した：

deluxe 明朝体・ゴシック体各 2 ウェイトと、丸ゴシック体を扱えるようになる。

expert 仮名が横組・縦組専用のものに切り替わり、ルビ用仮名も扱えるようになる。

bold ゴシック体を標準で太いウェイトのものに設定する。

しかしこれらのオプションは [luatexja-otf](#) パッケージには存在しない。*otf* パッケージが文書中で使用する和文用 TFM を自前の物に置き換えていたのに対し、[luatexja-otf](#) パッケージでは、そのようなことは行わないからである。

これら 3 オプションについては、`luatexja-preset` パッケージにプリセットを使う時に一緒に指定するか、あるいは対応する内容を 3.1 節、9.1 節 (NFSS2) や 3.2 節 (`fontspec`) の方法で手動で指定する必要がある。

4 パラメータの変更

LuaTeX-ja には多くのパラメータが存在する。そして LuaTeX の仕様のために、その多くは TeX のレジスタにではなく、LuaTeX-ja 独自の方法で保持されている。これらのパラメータを設定・取得するためには `\ltjsetparameter` と `\ltjgetparameter` を用いる。

4.1 JAchar の範囲の設定

LuaTeX-ja は、Unicode の U+0080–U+10FFFF の空間を 1 番から 217 番までの文字範囲に分割している。区分けは `\ltjdefcharrange` を用いることで (グローバルに) 変更することができ、例えば、次は追加漢字面 (SIP) にある全ての文字と「漢」を「100 番の文字範囲」に追加する。

```
\ltjdefcharrange{100}{"20000-"2FFFF,`漢}
```

各文字はただ一つの文字範囲に所属することができる。例えば、SIP は全て LuaTeX-ja のデフォルトでは 4 番の文字範囲に属しているが、上記の指定を行えば SIP は 100 番に属すようになり、4 番からは除かれる。

ALchar と **JAchar** の区別は文字範囲ごとに行われる。これは `jacharrange` パラメータによって編集できる。例えば、以下は LuaTeX-ja の初期設定であり、次の内容を設定している：

- 1 番, 4 番, 5 番の文字範囲に属する文字は **ALchar**。
- 2 番, 3 番, 6 番, 7 番, 8 番の文字範囲に属する文字は **JAchar**。

```
\ltjsetparameter{jacharrange={-1, +2, +3, -4, -5, +6, +7, +8}}
```

`jacharrange` パラメータの引数は非零の整数のリストである。リスト中の負の整数 $-n$ は「文字範囲 n に属する文字は **ALchar** として扱う」ことを意味し、正の整数 $+n$ は「**JAchar** として扱う」ことを意味する。

■初期設定 LuaTeX-ja では 8 つの文字範囲を予め定義しており、これらは以下のデータに基づいて決定している。

- Unicode 6.0 のブロック。
- Adobe-Japan1-6 の CID と Unicode の間の対応表 `Adobe-Japan1-UCS2`。
- 八登崇之氏による upTeX 用の `PXbase` バンドル。

以下ではこれら 8 つの文字範囲について記述する。添字のアルファベット「J」「A」は、その文字範囲内の文字が **JAchar** か **ALchar** かを表している。これらの設定は `PXbase` バンドルで定義されている `prefercjk` と類似のものである。なお、U+0080 以降でこれら 8 つの文字範囲に属さない文字は、217 番の文字範囲に属することになっている。

範囲 8^J ISO 8859-1 の上位領域 (ラテン 1 補助) と JIS X 0208 の共通部分。この文字範囲は以下の文字で構成される：

- § (U+00A7, Section Sign)
- ° (U+00B0, Degree sign)
- ¨ (U+00A8, Diaeresis)
- ± (U+00B1, Plus-minus sign)

表 1. 文字範囲 3 に指定されている Unicode ブロック.

U+2000–U+206F	General Punctuation	U+2070–U+209F	Superscripts and Subscripts
U+20A0–U+20CF	Currency Symbols	U+20D0–U+20FF	Comb. Diacritical Marks for Symbols
U+2100–U+214F	Letterlike Symbols	U+2150–U+218F	Number Forms
U+2190–U+21FF	Arrows	U+2200–U+22FF	Mathematical Operators
U+2300–U+23FF	Miscellaneous Technical	U+2400–U+243F	Control Pictures
U+2500–U+257F	Box Drawing	U+2580–U+259F	Block Elements
U+25A0–U+25FF	Geometric Shapes	U+2600–U+26FF	Miscellaneous Symbols
U+2700–U+27BF	Dingbats	U+2900–U+297F	Supplemental Arrows-B
U+2980–U+29FF	Misc. Mathematical Symbols-B	U+2B00–U+2BFF	Miscellaneous Symbols and Arrows

- ´ (U+00B4, Spacing acute)
- × (U+00D7, Multiplication sign)
- ¶ (U+00B6, Paragraph sign)
- ÷ (U+00F7, Division Sign)

範囲 1^A ラテン文字のうち, Adobe-Japan1-6 との共通部分があるもの. この範囲は以下の Unicode のブロックのうち**範囲 8**を除いた部分で構成されている:

- U+0080–U+00FF: Latin-1 Supplement
- U+0100–U+017F: Latin Extended-A
- U+0180–U+024F: Latin Extended-B
- U+0250–U+02AF: IPA Extensions
- U+02B0–U+02FF: Spacing Modifier Letters
- U+0300–U+036F: Combining Diacritical Marks
- U+1E00–U+1EFF: Latin Extended Additional

範囲 2^J ギリシャ文字とキリル文字. JIS X 0208 (したがってほとんどの和文フォント) はこれらの文字を持つ.

- U+0370–U+03FF: Greek and Coptic
- U+0400–U+04FF: Cyrillic
- U+1F00–U+1FFF: Greek Extended

範囲 3^J 句読点と記号類. ブロックのリストは表 1 に示してある.

範囲 4^A 通常和文フォントには含まれていない文字. この範囲は他の範囲にないほとんど全ての Unicode ブロックで構成されている. したがって, ブロックのリストを示す代わりに, 範囲の定義そのものを示す:

```
\ltjdefcharrange{4}{%
    "500-"10FF, "1200-"1DFF, "2440-"245F, "27C0-"28FF, "2A00-"2AFF,
    "2C00-"2E7F, "4DC0-"4DFF, "A4D0-"A82F, "A840-"ABFF, "FB00-"FE0F,
    "FE20-"FE2F, "FE70-"FEFF, "10000-"1FFFF, "E000-"F8FF} % non-Japanese
```

範囲 5^A 代用符号と補助私用領域.

範囲 6^J 日本語で用いられる文字. ブロックのリストは表 2 に示す.

範囲 7^J CJK 言語で用いられる文字のうち, Adobe-Japan1-6 に含まれていないもの. ブロックのリストは表 3 に示す.

4.2 kanjiskip と xkanjiskip

JAg glue は以下の 3 つのカテゴリに分類される:

- JFM で指定されたグルー/カーン. もし `\inhibitglue` が和文文字の周りで発行されていれば, このグルーは挿入されない.

表 2. 文字範囲 6 に指定されている Unicode ブロック.

U+2460–U+24FF	Enclosed Alphanumerics	U+2E80–U+2EFF	CJK Radicals Supplement
U+3000–U+303F	CJK Symbols and Punctuation	U+3040–U+309F	Hiragana
U+30A0–U+30FF	Katakana	U+3190–U+319F	Kanbun
U+31F0–U+31FF	Katakana Phonetic Extensions	U+3200–U+32FF	Enclosed CJK Letters and Months
U+3300–U+33FF	CJK Compatibility	U+3400–U+4DBF	CJK Unified Ideographs Extension A
U+4E00–U+9FFF	CJK Unified Ideographs	U+F900–U+FAFF	CJK Compatibility Ideographs
U+FE10–U+FE1F	Vertical Forms	U+FE30–U+FE4F	CJK Compatibility Forms
U+FE50–U+FE6F	Small Form Variants	U+20000–U+2FFFF	(Supplementary Ideographic Plane)
U+E0100–U+E01EF	Variation Selectors Supplement		

表 3. 文字範囲 7 に指定されている Unicode ブロック.

U+1100–U+11FF	Hangul Jamo	U+2F00–U+2FDF	Kangxi Radicals
U+2FF0–U+2FFF	Ideographic Description Characters	U+3100–U+312F	Bopomofo
U+3130–U+318F	Hangul Compatibility Jamo	U+31A0–U+31BF	Bopomofo Extended
U+31C0–U+31EF	CJK Strokes	U+A000–U+A48F	Yi Syllables
U+A490–U+A4CF	Yi Radicals	U+A830–U+A83F	Common Indic Number Forms
U+AC00–U+D7AF	Hangul Syllables	U+D7B0–U+D7FF	Hangul Jamo Extended-B

- デフォルトで 2 つの **J**Achar の間に挿入されるグルー ([kanjiskip](#)).
- デフォルトで **J**Achar と **A**Lchar の間に挿入されるグルー ([xkanjiskip](#)).

[kanjiskip](#) や [xkanjiskip](#) の値は以下のようにして変更可能である.

```
\ltjsetparameter{kanjiskip={0pt plus 0.4pt minus 0.4pt},
                    xkanjiskip={0.25\zw plus 1pt minus 1pt}}
```

ここで, `\zw` は現在の和文フォントの全角幅を表す長さであり, pTeX における長さ単位 `zw` と同じように使用できる.

JFM は「望ましい [kanjiskip](#) の値」や「望ましい [xkanjiskip](#) の値」を持っていることがある. これらのデータを使うためには, [kanjiskip](#) や [xkanjiskip](#) の値を `\maxdimen` の値に設定すればよい.

4.3 xkanjiskip の挿入設定

[xkanjiskip](#) がすべての **J**Achar と **A**Lchar の境界に挿入されるのは望ましいことではない. 例えば, [xkanjiskip](#) は開き括弧の後には挿入されるべきではない (「(あ」と「(あ」を比べてみよ). LuaTeX-ja では [xkanjiskip](#) をある文字の前／後に挿入するかどうかを, **J**Achar に対しては `jaxspmode` を, **A**Lchar に対しては `alxspmode` をそれぞれ変えることで制御することができる.

```
1 \ltjsetparameter{jaxspmode={`あ,preonly},
                  alxspmode={`\!,postonly}}          p あq い! う
2 p あq い! う
```

2 つ目の引数の `preonly` は「[xkanjiskip](#) の挿入はこの文字の前でのみ許され, 後では許さない」ことを意味する. 他に指定可能な値は `postonly`, `allow`, `inhibit` である.

なお, 現行の仕様では, `jaxspmode`, `alxspmode` はテーブルを共有しており, 上のコードの 1 行目を次のように変えても同じことになる:

```
\ltjsetparameter{alxspmode={`あ,preonly}, jaxspmode={`\!,postonly}}
```

また, これら 2 パラメータには数値で値を指定することもできる (7.1 節を参照).

もし全ての `kanjiskip` と `xkanjiskip` の挿入を有効化/無効化したければ、それぞれ `autospacing` と `autoxspacing` を `true/false` に設定すればよい。

4.4 ベースラインの移動

和文フォントと欧文フォントを合わせるためには、時々どちらかのベースラインの移動が必要になる。pTeX ではこれは `\ybaselineshift` を設定することでなされていた (`ALchar` のベースラインがその分だけ下がる)。しかし、日本語が主ではない文書に対しては、欧文フォントではなく和文フォントのベースラインを移動した方がよい。このため、LuaTeX-japan では欧文フォントのベースラインのシフト量 (`ybaselineshift` パラメータ) と和文フォントのベースラインのシフト量 (`yjbaselineshift` パラメータ) を独立に設定できるようになっている。

```
1 \vrule width 150pt height 0.4pt depth 0pt
   \hskip-120pt
2 \ltjsetparameter{yjbaselineshift=0pt,
   yalbaselineshift=0pt}abcあいう
   _____ abc あいう abc あいう _____
3 \ltjsetparameter{yjbaselineshift=5pt,
   yalbaselineshift=2pt}abcあいう
```

上の例において引かれている水平線がベースラインである。

この機能には面白い使い方がある：2つのパラメータを適切に設定することで、サイズの異なる文字を中心線に揃えることができる。以下は一つの例である (値はあまり調整されていないことに注意)：

```
1 xyz漢字
2 {\scriptsize
3   \ltjsetparameter{yjbaselineshift=-1pt,
   xyz 漢字 XYZ ひらがな abc かな
4   yalbaselineshift=-1pt}
5   XYZひらがな
6 }abcかな
```

■数式における挙動：pTeX との違い `ALchar` のベースラインを補正する `ybaselineshift` パラメータはほぼ pTeX における `\ybaselineshift` に対応しているものであるが、数式中の挙動は異なっているので注意が必要である。例えば、表 4 のように、数式中に明示的に現れた `\hbox` は、

- pTeX では、ボックス全体が `\ybaselineshift` だとシフトされるので、表 4 中の“い”のように、ボックス中の和文文字は `\ybaselineshift` だけシフトされ、一方、“for all”のように、ボックス内の欧文文字は 2 重にシフトされることになる。
- 一方、LuaTeX-japan ではそのようなことはおこらず、数式中に明示的に現れた `\hbox` はシフトしない。そのため、表 4 中の“い”も“for all”も、それぞれ本文中に書かれたときと同じ上下位置に組まれる。

表 4. 数式関係のベースライン補正 (yalbaselineshift = 10 pt)

入力	数式 abc: $\$あ a\hbox{い}\$, \int_0^x t dt = x^2/2, \Phi \vdash F(x) \text{ for all } x \in A$
pTeX	数式 あ abc: $aい, \int_0^x t dt = x^2/2, \Phi \vdash F(x) \text{ for all } x \in A$
LuaTeX-ja	数式 あ い abc: $aい, \int_0^x t dt = x^2/2, \Phi \vdash F(x) \text{ for all } x \in A$

表 5. \kcatcode in upTeX

\kcatcode	意図	制御綴中に使用	文字ウィドウ処理*	直後での改行
15	non-cjk		(treated as usual L ^A T _E X)	
16	kanji	Y	Y	ignored
17	kana	Y	Y	ignored
18	other	N	N	ignored
19	hangul	Y	Y	space

文字ウィドウ処理*: 「漢字が一文字だけ次の行に行くのを防ぐ」 `\jcharwidowpenalty` が、その文字の直前に挿入されるか否か、を示す。

第 II 部

リファレンス

5 LuaTeX-ja における \catcode

5.1 予備知識：pTeX と upTeX における \kcatcode

pTeX, upTeX においては、和文文字が制御綴内で利用できるかどうかは `\kcatcode` の値によって決定されるのであった。詳細は表 5 を参照されたい。

pTeX では `\kcatcode` は JIS X 0208 の区単位、upTeX では概ね Unicode ブロック単位^{*3}で設定可能になっている。そのため、pTeX と upTeX の初期状態では制御綴内で使用可能な文字が微妙に異なっている。

5.2 LuaTeX-ja の場合

LuaTeX-ja では、従来の pTeX・upTeX における `\kcatcode` の役割を分割している：

欧文/和文の区別 (upTeX) `\ltjdefcharrange` と `jacharrange` パラメータ (4.1 節)

制御綴中に使用可か LuaTeX 自身の `\catcode` でよい

`jcharwidowpenalty` が挿入可か `kcatcode` パラメータの最下位ビット

直後の改行の無視 日本語しか想定していないので `JAchar` については一律有効

ネイティブに Unicode 全部の文字を扱える X_YTeX や LuaTeX では、文字が制御綴内で利用できるかは通常の欧文文字と同じく `\catcode` で指定することとなる。X_YTeX における `\catcode` の初期設定は `unicode-letters.tex` 中に記述されており、LuaTeX ではそれを元にし

^{*3} U+FF00–U+FFEF (Halfwidth and Fullwidth Forms) は「全角英数字」「半角カナ」「その他」と3つに分割されており、それぞれ別々に `\kcatcode` が指定できるようになっている。

表 6. 制御綴中に使用出来る JIS X 0208 非漢字の違い

	区	点	pTeX	upTeX	LuaTeX-ja		区	点	pTeX	upTeX	LuaTeX-ja
□ (U+3000)	1	1	N	N	Y	◧ (U+FF0F)	1	31	N	N	Y
◦ (U+30FB)	1	6	N	Y	N	◨ (U+FF3C)	1	32	N	N	Y
◌ゝ (U+309B)	1	11	N	Y	N	◩ (U+FF5C)	1	35	N	N	Y
◌゜ (U+309C)	1	12	N	Y	N	⊕ (U+FF0B)	1	60	N	N	Y
◌㇀ (U+FF40)	1	14	N	N	Y	≡ (U+FF1D)	1	65	N	N	Y
◌㇁ (U+FF3E)	1	16	N	N	Y	◊ (U+FF1C)	1	67	N	N	Y
◌㇂ (U+FFE3)	1	17	N	N	Y	○ (U+FF1E)	1	68	N	N	Y
◌㇃ (U+FF3F)	1	18	N	N	Y	# (U+FF03)	1	84	N	N	Y
◌㇄ (U+30FD)	1	19	N	Y	Y	& (U+FF06)	1	85	N	N	Y
◌ゝゝ (U+30FE)	1	20	N	Y	Y	* (U+FF0A)	1	86	N	N	Y
◌㇆ (U+309D)	1	21	N	Y	Y	@ (U+FF20)	1	87	N	N	Y
◌㇇ (U+309E)	1	22	N	Y	Y	〒 (U+3012)	2	9	N	N	Y
// (U+3003)	1	23	N	N	Y	■ (U+3013)	2	14	N	N	Y
全 (U+4EDD)	1	24	N	Y	Y	◻ (U+FFE2)	2	44	N	N	Y
々 (U+3005)	1	25	N	N	Y	◌̂ (U+212B)	2	82	N	N	Y
㇀ (U+3006)	1	26	N	N	Y	ギリシャ文字 (6 区)			Y	N	Y
○ (U+3007)	1	27	N	N	Y	キリル文字 (7 区)			N	N	Y
◌㇉ (U+30FC)	1	28	N	Y	Y						

た `luatex-unicode-letters.tex` を用いている。

だが、 X_{TeX} における `\catcode` の初期設定と LuaTeX におけるそれは一致していない：

- `luatex-unicode-letters.tex` の元になった `unicode-letters.tex` が古い
- `unicode-letters.tex` の後半部では `\XeTeXcharclass` の設定を行っており、それによって漢字や仮名の `\catcode` が 11 に設定されている。

しかし、`luatex-unicode-letters.tex` ではこの「後半部」がまるごと省略されており、漢字や仮名の `\catcode` は 12 のまま。

言い換えると、 LuaTeX の初期状態では漢字や仮名を制御綴内に使用することはできない。

これでは pTeX で使用できた `\西暦` などが使えないこととなり、 LuaTeX-ja への移行で手間が生じる。そのため、 LuaTeX-ja では `unicode-letters.tex` の後半部にあたる内容を自前でパッチし、結果として X_{TeX} における初期設定と同じになるようにしている。

5.3 制御綴中に使用出来る JIS 非漢字の違い

エンジンが異なるので、 pTeX 、 upTeX 、 LuaTeX-ja において制御綴中に使用可能な JIS X 0208 の文字は異なる。異なっているところだけを載せると、表 6 のようになる。「◦」「◌ゝ」「◌゜」「=」を除けば、 LuaTeX-ja では upTeX より多くの文字が制御綴に使用可能になっている。特に重要なのは、全角空白 (U+3000) が LuaTeX-ja では制御綴中に使用可能であることである。

JIS X 0213 の範囲に広げると、差異はさらに大きくなる。詳細については例えば <https://github.com/h-kitagawa/kct> 中の `kct-uni-out.pdf` などを参照すること。

表 7. LuaTeX-ja に同梱されている JFM の違い

	jfm-ujis.lua	jfm-jis.lua	jfm-min.lua
例 1[6]	ある日モモちゃん んがお使いで迷 子になって泣き ました。	ある日モモちゃん んがお使いで迷 子になって泣き ました。	ある日モモちゃん んがお使いで迷 子になって泣き ました。
例 2	ちょっと！何 漢	ちょっと！何 漢	ちょっと！何 漢
Bounding Box			

6 フォントメトリックと和文フォント

6.1 \jfont 命令

フォントを和文フォントとして読み込むためには、`\jfont` を `\font` プリミティブの代わりに用いる。`\jfont` の文法は `\font` と同じである。LuaTeX-ja は `luaotfload` パッケージを自動的に読み込むので、TrueType/OpenType フォントに `feature` を指定したものを和文フォントとして用いることができる：

```
1 \jfont\tradgt={file:KozMinPr6N-Regular.otf:script=latn};%
2 +trad;-kern;jfm=ujis} at 14pt
3 \tradgt 当／体／医／区
```

なお、`\jfont` で定義された制御綴（上の例だと `\tradgt`）は `font.def` トークンではなくマクロである。従って、`\fontname\tradgt` のような入力はエラーとなる。以下では `\jfont` で定義された制御綴を `(jfont_cs)` で表す。

■**JFM** 「はじめに」の節で述べたように、JFM は文字と和文組版で自動的に挿入されるグルー／カーンの寸法情報を持っている。JFM の構造は次の節で述べる。`\jfont` 命令の呼び出しの際には、どの JFM を用いるのかを以下のキーで指定する必要がある：

`jfm=<name>` JFM の名前を指定する。もし以前に指定された JFM が読み込まれていなければ、`jfm-<name>.lua` を読み込む。以下の JFM が LuaTeX-ja には同梱されている：

`jfm-ujis.lua` LuaTeX-ja の標準 JFM である。この JFM は upTeX で用いられる UTF/OTF パッケージ用の和文用 TFM である `upnmlminr-h.tfm` を元にしていて、`luatexja-otf` パッケージを使うときはこの JFM を指定するべきである。

`jfm-jis.lua` pTeX で広く用いられている「JIS フォントメトリック」`jis.tfm` に相当する JFM である。`jfm-ujis.lua` とこの `jfm-jis.lua` の主な違いは、`jfm-ujis.lua` ではほとんどの文字が正方形であるのに対し、`jfm-jis.lua` では横長の長方形である。

`jfm-min.lua` pTeX に同梱されているデフォルトの和文用 TFM である `min10.tfm` に相当する JFM である。この JFM と他の 2 つの JFM の間には表 7 に示すような特筆すべき違いがある。

`jfmvar=<string>` 標準では、JFM とサイズが同じで、実フォントだけが異なる 2 つの和文フォントは「区別されない」。例えば下の例において、最初の「`）`」と「`【`」の実フォントは異なるが、JFM もサイズも同じなので、普通に「`）`」「`【`」と入力した時と同じように半角空きとなる。

しかし、時には……

ダイナミックダイクマ	ダイナミックダイクマ
ダイナミックダイクマ	ダイナミックダイクマ
ダイナミックダイクマ	ダイナミックダイクマ
ダイナミックダイクマ	ダイナミックダイクマ

```

1 \newcommand\test{\vrule ダイナミックダイクマ\vrule\}
2 \jfont\KMFw = KozMinPr6N-Regular:jfm=prop;-kern at 17pt
3 \jfont\KMFk = KozMinPr6N-Regular:jfm=prop at 17pt % kern is activated
4 \jfont\KMPw = KozMinPr6N-Regular:jfm=prop;script=dflt;+pwid;-kern at 17pt
5 \jfont\KMPk = KozMinPr6N-Regular:jfm=prop;script=dflt;+pwid;+kern at 17pt
6 \begin{multicols}{2}
7 \ltjsetparameter{kanjiskip=0pt}
8 {\KMFw\test \KMFk\test \KMPw\test \KMPk\test}
9
10 \ltjsetparameter{kanjiskip=3pt}
11 {\KMFw\test \KMFk\test \KMPw\test \KMPk\test}
12 \end{multicols}

```

図 1. Kerning information and kanjiskip

```

1 \ltjsetparameter{differentjfm=both}
2 \jfont\F=file:KozMinPr6N-Regular.otf:jfm=ujis
3 \jfont\G=file:KozGoPr6N-Medium.otf:jfm=ujis
4 \jfont\H=file:KozGoPr6N-Medium.otf:jfm=ujis;jfmvar=
   hoge
5
6 \F ) {\G 【】 } ( % halfwidth space
7 ) {\H 『』 } ( % fullwidth space
8
9 \ltjsetparameter{differentjfm=paverage}

```

■**ペアカーニング情報の使用** いくつかのフォントはグリフ間のスペースについての情報を持っている。このカーニング情報は以前の LuaTeX-ja とはあまり相性が良くなかったが、本バージョンではカーニングによる空白はイタリック補正と同様に扱うことになっている。つまり、カーニング由来の空白と JFM 由来のグルー・カーンは同時に入りうる。図 1 を参照。

- `\jfont` や、NFSS2 用の命令 (3.1 節, 9.1 節) における指定ではカーニング情報は標準で使用するようになっているようである。言い換えれば、カーニング情報を使用しない設定にするには、面倒でも

```

\jfont\hoge=KozMinPr6N-Regular:jfm=ujis;-kern at 3.5mm
\DeclareFontShape{JY3}{fuga}{m}{n} {<-> s*KozMinPr6N-Regular:jfm=ujis;-kern}{}

```

のように、`-kern` という指定を自分で追架しなければいけない。

- 一方、`luatexja-fontspec` の提供する `\setmainfont` などの命令の標準設定ではカーニング情報は使用しない (`Kerning=0ff`) ことになっている。これは以前のバージョンの LuaTeX-ja との互換性のためである。

■**extend** と **slant** OpenType font feature と見かけ上同じような形式で指定できるものに、

extend= $\langle extend \rangle$ 横方向に $\langle extend \rangle$ 倍拡大する。

slant= $\langle slant \rangle$ $\langle slant \rangle$ に指定された割合だけ傾ける。

の 2 つがある。 **extend** や **slant** を指定した場合は、それに応じた JFM を指定すべきである*4。例えば、次の例では無理やり通常の JFM を使っているために、文字間隔やイタリック補正量が正しくない：

```
1 \jfont\E=file:KozMinPr6N-Regular.otf:extend=1.5;jfm=ujis;-
   kern
2 \E あいうえお                                あいうえお
3                                                  あいうえお
4 \jfont\S=file:KozMinPr6N-Regular.otf:slant=1;jfm=ujis;-kern
5 \S あいう\ABC
```

6.2 psft プリフィックス

`luaotfload` で使用可能になった `file:` と `name:` のプリフィックスに加えて、`\jfont` (と `\font` プリミティブ) では `psft:` プリフィックスを用いることができる。このプリフィックスを用いることで、PDF には埋め込まれない「名前だけの」和文フォントを指定することができる。「標準的な」非埋め込み和文フォント、つまり「Ryumin-Light」「GothicBBB-Medium」の指定でこのプリフィックスが使われる。

`psft` プリフィックスの下では `+jp90` などの OpenType font feature の効力はない。非埋込フォントを PDF に使用すると、実際にどのようなフォントが表示に用いられるか予測できないからである。 `extend` と `slant` 指定は単なる変形のため `psft` プリフィックスでも使用可能である。

■**cid** キー 標準で `psft:` プリフィックスで定義されるフォントは日本語用のものであり、Adobe-Japan1-6 の CID に対応したものとなる。しかし、LuaTeX-ja は中国語の組版にも威力を発揮することが分かり、日本語フォントでない非埋込フォントの対応も必要となった。そのために追加されたのが `cid` キーである。

`cid` キーに値を指定すると、その CID を持った非埋込フォントを定義することができる：

```
1 \jfont\testJ={psft:Ryumin-Light:cid=Adobe-Japan1-6;jfm=jis} % Japanese
2 \jfont\testD={psft:Ryumin-Light:jfm=jis} % default value is Adobe-
   Japan1-6
3 \jfont\testC={psft:AdobeMingStd-Light:cid=Adobe-CNS1-6;jfm=jis} % Traditional Chinese
4 \jfont\testG={psft:SimSun:cid=Adobe-GB1-5;jfm=jis} % Simplified Chinese
5 \jfont\testK={psft:Batang:cid=Adobe-Korea1-2;jfm=jis} % Korean
```

上のコードでは中国語・韓国語用フォントに対しても JFM に日本語用の `jfm-jis.lua` を指定しているので注意されたい。

今のところ、LuaTeX-ja は上のサンプルコード中に書いた 4 つの値しかサポートしていない。

```
\jfont\test={psft:Ryumin-Light:cid=Adobe-Japan2;jfm=jis}
```

のようにそれら以外の値を指定すると、エラーが発生する：

```
1 ! Package luatexja Error: bad cid key `Adobe-Japan2'.
2
3 See the luatexja package documentation for explanation.
```

*4 LuaTeX-ja では、これらに対する JFM を特に提供することはしない予定である。

```

4 Type H <return> for immediate help.
5 <to be read again>
6             \par
7 1.78
8
9 ? h
10 I couldn't find any non-embedded font information for the CID
11 `Adobe-Japan2'. For now, I'll use `Adobe-Japan1-6'.
12 Please contact the LuaTeX-ja project team.
13 ?

```

6.3 JFM ファイルの構造

JFM ファイルはただ一つの関数呼び出しを含む Lua スクリプトである：

```
luatexja.jfont.define_jfm { ... }
```

実際のデータは上で { ... } で示されたテーブルの中に格納されている。以下ではこのテーブルの構造について記す。なお、JFM ファイル中の長さは全て design-size を単位とする浮動小数点数であることを注意する。

`dir`= $\langle direction \rangle$ (必須)

JFM の書字方向。現時点では横書き ('yoko') のみがサポートされる。

`zw`= $\langle length \rangle$ (必須)

「全角幅」の長さ。

`zh`= $\langle length \rangle$ (必須)

「全角高さ」(height + depth) の長さ。通常は全角幅と同じ長さになるだろう。

`kanjiskip`={ $\langle natural \rangle$, $\langle stretch \rangle$, $\langle shrink \rangle$ } (任意)

理想的な `kanjiskip` の量を指定する。4.2 節で述べたように、もし `kanjiskip` が `\maxdimen` の値ならば、このフィールドで指定された値が実際には用いられる (指定なしは 0pt として扱われる)。 $\langle stretch \rangle$ と $\langle shrink \rangle$ のフィールドも design-size が単位であることに注意せよ。

`xkanjiskip`={ $\langle natural \rangle$, $\langle stretch \rangle$, $\langle shrink \rangle$ } (任意)

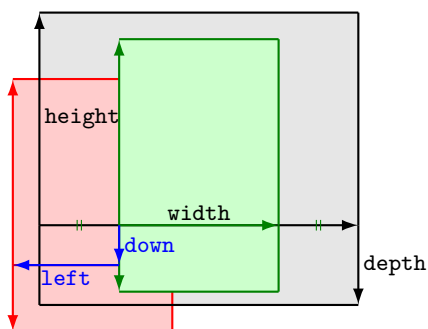
`kanjiskip` フィールドと同様に、`xkanjiskip` の理想的な量を指定する。

■**文字クラス** 上記のフィールドに加えて、JFM ファイルはそのインデックスが自然数であるいくつかのサブテーブルを持つ。インデックスが $i \in \omega$ であるテーブルは**文字クラス i** の情報を格納する。少なくとも、文字クラス 0 は常に存在するので、JFM ファイルはインデックスが [0] のサブテーブルを持たなければならない。それぞれのサブテーブル (そのインデックスを i で表わす) は以下のフィールドを持つ：

`chars`={ $\langle character \rangle$, ...} (文字クラス 0 を除いて必須)

このフィールドは文字クラス i に属する文字のリストである。このフィールドは $i = 0$ の場合には任意である (文字クラス 0 には、0 以外の文字クラスに属するものを除いた全ての `JAchar` が属するから)。このリスト中で文字を指定するには、以下の方法がある：

- Unicode におけるコード番号
- 「'あ'」のような、文字それ自体
- 「'あ*'」のような、文字それ自体の後にアスタリスクをつけたもの
- いくつかの「仮想的な文字」(後に説明する)



align フィールドの値が'middle' である和文文字を含むノードを考えよう.

- 黒色の長方形はノードの枠である. その幅, 高さ, 深さは JFM によって指定される.
- align フィールドは middle なので, 実際のグリフは水平方向の中心に配置される (緑色の長方形).
- さらに, グリフは left と down の値に従ってシフトされる. 最終的な実際のグリフの位置は赤色の長方形で示された位置になる.

図 2. 「実際の」グリフの位置.

`width=<length>, height=<length>, depth=<length>, italic=<length>` (必須)

文字クラス i に属する文字の幅, 高さ, 深さ, イタリック補正の量を指定する. 文字クラス i に属する全ての文字は, その幅, 高さ, 深さがこのフィールドで指定した値であるものとして扱われる.

例外として, `width` フィールドには数値以外に 'prop' が指定可能である. この場合, 文字の幅はその「実際の」グリフの幅となる. OpenType の prop feature と併用すれば, これによってプロポーショナル組を行うことができる.

`left=<length>, down=<length>, align=<align>`

これらのフィールドは実際のグリフの位置を調整するためである. align フィールドに指定できる値は 'left', 'middle', 'right' のいずれかである. もしこれら 3 つのフィールドのうちの 1 つが省かれた場合, left と down は 0, align フィールドは 'left' であるものとして扱われる. これら 3 つのフィールドの意味については図 2 で説明する.

多くの場合, left と down は 0 である一方, align フィールドが'middle' や'right' であることは珍しいことではない. 例えば, align フィールドを'right' に指定することは, 文字クラスが開き括弧類であるときに実際必要である.

`kern={ [j]=<kern>, [j']={<kern>, [<ratio>]}, ... }`

`glue={ [j]={<width>, <stretch>, <shrink>, [<priority>], [<ratio>]}, ... }` 文字クラス i の文字と j の文字の間に挿入される kern や glue の量を指定する.

`<priority>` は `luatexja-adjust` による優先順位付き行長調整 (10.3 節) が有効なときのみ意味を持つ. このフィールドは省略可能であり, 行調整処理におけるこの glue の優先度を -2 から $+2$ の間の整数で指定する. 大きい値ほど「伸びやすく, 縮みやすい」ことを意味する. 省略時の値は 0 であり, 範囲外の値が指定されたときの動作は未定義である.

`<ratio>` も省略可能フィールドであり, -1 から $+1$ の実数値をとる. 省略時の値は 0 である.

- -1 はこのグルーが「前の文字」由来であることを示す.
- $+1$ はこのグルーが「後の文字」由来であることを示す.
- それ以外の値は, 「前の文字」由来のグルーと「後の文字」由来のグルーが混合されていることを示す.

なお, このフィールドの値は `differentjfm` の値が `pleft`, `pright`, `paverage` の値のときのみ実際に用いられる.

例えば, [7] では, 句点と中点の間には, 句点由来の二分空きと中点由来の四分空きが挿入されるが, この場合には

- `<width>` には $0.5 + 0.25 = 0.75$ を指定する.

- $\langle ratio \rangle$ には次の値を指定する.

$$-1 \cdot \frac{0.5}{0.5 + 0.25} + 1 \cdot \frac{0.25}{0.5 + 0.25} = -\frac{1}{3}$$

`end_stretch=\langle kern \rangle`

`end_shrink=\langle kern \rangle` これらのフィールドは省略可能である. 優先順位付き行長調整が有効であり, かつ現在の文字クラスの文字が行末に来た時に, 行長を詰める調整・伸ばす調整のためにこの文字と行末の間に挿入可能なカーンの大きさを指定する.

■**文字クラスの決定** 文字からその文字の属する文字クラスを算出する過程は少々複雑である. 次の内容を一部に含んだ `jfm-test.lua` を用いて説明する.

```
[0] = {
  chars = { '漢', 'ヒ*' },
  align = 'left', left = 0.0, down = 0.0,
  width = 1.0, height = 0.88, depth = 0.12, italic=0.0,
},
[2000] = {
  chars = { '。', '、', '*', 'ヒ' },
  align = 'left', left = 0.0, down = 0.0,
  width = 0.5, height = 0.88, depth = 0.12, italic=0.0,
},
```

句点「。」の幅は二分であるので

```
1 \jfont\fontfile:KozMinPr6N-Regular.otf:jfm=test;+vert
2 \setbox0\hbox{\a 。 \inhibitglue 漢} 20.0pt
3 \the\wd0
```

では, 全角二分 (15.0 pt) となるのが自然……と思うかもしれないが, 上の実行結果では 20 pt となっている. それは以下の事情によるものである:

1. `vert` feature によって句点 (U+3002) が縦書き用のグリフと置き換わる (`luaotfload` による処理).
2. この縦書き用句点のグリフは U+FE12 であるため, その文字クラスは 0 となる.
3. 以上により文字クラス 0 とみなされるため, 結果として「。」の幅は全角だと認識されてしまう.

この例は, **文字クラスの決定は font feature の適用によるグリフ置換の後に行われることを示している.**

但し, 「、*」のようにアスタリスクつきの指定があると, 状況は異なる.

```
1 \jfont\fontfile:KozMinPr6N-Regular.otf:jfm=test;+vert 漢 漢
2 \a 漢、 \inhibitglue 漢
```

ここで, 読点「、」(U+3001) の文字クラスは, 以下のようにして決まる.

1. とりあえず句点の時と同じように, `luaotfload` によって縦書き用読点のグリフに置き換わる.
2. 置換後のグリフは U+FE11 であり, そのままでは文字クラスは 0 と判定される.
3. ところが, JFM には「、*」指定があるので, 置換前の横書き用読点のグリフによって文字クラスを判定する.
4. 結果として, 上の出力例中の読点の文字クラスは 2000 となる.

なお, 「ヒ*」のようにアスタリスクつきの指定があっても, 置換後のグリフで判定した文字クラスの値が 0 でなければ, そちらをそのまま作用する.

1 \jfont\A=file:KozMinPr6N-Regular.otf:jfm=test;+hwid

漢ヒ

2 \A 漢ヒ

上の例では、hwid feature により、「ヒ」が半角の「ヒ」に置き換わるが、文字クラスは「ヒ」の属する 0 ではなく、「ヒ」の属する 2000 となる。

■**仮想的な文字** 上で説明した通り、chars フィールド中にはいくつかの「特殊文字」も指定可能である。これらは、大半が pTeX の JFM グルーの挿入処理ではみな「文字クラス 0 の文字」として扱われていた文字であり、その結果として pTeX より細かい組版調整ができるようになっている。以下でその一覧を述べる：

'boxbdd' hbox の先頭と末尾、及びインデントされていない (\noindent で開始された) 段落の先頭を表す。

'parbdd' 通常の (\noindent で開始されていない) 段落の先頭。

'jcharbdd' 和文文字と「その他のもの」(欧文文字, glue, kern 等) との境界。

-1 行中数式と地の文との境界。

■**pTeX 用和文用 TFM の移植** 以下に、pTeX 用に作られた和文用 TFM を LuaTeX-ja 用に移植する場合の注意点を挙げておく。

- 実際に出力される和文フォントのサイズが design size となる。このため、例えば 1zw が design size の 0.962216 倍である JIS フォントメトリック等を移植する場合は、次のようにすべきである：

- JFM 中の全ての数値を 1/0.962216 倍しておく。

- TeX ソース中で使用するところで、サイズ指定を 0.962216 倍にする。LaTeX でのフォント宣言なら、例えば次のように：

```
\DeclareFontShape{JY3}{mc}{m}{n}{<-> s*[0.962216] psft:Ryumin-Light:jfm=jis}{}
```

- 上に述べた特殊文字は、'boxbdd' を除き文字クラスを全部 0 とする (JFM 中に単に書かなければよい)。

- 'boxbdd' については、そのみで一つの文字クラスを形成し、その文字クラスに関してはグルー／カーンの設定はしない。

これは、pTeX では、hbox の先頭・末尾とインデントされていない (\noindent で開始された) 段落の先頭には JFM グルーは入らないという仕様を実現させるためである。

- pTeX の組版を再現させようというのが目的であれば以上の注意を守れば十分である。

ところで、pTeX では通常の段落の先頭に JFM グルーが残るという仕様があるので、段落先頭の開き括弧は全角二分下がりになる。全角下がりを実現させるには、段落の最初に手動で \inhibitglue を追加するか、あるいは \everypar のハックを行い、それを自動化させるしかなかった。

一方、LuaTeX-ja では、'parbdd' によって、それが JFM 側で調整できるようになった。例えば、LuaTeX-ja 同梱の JFM のように、'boxbdd' と同じ文字クラスに 'parbdd' を入れれば全角下がりとなる。

1 \jfont\g=KozMinPr6N-Regular:jfm=test \g

2 \parindent1\zw\noindent{}

3 \par 「◆◆←二分下がり

4 \par 【◆◆←全角下がり

5 \par [◆◆←全角二分下がり



「◆◆←二分下がり

【◆◆←全角下がり

[◆◆←全角二分下がり

表 8. 和文数式フォントに対する命令

和文フォント	欧文フォント
$\backslash\text{jfam} \in [0, 256)$	$\backslash\text{fam}$
$\text{jatextfont}=\{\langle\text{jfam}\rangle, \langle\text{jfont_cs}\rangle\}$	$\backslash\text{textfont}\langle\text{fam}\rangle=\langle\text{font_cs}\rangle$
$\text{jascriptfont}=\{\langle\text{jfam}\rangle, \langle\text{jfont_cs}\rangle\}$	$\backslash\text{scriptfont}\langle\text{fam}\rangle=\langle\text{font_cs}\rangle$
$\text{jascriptscriptfont}=\{\langle\text{jfam}\rangle, \langle\text{jfont_cs}\rangle\}$	$\backslash\text{scriptscriptfont}\langle\text{fam}\rangle=\langle\text{font_cs}\rangle$

但し、 $\backslash\text{everypar}$ を利用している場合にはこの仕組みは正しく動かない。そのような例としては箇条書き中の $\backslash\text{item}$ で始まる段落があり、`ltjclasses` では人工的に「`'parbdd'` の意味を持つ」`whatsit` ノードを作ることによって対処している^{*5}。

6.4 数式フォントファミリ

$\text{T}_{\text{E}}\text{X}$ は数式フォントを 16 のファミリ^{*6}で管理し、それぞれのファミリは 3 つのフォントを持っている： $\backslash\text{textfont}$ 、 $\backslash\text{scriptfont}$ そして $\backslash\text{scriptscriptfont}$ である。

Lua $\text{T}_{\text{E}}\text{X}$ -ja の数式中での和文フォントの扱いも同様である。表 8 は数式フォントファミリに対する $\text{T}_{\text{E}}\text{X}$ のプリミティブと対応するものを示している。 $\backslash\text{fam}$ と $\backslash\text{jfam}$ の値の間には関係はなく、適切な設定の下では $\backslash\text{fam}$ と $\backslash\text{jfam}$ の両方に同じ値を設定することができる。

6.5 コールバック

Lua $\text{T}_{\text{E}}\text{X}$ 自体のものに加えて、Lua $\text{T}_{\text{E}}\text{X}$ -ja もコールバックを持っている。これらのコールバックには、他のコールバックと同様に `luatexbase.add_to_callback` 関数などを用いることでアクセスすることができる。

luatexja.load_jfm コールバック このコールバックを用いることで JFM を上書きすることができる。このコールバックは新しい JFM が読み込まれるときに呼び出される。

```
1 function (<table> jfm_info, <string> jfm_name)
2   return <table> new_jfm_info
3 end
```

引数 `jfm_info` は JFM ファイルのテーブルと似たものが格納されるが、クラス 0 を除いた文字のコードを含んだ `chars` フィールドを持つ点が異なる。

このコールバックの使用例は `ltjarticle` クラスにあり、`jfm-min.lua` 中の `'parbdd'` を強制的にクラス 0 に割り当てている。

luatexja.define_jfont コールバック このコールバックと次のコールバックは組をなしており、Unicode 中に固定された文字コード番号を持たない文字を非零の文字クラスに割り当てることができる。このコールバックは新しい和文フォントが読み込まれたときに呼び出される。

```
1 function (<table> jfont_info, <number> font_number)
2   return <table> new_jfont_info
3 end
```

`jfont_info` は最低限以下のフィールドを持つが、これらを書き換えてはならない：

size 実際に使われるフォントサイズ (sp 単位)。 $1\text{ sp} = 2^{-16}\text{ pt}$ 。

^{*5} `ltjclasses.dtx` を参照されたい。JFM 側で一部の対処ができることにより、`jsclasses` のように if 文の判定はしていない。

^{*6} Omega, Aleph, Lua $\text{T}_{\text{E}}\text{X}$, そして ϵ -(u)p $\text{T}_{\text{E}}\text{X}$ では 256 の数式ファミリを扱うことができるが、これをサポートするために plain $\text{T}_{\text{E}}\text{X}$ と L $\text{A}^{\text{M}}\text{T}_{\text{E}}\text{X}$ では外部パッケージを読み込む必要がある。

zw, zh, kanjiskip, xkanjiskip JFM ファイルで指定されているそれぞれの値をフォントサイズに合わせてスケールしたものを sp 単位で格納している。

jfm 利用されている JFM を識別するための番号。

var `\jfont` で指定された `jfmvar` キーの値 (未指定のときは空文字列)。

chars 文字コードから文字クラスへの対応が記述されたテーブル。

JFM 内の `[i].chars={⟨character⟩, ...}` という指定は `chars={⟨[⟨character⟩]=i, ...}` という形式に変換されている。

char_type $i \in \omega$ に対して, `char_type[i]` は文字クラス i の文字の寸法を格納しており, 以下のフィールドを持つ。

- `width, height, depth, italic, down, left` は JFM で指定されているそれぞれの値をスケールしたものである。
- `align` は JFM で指定されている値によって,

$$\begin{cases} 0 & \text{'left' や省略時} \\ 0.5 & \text{'middle' } \\ 1 & \text{'right' } \end{cases}$$

のいずれかの値をとる。

- $j \in \omega$ に対して, `[j]` は文字クラス i の文字と j の文字の間に挿入される kern や glue を格納している。間に入るものが kern であれば, このフィールドの値は `[j]={false, ⟨kern_node⟩, ⟨ratio⟩}` である。⟨kern_node⟩ は kern を表すノードそのものである*7。glue であれば, `[j]={false, ⟨spec_node⟩, ⟨ratio⟩, ⟨icflag⟩}` である。⟨spec_node⟩ は glue の長さを表すノードそのものであり, `⟨icflag⟩ = from_jfm + ⟨priority⟩` である。

戻り値の `new_jfont_info` テーブルも上に述べたフィールドをそのまま含まなければならないが, それ以外にユーザが勝手にフィールドを付け加えることは自由である。`font_number` はフォント番号である。

これと次のコールバックの良い使用例は `luatexja-otf` パッケージであり, JFM 中で Adobe-Japan1 CID の文字を "AJ1-xxx" の形で指定するために用いられている。

luatexja.find_char_class コールバック このコールバックは LuaTeX-já が `chr_code` の文字がどの文字クラスに属するかを決定しようとする際に呼び出される。このコールバックで呼び出される関数は次の形をしていなければならない:

```
1 function (⟨number⟩ char_class, ⟨table⟩ jfont_info, ⟨number⟩ chr_code)
2   if char_class~=0 then return char_class
3   else
4     ....
5     return (⟨number⟩ new_char_class or 0)
6   end
7 end
```

引数 `char_class` は LuaTeX-já のデフォルトルーチンか, このコールバックの直前の関数呼び出しの結果を含んでおり, したがってこの値は 0 ではないかもしれない。さらに, 戻り値の `new_char_class` は `char_class` が非零のときには `char_class` の値と同じであるべきで, そうでないときは LuaTeX-já のデフォルトルーチンを書き換えることになる。

luatexja.set_width コールバック このコールバックは LuaTeX-já が `JAchar` の寸法と位置を調節するためにその `glyph_node` をカプセル化しようとする際に呼び出される。

*7 本バージョンでは利用可能ならばノードのアクセス手法に `direct access model` を用いている。そのため, 例えば LuaTeX beta-0.78.2 では, 単なる自然数のようにしか見えないことに注意。

```

1 function (<table> shift_info, <table> jfont_info, <number> char_class)
2   return <table> new_shift_info
3 end

```

引数 `shift_info` と戻り値の `new_shift_info` は `down` と `left` のフィールドを持ち、これらの値は文字の下／左へのシフト量 (sp 単位) である。

良い例が `test/valign.lua` である。このファイルが読み込まれた状態では、JFM 内で規定された文字クラス 0 の文字における (高さ) : (深さ) の比になるように、実際のフォントの出力上下位置が自動調整される。例えば、

- JFM 側の設定 : (高さ) = $88x$, (深さ) = $12x$ (和文 OpenType フォントの標準値)
 - 実フォント側の数値 : (高さ) = $28y$, (深さ) = $5y$ (和文 TrueType フォントの標準値)
- となっていたとする。すると、実際の文字の出力位置は、以下の量だけ上にせらされることとなる :

$$\frac{88x}{88x + 12x}(28y + 5y) - 28y = \frac{26}{25}y = 1.04y.$$

7 パラメータ

7.1 `\ltjsetparameter`

先に述べたように、`\ltjsetparameter` と `\ltjgetparameter` は LuaTeX-ja のほとんどのパラメータにアクセスするための命令である。LuaTeX-ja が pTeX のような文法 (例えば、`\prebreakpenalty`) = 10000) を採用しない理由の一つは、LuaTeX のソースにおける `hpack_filter` コールバックの位置にある。11 章を参照。

`\ltjsetparameter` と `\ltjglobalsetparameter` はパラメータを指定するための命令である。これらは `<key>=<value>` のリストを引数としてとる。許されるキーの一覧は次の節にある。`\ltjsetparameter` と `\ltjglobalsetparameter` の違いはスコープの違いのみで、`\ltjsetparameter` はローカルな指定、`\ltjglobalsetparameter` はグローバルな指定を行う。これらは他のパラメータ指定と同様に `\globaldefs` の値に従う。

以下は `\ltjsetparameter` に指定することができるパラメータの一覧である。[`\cs`] は pTeX における対応物を示す。また、それぞれのパラメータの右上にある記号には次の意味がある :

- “*” : 段落や `hbox` の終端での値がその段落 / `hbox` 全体で用いられる。
- “†” : 指定は常にグローバルになる。

`jcharwidowpenalty=<penalty>*` [`\jcharwidowpenalty`] パラグラフの最後の字が孤立して改行されるのを防ぐためのペナルティの値。このペナルティは (日本語の) 句読点として扱われない最後の `JAchar` の直後に挿入される。

`kcatcode={<chr_code>,<natural number>}*` 文字コードが `<chr_code>` の文字を持つ付加的な属性値 (attribute)。現在のバージョンでは、`<natural number>` の最下位ビットが、その文字が句読点とみなされるかどうかを表している (上の `jcharwidowpenalty` の記述を参照)。

`prebreakpenalty={<chr_code>,<penalty>}*` [`\prebreakpenalty`] 文字コード `<chr_code>` の `JAchar` が行頭にくることを抑止するために、この文字の前に挿入/追加されるペナルティの量を指定する。

例えば閉じ括弧「`」` は絶対に行頭にきてはならないので、

```
\ltjsetparameter{prebreakpenalty={`」},10000}}
```

と、最大値の 10000 が標準で指定されている。他にも、小書きのカナなど、絶対禁止というわ

けではないができれば行頭にはきて欲しくない場合に、0 と 10000 の間の値を指定するのも有用であろう。

`postbreakpenalty={⟨chr_code⟩,⟨penalty⟩}* [\postbreakpenalty]` 文字コード `⟨chr_code⟩` の **J**Achar が行末にくることを抑止するために、この文字の後に挿入/追加されるペナルティの量を指定する。

pTeX では、`\prebreakpenalty`、`\postbreakpenalty` において、

- 一つの文字に対して、pre, post どちらか一つしか指定することができなかった（後から指定した方で上書きされる）。
- pre, post 合わせて 256 文字分の情報を格納することしかできなかった。という制限があったが、LuaTeX-ja ではこれらの制限は解消されている。

`jatextfont={⟨jfam⟩,⟨jfont_cs⟩}* [TeX の \textfont]`

`jascriptfont={⟨jfam⟩,⟨jfont_cs⟩}* [TeX の \scriptfont]`

`jascriptscriptfont={⟨jfam⟩,⟨jfont_cs⟩}* [TeX の \scriptscriptfont]`

`yjabaselineshift=⟨dimen⟩`

`yalbaselineshift=⟨dimen⟩ [\ybaselineshift]`

`jaxspmode={⟨chr_code⟩,⟨mode⟩}* 文字コードが ⟨chr_code⟩ の JAchar の前/後ろに xkanjiskip の挿入を許すかどうかの設定。以下の ⟨mode⟩ が許される：`

- 0, **inhibit** `xkanjiskip` の挿入は文字の前/後ろのいずれでも禁止される。
- 1, **preonly** `xkanjiskip` の挿入は文字の前では許されるが、後ろでは許されない。
- 2, **postonly** `xkanjiskip` の挿入は文字の後ろでは許されるが、前では許されない。
- 3, **allow** `xkanjiskip` の挿入は文字の前/後ろのいずれでも許される。これがデフォルトの値である。

このパラメータは pTeX の `\inhibitxspcode` プリミティブと似ているが、互換性はない。

`alxspmode={⟨chr_code⟩,⟨mode⟩}* [\xspcode]`

文字コードが `⟨chr_code⟩` の **A**Lchar の前/後ろに `xkanjiskip` の挿入を許すかどうかの設定。以下の `⟨mode⟩` が許される：

- 0, **inhibit** `xkanjiskip` の挿入は文字の前/後ろのいずれでも禁止される。
- 1, **preonly** `xkanjiskip` の挿入は文字の前では許されるが、後ろでは許されない。
- 2, **postonly** `xkanjiskip` の挿入は文字の後ろでは許されるが、前では許されない。
- 3, **allow** `xkanjiskip` の挿入は文字の前/後ろのいずれでも許される。これがデフォルトの値である。

`jaxspmode` と `alxspmode` は共通のテーブルを用いているため、これら 2 つのパラメータは互いの異名となっていることに注意する。

`autospadding=⟨bool⟩ [\autospadding]`

`autoxspacing=⟨bool⟩ [\autoxspacing]`

`kanjiskip=⟨skip⟩* [\kanjiskip]` デフォルトで 2 つの **J**Achar の間に挿入されるグルーである。通常では、pTeX と同じようにフォントサイズに比例して変わることはない。しかし、自然長が `\maxdimen` の場合は、例外的に和文フォントの JFM 側で指定されている値を採用（こちらはフォントサイズに比例）することになっている。

`xkanjiskip=⟨skip⟩* [\xkanjiskip]` デフォルトで **J**Achar と **A**Lchar の間に挿入されるグルーである。`kanjiskip` と同じように、通常ではフォントサイズに比例して変わることはないが、自然長が `\maxdimen` の場合が例外である。

`differentjfm=⟨mode⟩† JFM（もしくはサイズ）が異なる 2 つの JAchar の間にグルー/カーンをどのように入れるかを指定する。許される値は以下の通り：`

`average, both, large, small, pleft, pright, paverage`

デフォルト値は `paverage` である。各々の値による差異の詳細は 13.4 節の「『右空白』の算出」を参照してほしい。

```
jacharrange= $\langle ranges \rangle$   
kansujichar= $\{\langle digit \rangle, \langle chr\_code \rangle\}^* [\backslash kansujichar]$ 
```

7.2 `\ltjgetparameter`

`\ltjgetparameter` はパラメータの値を取得するための命令であり、常にパラメータの名前を第一引数にとる。

```
1 \ltjgetparameter{differentjfm},  
2 \ltjgetparameter{autospacing},           paverage, 1, 0.0pt plus 0.92491pt minus  
3 \ltjgetparameter{kanjiskip},           0.09242pt, 10000.  
4 \ltjgetparameter{prebreakpenalty}{ $\wedge$  }.
```

`\ltjgetparameter` の戻り値は常に文字列である。これは `tex.write()` によって出力しているため、空白「」(U+0020)を除いた文字のカテゴリコードは全て 12 (other) となる。一方、空白のカテゴリコードは 10 (space) である。

- 第 1 引数が次のいずれかの場合には、追加の引数は必要ない。
`jcharwidowpenalty`, `yjabaselineshift`, `yalbaselineshift`, `autospacing`, `autoxspacing`,
`kanjiskip`, `xkanjiskip`, `differentjfm`
`\ltjgetparameter{autospacing}` と `\ltjgetparameter{autoxspacing}` は、`true` や `false` を返すのではなく、1 と 0 のいずれかを返すことに注意。
- 第 1 引数が次のいずれかの場合には、さらに文字コードを第二引数としてとる。
`kcatcode`, `prebreakpenalty`, `postbreakpenalty`, `jaxspmode`, `alxspmode`
`\ltjgetparameter{jaxspmode}{...}` や `\ltjgetparameter{alxspmode}{...}` は、`preonly` などといった文字列ではなく、0 から 3 までの値を返す。
- `\ltjgetparameter{jacharrange}{ $\langle range \rangle$ }` は、 $\langle range \rangle$ が **J**Achar 達の範囲ならば 0 を、そうでなければ 1 を返す。「-1 番の文字範囲」は存在しないが、 $\langle range \rangle$ に -1 を指定してもエラーは発生しない (1 を返す)。
- 0-9 の数 $\langle digit \rangle$ に対して、`\ltjgetparameter{kansujichar}{ $\langle digit \rangle$ }` は、`\kansuji $\langle digit \rangle$` で出力される文字の文字コードを返す。
- 次のパラメータ名を `\ltjgetparameter` に指定することはできない。
`jatextfont`, `jascriptfont`, `jascriptscriptfont`, `jacharrange`
- `\ltjgetparameter{chartorange}{ $\langle chr_code \rangle$ }` によって $\langle chr_code \rangle$ の属する文字範囲の番号を知ることができる。
 $\langle chr_code \rangle$ に 0-127 の値を指定した場合 (このとき、 $\langle chr_code \rangle$ が属する文字範囲は存在しない) は -1 が返る。

そのため、 $\langle chr_code \rangle$ が **J**Achar か **A**Lchar かは次で知ることができる：

```
\ltjgetparameter{jacharrange}{\ltjgetparameter{chartorange}{ $\langle chr\_code \rangle$ }}  
% 0 if JAchar, 1 if ALchar
```


8 plain でも L^AT_EX でも利用可能なその他の命令

8.1 p_TE_X 互換用命令

以下の命令は p_TE_X との互換性のために実装されている。そのため、JIS X 0213 には対応せず、p_TE_X と同じように JIS X 0208 の範囲しかサポートしていない。

```
\kuten, \jis, \euc, \sjis, \jis, \kansuji
```

これら 6 命令は内部整数を引数とするが、実行結果は文字列であることに注意。

```
1 \newcount\hoge
2 \hoge="2423 %"          9251, 九二五一
3 \the\hoge, \kansuji\hoge\ 12355, い
4 \jis\hoge, \char\jis\hoge\ 一七〇一
5 \kansuji1701
```

8.2 \inhibitglue

\inhibitglue は J^Aglue の挿入を抑制する。以下は、ボックスの始めと「あ」の間、「あ」「ウ」の間にグルーが入る特別な JFM を用いた例である。

```
1 \jfont\g=file:KozMinPr6N-Regular.otf:jfm=test \g
2 \fbox{\hbox{あウあ\inhibitglue ウ}}
3 \inhibitglue\par\noindent あ1
4 \par\inhibitglue\noindent あ2
5 \par\noindent\inhibitglue あ3
6 \par\hrule\noindent あoff\inhibitglue ice
```

あ	ウあウ
あ	1
あ	2
あ	3
あ	office

この例を援用して、\inhibitglue の仕様について述べる。

- \inhibitglue の垂直モード中での呼び出しは意味を持たない。4 行目の入力で有効にならないのは、\inhibitglue の時点では垂直モードであり、\noindent の時点で水平モードになるからである。
- \inhibitglue の（制限された）水平モード中での呼び出しはその場でのみ有効であり、段落の境界を乗り越えない。さらに、\inhibitglue は上の例の最終行のように（欧文における）リガチャとカーニングを打ち消す。これは、\inhibitglue が内部的には「現在のリスト中に whatsit ノードを追加する」ことを行なっているからである。
- \inhibitglue を数式モード中で呼び出した場合はただ無視される。
- L^AT_EX で Lua_TE_X-ja を使用する場合は、\inhibitglue の代わりとして \< を使うことができる。既に \< が定義されていた場合は、Lua_TE_X-ja の読み込みで強制的に上書きされるので注意すること。

8.3 \ltjdeclarealtfont

\jfont の書式を見ればわかるように、基本的には Lua_TE_X-ja における 1 つの和文フォントに使用出来る「実際のフォント」は 1 つである。しかし、\ltjdeclarealtfont を用いると、この原則から外れることができる。

\ltjdeclarealtfont は以下の書式で使用する：

```

1 \DeclareKanjiFamily{JY3}{edm}{}
2 \DeclareFontShape{JY3}{edm}{m}{n}    {<-> s*KozMinPr6N-Regular:jfm=ujis;-kern}{}
3 \DeclareFontShape{JY3}{edm}{m}{green}{<-> s*KozMinPr6N-Regular:jfm=ujis;-kern;color=007F00
   }{}
4 \DeclareFontShape{JY3}{edm}{m}{blue} {<-> s*KozMinPr6N-Regular:jfm=ujis;-kern;color=0000FF
   }{}
5 \DeclareAlternateKanjiFont{JY3}{edm}{m}{n}{JY3}{edm}{m}{green}{"4E00-"67FF,{-2}-{-2}}
6 \DeclareAlternateKanjiFont{JY3}{edm}{m}{n}{JY3}{edm}{m}{blue}{ "6800-"9FFF}
7 {\kanjifamily{edm}\selectfont
8 日本国民は、正当に選挙された国会における代表者を通じて行動し、……}

```

日本国民は、正当に選挙された国会における代表者を通じて行動し、……

図 3. `\DeclareAlternateKanjiFont` の使用例

```
\ltjdeclarealtfont<base_font_cs><alt_font_cs><range>
```

これは「現在の和文フォント」が `<base_font_cs>` であるとき、`<range>` に属する文字は `<alt_font_cs>` を用いて組版される、という意味である。

- `<base_font_cs>`, `<alt_font_cs>` は `\jfont` によって定義された和文フォントである。
- `<range>` は文字コードの範囲を表すコンマ区切りのリストであるが、例外として負数 $-n$ は「`<base_font_cs>` の JFM の文字クラス n に属する全ての文字」を意味する。`<range>` 中に `<alt_font_cs>` 中に実際には存在しない文字が指定された場合は、その文字に対する設定は無視される。

例えば、`\hoge` の JFM が Lua_{TeX}-ja 標準の `jfm-ujis.lua` であった場合、

```
\ltjdeclarealtfont\hoge\piyo{"3000-"30FF, {-1}-{-1}}
```

は「`\hoge` を利用しているとき、`U+3000-U+30FF` と文字クラス 1 (開き括弧類) 中の文字だけは `\piyo` を用いる」ことを設定する。`{-1}-{-1}` という変わった指定の仕方をしているのは、普通に `-1` と指定したのでは正しく `-1` と読み取られないというマクロの都合による。

9 L^AT_EX 2_ε 用の命令

9.1 NFSS2 へのパッチ

Lua_{TeX}-ja の NFSS2 への日本語パッチは pL^AT_EX 2_ε で同様の役割を果たす `plfonts.dtx` をベースに、和文エンコーディングの管理等を Lua で書きなおしたものである。ここでは 3.1 節で述べていなかった命令について記述しておく。

追加の長さ変数達

pL^AT_EX 2_ε と同様に、Lua_{TeX}-ja は「現在の和文フォントの情報」を格納する長さ変数

```
\cht (height), \cdp (depth), \cHT (sum of former two),
\cwd (width), \cvs (lineskip), \chs (equals to \cwd)
```

と、その `\normalsize` 版である

```
\Cht (height), \Cdp (depth), \Cwd (width),
```

`\Cvs` (equals to `\baselineskip`), `\Chs` (equals to `\cwd`)

を定義している。なお、`\cwd` と `\zw`, また `\cHT` と `\zh` は一致しない可能性がある。なぜなら、`\cwd`, `\cHT` は「あ」の寸法から決定されるのに対し、`\zw` と `\zh` は JFM に指定された値に過ぎないからである。

`\DeclareYokoKanjiEncoding{⟨encoding⟩}{⟨text-settings⟩}{⟨math-settings⟩}`

LuaTeX-ja の NFSS2 においては、欧文フォントファミリと和文フォントファミリはそのエンコーディングによってのみ区別される。例えば、OT1 と T1 のエンコーディングは欧文フォントファミリに対するものであり、和文フォントファミリはこれらのエンコーディングを持つことはできない。このコマンドは横書き用和文フォントのための新しいエンコーディングを定義する。

`\DeclareKanjiEncodingDefaults{⟨text-settings⟩}{⟨math-settings⟩}`

`\DeclareKanjiSubstitution{⟨encoding⟩}{⟨family⟩}{⟨series⟩}{⟨shape⟩}`

`\DeclareErrorKanjiFont{⟨encoding⟩}{⟨family⟩}{⟨series⟩}{⟨shape⟩}{⟨size⟩}`

上記 3 つのコマンドはちょうど `\DeclareFontEncodingDefaults` などに対応するものである。

`\reDeclareMathAlphabet{⟨unified-cmd⟩}{⟨al-cmd⟩}{⟨ja-cmd⟩}`

和文・欧文の数式用フォントファミリを一度に変更する命令を作成する。具体的には、欧文数式用フォントファミリ変更の命令 `⟨al-cmd⟩` (`\mathrm` 等) と、和文数式用フォントファミリ変更の命令 `⟨ja-cmd⟩` (`\mathmc` 等) の 2 つを同時に行う命令として `⟨unified-cmd⟩` を (再) 定義する。実際の使用では `⟨unified-cmd⟩` と `⟨al-cmd⟩` に同じものを指定する、すなわち、`⟨al-cmd⟩` で和文側も変更させるようにするのが一般的と思われる。

本命令は

`⟨unified-cmd⟩{⟨arg⟩} → ((⟨al-cmd⟩ の 1 段展開結果){⟨ja-cmd⟩ の 1 段展開結果){⟨arg⟩}`

と定義を行うので、使用には注意が必要である：

- `⟨al-cmd⟩`, `⟨ja-cmd⟩` は既に定義されていなければならない。`\reDeclareMathAlphabet` の後に両命令の内容を再定義しても、`⟨unified-cmd⟩` の内容にそれは反映されない。
- `⟨al-cmd⟩`, `⟨ja-cmd⟩` に `\@mathrm` などと `@` をつけた命令を指定した時の動作は保証できない。

`\DeclareRelationFont{⟨ja-encoding⟩}{⟨ja-family⟩}{⟨ja-series⟩}{⟨ja-shape⟩}`

`{⟨al-encoding⟩}{⟨al-family⟩}{⟨al-series⟩}{⟨al-shape⟩}`

いわゆる「従属欧文」を設定するための命令である。前半の 4 引数で表される和文フォントファミリに対して、そのフォントに対応する「従属欧文」のフォントファミリを後半の 4 引数により与える。

`\SetRelationFont`

このコマンドは `\DeclareRelationFont` とローカルな指定であることを除いてほとんど同じである (`\DeclareRelationFont` はグローバル)。

`\userelfont`

現在の欧文フォントのエンコーディング/ファミリ/…… を、`\DeclareRelationFont` か `\SetRelationFont` で指定された現在の和文フォントファミリに対応する「従属欧文」フォントファミリに変更する。`\fontfamily` のように、有効にするためには `\selectfont` が必要である。

`\adjustbaseline`

LaTeX_{2_ε} では、`\adjustbaseline` は縦書時に「M」と「あ」の中心線を一致させるために、

`\tbaselineshift` を設定する役割を持っていた：

$$\text{\tbaselineshift} \leftarrow \frac{(h_M + d_M) - (h_a + d_a)}{2} + d_a - d_M,$$

ここで、 h_a, d_a はそれぞれ「 a 」の高さ・深さを表す。

現在の LuaTeX-ja は縦書きをサポートしていないので、この `\adjustbaseline` はほとんど何もしていない。

`\fontfamily{<family>}`

元々の L^AT_EX 2_ε におけるものと同様に、このコマンドは現在のフォントファミリー（欧文、和文、もしくは両方）を `<family>` に変更する。どのファミリーが変更されるかは以下のようにして決定される：

- 現在の和文フォントに対するエンコーディングが `<ja-enc>` であるとしよう。現在の和文フォントファミリーは、以下の 2 つの条件のうちの 1 つが満たされているときに `<family>` に変更される：
 - エンコーディング `<ja-enc>` におけるファミリー `<family>` が既に `\DeclareKanjiFamily` によって定義されている。
 - フォント定義ファイル `<ja-enc><family>.fd`（ファイル名は全て小文字）が存在する。
- 現在の欧文フォントに対するエンコーディングを `<al-enc>` とする。欧文フォントファミリーに対しても、上記の基準が用いられる。
- 上記のいずれれもが適用されない、つまり `<family>` が `<ja-enc>` と `<al-enc>` のどちらでも定義されないような場合がある。この場合、代替フォントに用いられるデフォルトのフォントファミリーが欧文フォントと和文フォントに用いられる。L^AT_EX のオリジナルの実装とは異なり、現在のエンコーディングは `<family>` には設定されないことに注意する。

`\DeclareAlternateKanjiFont{<base-encoding>}{<base-family>}{<base-series>}{<base-shape>}{<alt-encoding>}{<alt-family>}{<alt-series>}{<alt-shape>}{<range>}`

8.3 節の `\ltjdeclarealtfont` と同様に、前半の 4 引数の和文フォント（基底フォント）のうち `<range>` 中の文字を第 5– 第 8 引数の和文フォントを使って組むように指示する。使用例を図 3 に載せた。

- `\ltjdeclarealtfont` では基底フォント・置き換え先和文フォントはあらかじめ定義されていないといけない（その代わり即時発効）であったが、`\DeclareAlternateKanjiFont` の設定が実際に効力が発揮するのは、書体変更やサイズ変更を行った時、あるいは（これらを含むが）`\selectfont` が実行された時である。
- 段落や `hbox` の最後での設定値が段落 / `hbox` 全体にわたって通用する点や、`<range>` に負数 $-n$ を指定した場合、それが「基底フォントの文字クラス n に属する文字全体」と解釈されるのは `\ltjdeclarealtfont` と同じである。

この節の終わりに、`\SetRelationFont` と `\userelfont` の例を紹介しておこう。`\userelfont` の使用によって、「abc」の部分のフォントが Avant Garde (OT1/pag/m/n) に変わっていることがわかる。

```
1 \makeatletter
2 \SetRelationFont{JY3}{\k@family}{m}{n}{OT1}{pag}{m}{n}
3 % \k@family: current Japanese font family
4 \userelfont\selectfont あいうabc
```

あいう abc

10 拡張

LuaTeX-ja には（動作には必須ではないが）自由に読み込める拡張が付属している。これらは L^AT_EX のパッケージとして制作しているが、[luatexja-otf](#) と [luatexja-adjust](#) については plain LuaTeX でも `\input` で読み込み可能である。

10.1 luatexja-fontspec.sty

3.2 節で述べたように、この追加パッケージは [fontspec](#) パッケージで定義されているコマンドに対応する和文フォント用のコマンドを提供する。[fontspec](#) パッケージで指定可能な各種 font feature に加えて、和文版のコマンドには以下の“font feature”を指定することができる：

`CID=<name>`

`JFM=<name>`

`JFM-var=<name>`

これら 3 つのキーはそれぞれ `\jfont` に対する `cid`, `jfm`, `jfmvar` キーとそれぞれ対応する。

`CID` は下の `NoEmbed` と合わせて用いられたときのみ有効である。`\jfont` プリミティブに対する `cid`, `jfm`, `jfmvar` キーの詳細は 6.1 節と 6.2 節を参照。

`NoEmbed` これを指定することで、PDF に埋め込まれない「名前だけ」のフォントを指定することができる。6.2 節を参照。

なお、[luatexja-fontspec](#) 読み込み時には和文フォント定義ファイル `<ja-enc><family>.fd` は全く参照されなくなる。

10.2 luatexja-otf.sty

この追加パッケージは Adobe-Japan1（フォント自身が持っていれば、別の CID 文字セットでも可）の文字の出力をサポートする。[luatexja-otf](#) は以下の 2 つの低レベルコマンドを提供する：

`\CID{<number>}` CID 番号が `<number>` の文字を出力する。

`\UTF{<hex_number>}` 文字コードが（16 進で）`<hex_number>` の文字を出力する。このコマンドは `\char"<hex_number>` と似ているが、下の記述に注意すること。

このパッケージは、`ajmacros.sty` (`otf` パッケージ付属のマクロ集, 井上浩一氏作) から漢字コードを UTF8 にしたり、plain LuaTeX でも利用可能するという修正を加えた `luatexja-ajmacros.sty` も自動的に読み込む。そのため、マクロ集にある `\aj` 半角などのマクロもそのまま使用可能である。

■注意 `\CID` と `\UTF` コマンドによって出力される文字は以下の点で通常の文字と異なる：

- 常に **J**Achar として扱われる。
- OpenType feature（例えばグリフ置換やカーニング）をサポートするための [luaotfload](#) パッケージのコードはこれらの文字には働かない。

■JFM への記法の追加 [luatexja-otf](#) パッケージを読み込むと、JFM の `chars` テーブルのエントリとして 'AJ1-xxx' の形の文字列が使えるようになる。これは Adobe-Japan1 における CID 番号が xxx の文字を表す。

この拡張記法は、標準 JFM `jfm-ujis.lua` で、半角ひらがなのグリフ（CID 516-598）を正しく半角幅で組むために利用されている。

■IVS サポート 最近の OpenType フォントや TrueType フォントには、U+E0100–U+E01EF の範囲の「文字」（漢字用異体字セレクタ）を後置することによって字形を指定する仕組み（IVS）が含まれている。執筆時点の 2013 年 12 月では、`luaotfload` や `fontspec` パッケージ類は IVS に対応してはいないようである。これらのパッケージで対応してくれるのが理想的だが、それまでのつなぎとして、`luatexja-otf` パッケージ内に IVS 対応を仕込んでおいた。

IVS 対応は試験的なものである。有効にするには、`luatexja-otf` パッケージを読み込んだ上で以下の命令を実行する*8：

```
\directlua{luatexja.otf.enable_ivs()}
```

すると、上の命令を実行した箇所以降では、以下のように IVS による字形指定が有効となる。

```
1 \Large
2 \jfontspec{KozMinPr6N-Regular}
3 奈良県葛0E00城市と、東京都葛0E00飾区。 \
4 こんにちは、渡
5 邊0E00邊0E01邊0E02邊0E03邊0E04
6 邊0E05邊0E06邊0E07邊0E08邊0E09
7 邊0E0A邊0E0B邊0E0C邊0E0D邊0E0E
8 さん。
```

奈良県葛城市と、東京都葛飾区。
こんにちは、渡邊邊邊邊邊邊邊邊邊邊邊邊邊邊邊邊邊さん。

左上側の入力においては、漢字用異体字セレクタを明示するため、例えば Variation Selector 18 (U+E0101) を^{0E01}のように表記している。

また、IVS による字形指定は、font feature によるそれに優先されることとした。下の例において、jp78, jp90 指定で字形が変化した文字は異体字セレクタが続いていない「葛西」中の「葛」のみである。

```
1 \def\TEST#1{%
2   {\jfontspec[#1]{KozMinPr6N-Regular}}% 指定なし：葛城市，葛飾区，葛西
3   葛0E00城市，葛0E00飾区，葛西} \
4 指定なし：\TEST{ } jp78：葛城市，葛飾区，葛西
5 \texttt{jp78}：\TEST{CJKShape=JIS1978} jp90：葛城市，葛飾区，葛西
6 \texttt{jp90}：\TEST{CJKShape=JIS1990}
```

現状では、 \TeX 側のインターフェースとなる `luatexja-otf.sty` は一切変更していないので、ZR さんによる `PXipamjm` パッケージ*9にあるような気の利いた命令はまだない。異体字の一覧表示を行いたい場合は、git リポジトリ内の `test/test19-ivs.tex` 中にある Lua・ \TeX コードが参考になるだろう。

10.3 luatexja-adjust.sty

$\text{p}\TeX$ では、行長調整において優先度の概念が存在しなかったため、図 4 上段における半角分の半端は、図 4 中段のように、鍵括弧周辺の空白と和文間空白 (`kanjiskip`) の両方によって負担される。しかし、「日本語組版処理の要件」[5] や JIS X 4051 [7] においては、このような状況では半端は鍵括弧周辺の空白のみで負担し、その他の和文文字はベタ組で組まれる (図 4 下段) ことになっている。この追加パッケージは [5] や [7] における規定のような、優先順位付きの行長調整を提供する。詳細な

*8 この命令を 2 回以上実行しても意味がない。

*9 <https://github.com/zr-tex8r/PXipamjm>。説明は彼のブログ記事「pxipamjm パッケージの説明書のような何か (<http://d.hatena.ne.jp/zrbabbler/20131221>)」にある。

no adjustment	以上の原理は、「包除原理」とよく呼ばれるが
without priority	以上の原理は、「包除原理」とよく呼ばれるが
with priority	以上の原理は、「包除原理」とよく呼ばれるが

Note: the value of `kanjiskip` is $0\text{pt}^{+1/5}_{-1/5}\text{em}$ in this figure, for making the difference obvious.

図 4. 行長調整

仕様については 15 章を参照してほしい。

- 優先度付き行長調整は、段落を行分割した後に個々の行について行われるものである。そのため、行分割の位置は変化することはない。
また、`\hbox to ... {...}` のような「幅が指定された hbox」では無効である。
- 優先度付き行長調整を行うと、和文処理グルーの自然長は変化しないが、伸び量や縮み量は一般に変化する。そのため、既に組まれた段落を `\unhbox` などを利用して組み直す処理を行う場合には注意が必要である。

`luatexja-adjust` は、以下の命令を提供する。これらはすべてグローバルに効力を発揮する。

`\ltjdisableadjust` 優先順位付きの行長調整を無効化する。

`\ltjenableadjust` 優先順位付きの行長調整を有効化する。

`adjust=<bool> \ltjsetparameter` で指定可能な追加パラメータであり、`<bool>` が `true` なら `\ltjenableadjust` を、そうでなければ `\ltjdisableadjust` を実行する。

第 III 部 実装

11 パラメータの保持

11.1 LuaTeX-ja で用いられるレジスタと whatsit ノード

以下は LuaTeX-ja で用いられる寸法レジスタ (dimension)、属性レジスタ (attribute) のリストである。

`\jQ` (dimension) `\jQ` は写植で用いられた $1Q = 0.25\text{mm}$ (「級」とも書かれる) に等しい。したがって、この寸法レジスタの値を変更してはならない。

`\jH` (dimension) 同じく写植で用いられていた単位として「齒」があり、これも 0.25mm と等しい。この `\jH` は `\jQ` と同じ寸法レジスタを指す。

`\ltj@zw` (dimension) 現在の和文フォントの「全角幅」を保持する一時レジスタ。`\zw` 命令は、このレジスタを適切な値に設定した後、「このレジスタ自体を返す」。

`\ltj@zh` (dimension) 現在の和文フォントの「全角高さ」(通常、高さ と 深さの和) を保持する一時レジスタ。`\zh` 命令は、このレジスタを適切な値に設定した後、「このレジスタ自体を返す」。

`\jfam` (attribute) 数式用の和文フォントファミリの現在の番号。

`\ltj@curjfnt` (attribute) 現在の和文フォントのフォント番号。

`\ltj@charclass` (attribute) 和文文字の `glyph_node` の文字クラス。

`\ltj@yablsift` (attribute) スケールド・ポイント (2^{-16}pt) を単位とした欧文フォントのベースラインの移動量。

`\ltj@ykbllshift` (attribute) スケールド・ポイント (2^{-16} pt) を単位とした和文フォントのベースラインの移動量.

`\ltj@autospc` (attribute) そのノードで `kanjiskip` の自動挿入が許されるかどうか.

`\ltj@autoxspc` (attribute) そのノードで `xkanjiskip` の自動挿入が許されるかどうか.

`\ltj@icflag` (attribute) ノードの「種類」を区別するための属性. 以下のうちのひとつが値として割り当てられる:

italic (1) イタリック補正 (`\/`) によるグルー. このグルーの由来の区別 (`\kern` か `\/` か) は `xkanjiskip` の挿入過程において必要になる.

packed (2)

kinsoku (3) 禁則処理のために挿入されたペナルティ.

(from_jfm - 2)–(from_jfm + 2) (4–8) JFM 由来のグルー／カーン.

kanji_skip (9), ***kanji_skip_jfm*** (10) 和文間空白 `kanjiskip` を表すグルー.

xkanji_skip (11), ***xkanji_skip_jfm*** (12) 和欧文間空白 `xkanjiskip` を表すグルー.

processed (13) LuaTeX-ja の内部処理によって既に処理されたノード.

ic_processed (14) イタリック補正に由来するグルーであって、既に `JAg glue` 挿入処理にかかったもの.

boxbdd (15) `hbox` か段落の最初か最後に挿入されたグルー／カーン.

また、挿入処理の結果であるリストの最初のノードでは、`\ltj@icflag` の値に `processed_begin_flag` (128) が追加される. これによって、`\unhbox` が連続した場合でも「ボックスの境界」が識別できるようになっている.

`\ltj@kcati` (attribute) i は 7 より小さい自然数. これら 7 つの属性レジスタは、どの文字ブロックが `J Achar` のブロックとして扱われるかを示すビットベクトルを格納する.

さらに、LuaTeX-ja はいくつかの user-defined whatsit node を内部処理に用いる. これらの whatsit ノードの `type` は 100 であり、ノードは自然数を格納している. user-defined whatsit を識別するための `user_id` は `luatexbase.newuserwhatsitid` により確保されており、下の見出しは単なる識別用でしかない.

inhibitglue `\inhibitglue` が指定されたことを示すノード. これらのノードの `value` フィールドは意味を持たない.

stack_marker LuaTeX-ja のスタックシステム (次の節を参照) のためのノード. これらのノードの `value` フィールドは現在のグルーネストレベルを表す.

char_by_cid `luaotfload` のコールバックによる処理が適用されない `J Achar` のためのノードで、`value` フィールドに文字コードが格納されている. この種類のノードはそれぞれが `luaotfload` のコールバックの処理の後で `glyph_node` に変換される. `\CID`, `\UTF` や `IVS` 対応処理でこの種類のノードが利用されている.

replace_vs 上の `char_by_cid` と同様に、これらのノードは `luaotfload` のコールバックによる処理が適用されない `ALchar` のためのものである.

begin_par 「段落の開始」を意味するノード. `list` 環境, `itemize` 環境などにおいて、`\item` で始まる各項目は……

これらの whatsit ノードは `J A glue` の挿入処理の間に取り除かれる.

11.2 LuaTeX-ja のスタックシステム

■背景 LuaTeX-ja は独自のスタックシステムを持ち、LuaTeX-ja のほとんどのパラメータはこれを用いて保持されている。その理由を明らかにするために、`kanjiskip` パラメータがスキップレジスタで保持されているとし、以下のコードを考えてみよう：

```
1 \ltjsetparameter{kanjiskip=0pt}ふがふが.%
2 \setbox0=\hbox{%
3   \ltjsetparameter{kanjiskip=5pt}ほげほげ}   ふがふが.ほげほげ.びよびよ
4 \box0.びよびよ\par
```

7.1 節で述べたように、ある `hbox` の中で効力を持つ `kanjiskip` の値は最後に現れた値のみであり、したがってボックス全体に適用される `kanjiskip` は 5pt であるべきである。しかし、LuaTeX の実装を観察すると、この 5pt という長さは何のコールバックからも知ることはできないことがわかる。LuaTeX のソースファイルの 1 つ `tex/packaging.w` の中に、以下のコードがある：

```
1226 void package(int c)
1227 {
1228     scaled h;          /* height of box */
1229     halfword p;        /* first node in a box */
1230     scaled d;          /* max depth */
1231     int grp;
1232     grp = cur_group;
1233     d = box_max_depth;
1234     unsave();
1235     save_ptr -= 4;
1236     if (cur_list.mode_field == -hmode) {
1237         cur_box = filtered_hpack(cur_list.head_field,
1238                                 cur_list.tail_field, saved_value(1),
1239                                 saved_level(1), grp, saved_level(2));
1240         subtype(cur_box) = HLIST_SUBTYPE_HBOX;
```

`unsave()` が `filtered_hpack()` (これは `hpack_filter` コールバックが実行される場所である) の前に実行されていることに注意する。したがって、上記ソース中で 5pt は `unsave()` のところで捨てられ、`hpack_filter` コールバックからはアクセスすることができない。

■解決法 スタックシステムのコードは Dev-luatex メーリングリストのある投稿^{*10}をベースにしている。

情報を保持するために、2つの TeX の整数レジスタを用いている：`\ltj@@stack` にスタックレベル、`\ltj@@group@level` に最後の代入がなされた時点での TeX のグループレベルを保持している。パラメータは `charprop_stack_table` という名前のひとつの大きなテーブルに格納される。ここで、`charprop_stack_table[i]` はスタックレベル i のデータを格納している。もし新しいスタックレベルが `\ltjsetparameter` によって生成されたら、前のレベルの全てのデータがコピーされる。

上の「背景」で述べた問題を解決するために、LuaTeX-ja では次の手法を用いる：スタックレベルが増加するするとき、`type`、`subtype`、`value` がそれぞれ 44 (`user_defined`)、`stack_marker`、そして現在のグループレベルである `whatsit` ノードを現在のリストに付け加える (このノードを `stack_flag` とする)。これにより、ある `hbox` の中で代入がなされたかどうかを知ることが可能となる。スタックレベルを s 、その `hbox` group の直後の TeX のグループレベルを t とすると：

^{*10} [Dev-luatex] `tex.currentgrouplevel`: Jonathan Sauer による 2008/8/19 の投稿。

```

380 \protected\def\ltj@setpar@global{%
381   \relax\ifnum\globaldefs>0\directlua{luatexja.isglobal='global'}%
382   \else\directlua{luatexja.isglobal=''}\fi
383 }
384 \protected\def\ltjsetparameter#1{%
385   \ltj@setpar@global\setkeys[ltj]{japaram}{#1}\ignorespaces}
386 \protected\def\ltjglobalsetparameter#1{%
387   \relax\ifnum\globaldefs<0\directlua{luatexja.isglobal=''}%
388   \else\directlua{luatexja.isglobal='global'}\fi%
389   \setkeys[ltj]{japaram}{#1}\ignorespaces}

```

図 5. パラメータ設定命令の定義

- もしその hbox の中身を表すリストの中に *stack_flag* ノードがなければ、hbox の中では代入は起こらなかったということになる。したがって、その hbox の終わりにおけるパラメータの値はスタックレベル *s* に格納されている。
- もし値が $t + 1$ の *stack_flag* ノードがあれば、その hbox の中で代入が起こったことになる。したがって、hbox の終わりにおけるパラメータの値はスタックレベル $s + 1$ に格納されている。
- もし *stack_flag* ノードがあるがそれらの値が全て $t + 1$ より大きい場合、そのボックスの中で代入が起こったが、それは「より内部の」グループで起こったということになる。したがって、hbox の終わりでのパラメータの値はスタックレベル *s* に格納されている。

このトリックを正しく働かせるためには、`\ltj@@stack` と `\ltj@@group@level` への代入は `\globaldefs` の値によらず常にローカルでなければならぬことに注意する。この問題は `\directlua{tex.globaldefs=0}` (この代入は常にローカル) を用いることで解決している。

11.3 スタックシステムで使用される関数

本節では、ユーザが LuaTeX-ja のスタックシステムを使用して、TeX のグルーピングに従うような独自のデータを取り扱う方法を述べる。

スタックに値を設定するには、以下の Lua 関数を呼びだせば良い：

```
luatexja.stack.set_stack_table(<any> index, <any> data)
```

直感的には、スタックテーブル中のインデックス *index* の値を *data* にする、という意味である。*index* の値としては `nil` と `NaN` 以外の任意の値を使えるが、自然数は LuaTeX-ja が使用する (将来の拡張用も含む) ので、ユーザが使用する場合は負の整数値か文字列の値にすることが望ましい。また、ローカルに設定されるかグローバルに設定されるかは、`luatexja.isglobal` の値に依存する (グローバルに設定されるのは、`luatexja.isglobal == 'global'` であるちょうどその時)。

スタックの値は、

```
luatexja.stack.get_stack_table(<any> index, <any> default, <number> level)
```

の戻り値で取得できる。*level* はスタックレベルであり、通常は `\ltj@@stack` の値を指定することになるだろう。*default* はレベル *level* のスタックに値が設定されていなかった場合に返すデフォルト値である。

11.4 パラメータの拡張

ここでは、`luatexja-adjust` で行なっているように、`\ltjsetparameter`、`\ltjgetparameter` に指定可能なキーを追加する方法を述べる。

■**パラメータの設定** `\ltjsetparameter` と、`\ltjglobalsetparameter` の定義は図 5 のようになっている。本質的なのは最後の `\setkeys` で、これは `xkeyval` パッケージの提供する命令である。

このため、`\ltjsetparameter` に指定可能なパラメータを追加するには、 $\langle prefix \rangle$ を `ltj`、 $\langle family \rangle$ を `japaram` としたキーを

```
\define@key[ltj]{japaram}{...}{...}
```

のように定義すれば良いだけである。なお、パラメータ指定がグローバルかローカルかどうかを示す `luatexja.isglobal` が、

$$\text{luatexja.isglobal} = \begin{cases} \text{'global'} & \text{パラメータ設定はグローバル} \\ \text{''} & \text{パラメータ設定はローカル} \end{cases} \quad (1)$$

として自動的にセットされる*11。

■**パラメータの取得** 一方、`\ltjgetparameter` は Lua スクリプトによって実装されている。値を取得するのに追加引数の要らないパラメータについては、`luatexja.unary_pars` 内に処理内容を記述した関数を定義すれば良い。例えば、Lua スクリプトで

```
1 function luatexja.unary_pars.hoge (t)
2   return 42
3 end
```

を実行すると、`\ltjgetparameter{hoge}` は 42 という文字列を返す。関数 `luatexja.unary_pars.hoge` の引数 t は、11.2 節で述べた LuaTeX-jā のスタックシステムにおけるスタックレベルである。戻り値はいかなる値であっても、最終的には文字列として出力されることに注意。

一方、追加引数（数値しか許容しない）が必要なパラメータについては、まず Lua スクリプトで処理内容の本体を記述しておく：

```
1 function luatexja.binary_pars.fuga (c, t)
2   return tostring(c) .. ', ' .. tostring(42)
3 end
```

引数 t は、先に述べた通りのスタックレベルである。一方、引数 c は `\ltjgetparameter` の第 2 引数を表す数値である。しかしこれだけでは駄目で、

```
\ltj@@decl@array@param{fuga}
```

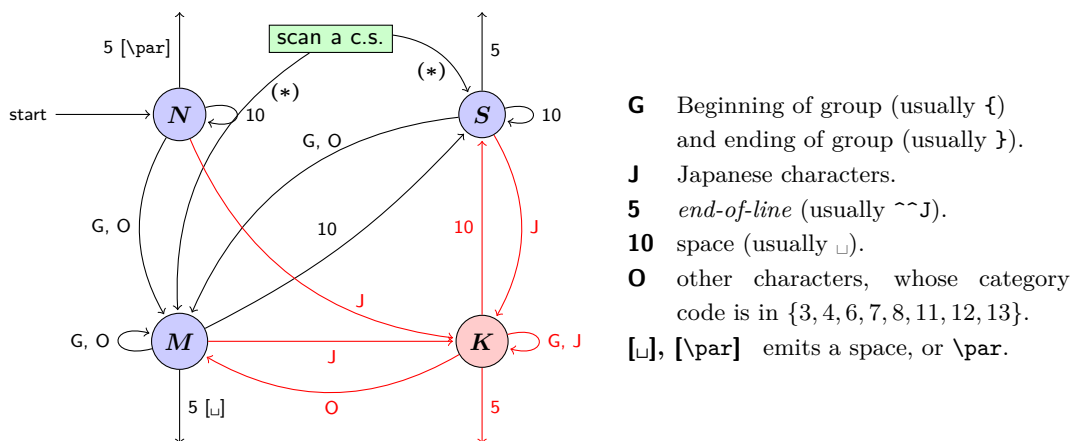
を実行し、TeX インターフェース側に「`\ltjgetparameter{fuga}` は追加引数が必要」ということを通知する必要がある。

12 和文文字直後の改行

12.1 参考：pTeX の動作

欧文では文章の改行は単語間でしか行わない。そのため、TeX では、（文字の直後の）改行は空白文字と同じ扱いとして扱われる。一方、和文ではほとんどどこでも改行が可能なため、pTeX では和

*11 命令が `\ltjglobalsetparameter` かどうかだけではなく、実行時の `\globaldefs` の値にも依存して定まる。



- We omitted about category codes 9 (*ignored*), 14 (*comment*), and 15 (*invalid*) from the above diagram. We also ignored the input like “`^^A`” or “`^^df`”.
- When a character whose category code is 0 (*escape character*) is seen by T_EX, the input processor scans a control sequence (*scan a c.s.*). These paths are not shown in the above diagram. After that, the state is changed to State S (skipping blanks) in most cases, but to State M (middle of line) sometimes.

図 6. pT_EX の入力処理部の状態遷移

文文字の直後の改行は単純に無視されるようになっている。

このような動作は、pT_EX が T_EX からエンジンとして拡張されたことによって可能になったことである。pT_EX の入力処理部は、T_EX におけるそれと同じように、有限オートマトンとして記述することができ、以下に述べるような 4 状態を持っている。

- State N: 行の開始。
- State S: 空白読み飛ばし。
- State M: 行中。
- State K: 行中 (和文文字の後)。

また、状態遷移は、図 6 のようになっており、図中の数字はカテゴリーコードを表している。最初の 3 状態は T_EX の入力処理部と同じであり、図中から状態 K と「j」と書かれた矢印を取り除けば、T_EX の入力処理部と同じものになる。

この図から分かることは、

行が和文文字 (とグループ境界文字) で終わっていれば、改行は無視される

ということである。

12.2 LuaT_EX-ja の動作

LuaT_EX の入力処理部は T_EX のそれと全く同じであり、コールバックによりユーザがカスタマイズすることはできない。このため、改行抑制の目的でユーザが利用できそうなコールバックとしては、`process_input_buffer` や `token_filter` に限られてしまう。しかし、T_EX の入力処理部をよく見ると、後者も役には経たないことが分かる：改行文字は、入力処理部によってトークン化される時に、カテゴリーコード 10 の 32 番文字へと置き換えられてしまうため、`token_filter` で非標準なトークン読み出しを行おうとしても、空白文字由来のトークンと、改行文字由来のトークンは区別できないのだ。

すると、我々のとれる道は、`process_input_buffer` を用いて Lua_{TeX} の入力処理部に引き渡される前に入力文字列を編集するというものしかない。以上を踏まえ、Lua_{TeX}-ja における「和文文字直後の改行抑制」の処理は、次のようになっている：

各入力行に対し、その入力行が読まれる前の内部状態で以下の 3 条件が満たされている場合、Lua_{TeX}-ja は U+FFFFF の文字^{*12}を末尾に追加する。よって、その場合に改行は空白とは見做されないこととなる。

1. `\endlinechar` の文字^{*13}のカテゴリーコードが 5 (*end-of-line*) である。
2. U+FFFFF のカテゴリーコードが 14 (*comment*) である。
3. 入力行は次の「正規表現」にマッチしている：

$$(\text{any char})^*(\mathbf{JA}\text{char})({\text{catcode}} = 1} \cup {\text{catcode}} = 2})^*$$

この仕様は、前節で述べた p_{TeX} の仕様にできるだけ近づけたものとなっている。条件 1. は、`verbatim` 系環境などの日本語対応マクロを書かなくてすませるためのものである。

しかしながら、p_{TeX} と完全に同じ挙動が実現できたわけではない。次のように、和文文字の範囲を変更したちょうどその行においては挙動が異なる：

```

1 \fontspec[Ligatures=TeX]{TeX Gyre Termes}
2 \ltjsetparameter{autoxspacing=false}
3 \ltjsetparameter{jacharrange={-6}}xあ          xyzい u
4 y\ltjsetparameter{jacharrange={+6}}zい
5 u

```

上ソース中の「あ」は欧文文字扱いであり、ここで使用している欧文フォント TeX Gyre Termes は「あ」を含まない。よって、出力に「あ」は現れないことは不思議ではない。それでも、p_{TeX} とまったく同じ挙動を示すならば、出力は「x yzいu」となるはずである。しかし、実際には上のように異なる挙動となっているが、それは以下の理由による：

- 3 行目を `process_input_buffer` で処理する時点では、「あ」は和文文字扱いである。よって 3 行目は和文文字で終わることになり、コメント文字 U+FFFFF が追加される。よって、直後の改行文字は無視されることになり、空白は入らない。
- 4 行目を `process_input_buffer` で処理する時点では、「い」は欧文文字扱いである。よって 4 行目は欧文文字で終わることになり、直後の改行文字は空白に置き換わる。

このため、トラブルを避けるために、和文文字の範囲を `\ltjsetparameter` で編集した場合、その行はそこで改行するようにした方がいいだろう。

13 JFM グルーの挿入, `kanjiskip` と `xkanjiskip`

13.1 概要

Lua_{TeX}-ja における `JAg glue` の挿入方法は、p_{TeX} のそれとは全く異なる。p_{TeX} では次のような仕様であった：

- JFM グルーの挿入は、和文文字を表すトークンを元に水平リストに（文字を表す）`<char_node>` を追加する過程で行われる。
- `xkanjiskip` の挿入は、`hbox` へのパッケージングや行分割前に行われる。

^{*12} この文字はコメント文字として扱われるように Lua_{TeX}-ja 内部で設定をしている。

^{*13} 普通は、改行文字（文字コード 13 番）である。

- `kanjiskip` はノードとしては挿入されない。パッケージングや行分割の計算時に「和文文字を表す 2 つの `\char_node` の間には `kanjiskip` がある」ものとみなされる。

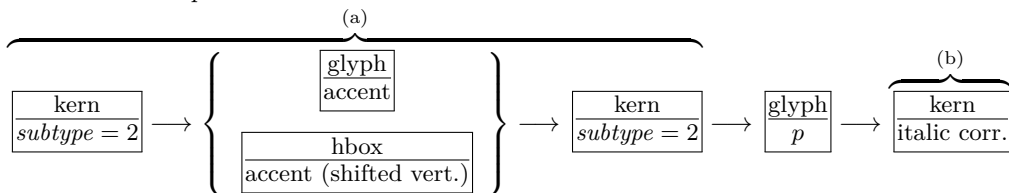
しかし、`LuaTeX-ja` では、`hbox` へのパッケージングや行分割前に全ての **JAglue**、即ち JFM グルー・`xkanjiskip`・`kanjiskip` の 3 種類を一度に挿入することになっている。これは、`LuaTeX` において欧文の合字・カーニング処理がノードベースになったことに対応する変更である。

`LuaTeX-ja` における **JAglue** 挿入処理では、次節で定義する「クラスタ」を単位にして行われる。大雑把にいうと、「クラスタ」は文字とそれに付随するノード達（アクセント位置補正用のカーンや、イタリック補正）をまとめたものであり、2 つのクラスタの間には、ペナルティ、`\adjust`、`whatsit` など、行組版には関係しないものがある。

13.2 「クラスタ」の定義

定義 1. クラスタは以下の形のうちのどれかひとつをとる連続的なノードのリストである：

1. その `\tj@icflag` の値が $[3, 15)$ に入るノードのリスト。これらのノードはある既にパッケージングされた `hbox` から `\unhbox` でアンパックされたものである。その `id` は `id_pbox` である。
2. インライン数式でその境界に 2 つの `math_node` を含むもの。その `id` は `id_math` である。
3. `glyph_node p` とそれに関するノード：
 - (1) `p` のイタリック補正のためのカーン。
 - (2) `\accent` による `p` に付随したアクセント。



`id` は `glyph_node` が和文文字を表すかどうかによって `id_jglyph`、もしくは `id_glyph` となる。

4. ボックス様のノード、つまり水平ボックス、垂直ボックス、罫線 (`\vrule`)、そして `unset_node`。その `id` は垂直に移動していない `hbox` ならば `id_hlist`、そうでなければ `id_box_like` となる。
5. グルー、`subtype` が 2 (`accent`) ではないカーン、そして任意改行。その `id` はそれぞれ `id_glue`、`id_kern`、そして `id_disc` である。

以下では `Np`、`Nq`、`Nr` でクラスタを表す。

■**id の意味** `Np.id` の意味を述べるとともに、「先頭の文字」を表す `glyph_node Np.head` と、「最後の文字」を表す `glyph_node Np.tail` を次のように定義する。直感的に言うとも、`Np` は `Np.head` で始まり `Np.tail` で終わるような単語、と見做すことができる。これら `Np.head`、`Np.tail` は説明用に準備した概念であって、実際の Lua コード中にそのように書かれているわけではないことに注意。

id_jglyph 和文文字。

`Np.head`、`Np.tail` は、その和文文字を表している `glyph_node` そのものである。

id_glyph 和文文字を表していない `glyph_node p`。

多くの場合、`p` は欧文文字を格納しているが、「缶」などの合字によって作られた `glyph_node` である可能性もある。前者の場合、`Np.head`、`Np.tail` = `p` である。一方、後者の場合、

- `Np.head` は、合字の構成要素の先頭 → (その `glyph_node` における) 合字の構成要素の先頭 → …… と再帰的に検索していったどり着いた `glyph_node` である。
- `Np.last` は、同様に末尾 → 末尾 → と検索していったどり着いた `glyph_node` である。

id.math インライン数式.

便宜的に, *Np.head*, *Np.tail* とともに「文字コード -1 の欧文文字」とおく.

id.hlist 縦方向にシフトされていない hbox.

この場合, *Np.head*, *Np.tail* はそれぞれ *p* の内容を表すリストの, 先頭・末尾のノードである.

- 状況によっては, TeX ソースで言うと
$$\hbox{\hbox{abc}...\hbox{\lower1pt\hbox{xyz}}}$$
のように, *p* の内容が別の hbox で開始・終了している可能性も十分あり得る. そのような場合, *Np.head*, *Np.tail* の算出は, 垂直方向にシフトされていない hbox の場合だけ内部を再帰的に探索する. 例えば上の例では, *Np.head* は文字「a」を表すノードであり, 一方 *Np.tail* は垂直方向にシフトされた hbox, $\lower1pt\hbox{xyz}$ に対応するノードである.
- また, 先頭にアクセント付きの文字がきたり, 末尾にイタリック補正用のカーンが来ることもあり得る. この場合は, クラスタの定義のところにもあったように, それらは無視して算出を行う.
- 最初・最後のノードが合字によって作られた *glyph_node* のときは, それぞれに対して *id.glyph* と同様に再帰的に構成要素をたどっていく.

id.pbox 「既に処理された」ノードのリストであり, これらのノードが二度処理を受けないためにまとめて1つのクラスタとして取り扱うだけである. *id.hlist* と同じ方法で *Np.head*, *Np.tail* を算出する,

id.disc discretionary break ($\discretionary\{pre\}\{post\}\{nobreak\}$).

id.hlist と同じ方法で *Np.head*, *Np.tail* を算出するが, 第3引数の *nobreak* (行分割が行われない時の内容) を使う. 言い換えれば, ここで行分割が発生した時の状況は全く考慮に入れない.

id.box_like *id.hlist* とならない box や, rule.

この場合は, *Np.head*, *Np.tail* のデータは利用されないので, 2つの算出は無意味である. 敢えて明示するならば, *Np.head*, *Np.tail* は共に nil 値である.

他 以上でない *id* に対しても, *Np.head*, *Np.tail* の算出は無意味.

■**クラスタの別の分類** さらに, JFM グルー挿入処理の実際の説明により便利なように, *id* とは別のクラスタの分類を行っておく. 挿入処理では2つの隣り合ったクラスタの間に空白等の実際の挿入を行うことは前に書いたが, ここでの説明では, 問題にしているクラスタ *Np* は「後ろ側」のクラスタであるとす. 「前側」のクラスタについては, 以下の説明で *head* が *last* に置き換わることに注意すること.

和文 A リスト中に直接出現している和文文字. *id* が *id.jglyph* であるか,

id が *id.pbox* であって *Np.head* が **J**Achar であるとき.

和文 B リスト中の hbox の中身の先頭として出現した和文文字. 和文 A との違いは, これの前に JFM グルーの挿入が行われない (*xkanjiskip*, *kanjiskip* は入り得る) ことである.

id が *id.hlist* か *id.disc* であって *Np.head* が **J**Achar であるとき.

欧文 リスト中に直接 / hbox の中身として出現している欧文文字. 次の3つの場合が該当:

- *id* が *id.glyph* である.
- *id* が *id.math* である.
- *id* が *id.pbox* か *id.hlist* か *id.disc* であって, *Np.head* が **A**Lchar.

箱 box, またはそれに類似するもの. 次の2つが該当:

- *id* が *id.pbox* か *id.hlist* か *id.disc* であって, *Np.head* が *glyph_node* でない.
- *id* が *id.box_like* である.

13.3 段落 / hbox の先頭や末尾

■**先頭部の処理** まず、段落 / hbox の一番最初にあるクラス N_p を探索する。hbox の場合は何の問題もないが、段落の場合では以下のノード達を事前に読み飛ばしておく：

- `\parindent` 由来の `hbox(subtype = 3)`
- `subtype` が 44 (`user_defined`) でないような `whatsit`

これは、`\parindent` 由来の `hbox` がクラスを構成しないようにするためである。

次に、 N_p の直前に空白 g を必要なら挿入する：

1. この処理が働くような N_p は**和文 A** である。
2. 問題のリストが字下げありの段落 (`\parindent` 由来の `hbox` あり) の場合は、この空白 g は「文字コード 'parb' の文字」と N_p の間に入るグルー / カーンである。
3. そうでないとき (`noindent` で開始された段落や `hbox`) は、 g は「文字コード 'boxb' の文字」と N_p の間に入るグルー / カーンである。

ただし、もし g が `glue` であった場合、この挿入によって N_p による行分割が新たに可能になるべきではない。そこで、以下の場合には、 g の直前に `\penalty10000` を挿入する：

- 問題にしているリストが段落であり、かつ
- N_p の前には予めペナルティがなく、 g は `glue`。

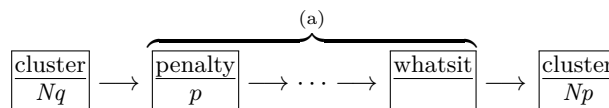
■**末尾の処理** 末尾の処理は、問題のリストが段落のものか `hbox` のものかによって異なる。後者の場合は容易い：最後のクラスを N_q とおくと、 N_q と「文字コード 'boxb' の文字」の間に入るグルー / カーンを、 N_q の直後に挿入するのみである。

一方、前者 (段落) の場合は、リストの末尾は常に `\penalty10000` と、`\parfillskip` 由来のグルーが存在する。段落の最後の「通常の和文文字 + 句点」が独立した行となるのを防ぐために、`jcharwidowpenalty` の値の分だけ適切な場所のペナルティを増やす。

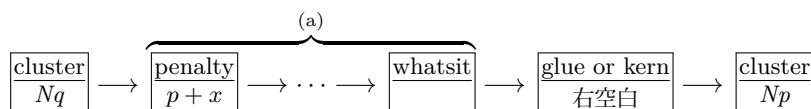
ペナルティ量を増やす場所は、`head` が **J**`A``char` であり、かつその文字の `kcatcode` が偶数であるような最後のクラスの直前にあるものたちである*14。

13.4 概観と典型例：2つの「和文 A」の場合

先に述べたように、2つの隣り合ったクラス N_q と N_p の間には、ペナルティ、`\adjust`、`whatsit` など、行組版には関係しないものがある。模式的に表すと、



のようになっている。間の (a) に相当する部分には、何のノードもない場合ももちろんあり得る。そうして、JFM グルー挿入後には、この2クラス間には次のようになる：



*14 大雑把に言えば、`kcatcode` が奇数であるような **J**`A``char` を約物として考えていることになる。`kcatcode` の最下位ビットはこの `jcharwidowpenalty` 用にも利用される。

以後、典型的な例として、クラス Nq と Np が共に和文 A である場合を見ていこう、この場合が全ての場合の基本となる。

■「右空白」の算出 まず、「右空白」にあたる量を算出する。通常はこれが、隣り合った2つの和文文字間に入る空白量となる。

JFM 由来 [M] JFM の文字クラス指定によって入る空白を以下によって求める。この段階で空白量が未定義（未指定）だった場合、デフォルト値 `kanjiskip` を採用することとなるので、次へ。

- もし両クラスの間で `\inhibitglue` が実行されていた場合（証として `whatsit` ノードが自動挿入される）、代わりに `kanjiskip` が挿入されることとなる。次へ。
- Nq と Np が同じ JFM・同じ `jfmvar` キー・同じサイズの和文フォントであったならば、共通に使っている JFM 内で挿入される空白（グルー／カーン）が決まっているか調べ、決まっていればそれを採用。
1. でも 2. でもない場合は、JFM・`jfmvar`・サイズの3つ組は Nq と Np で異なる。この場合、まず

$$gb := (Nq \text{ と「使用フォントが } Nq \text{ のそれと同じで、} \\ \text{文字コードが } Np \text{ のその文字} \text{」との間に入るグルー／カーン}) \\ ga := (\text{「使用フォントが } Np \text{ のそれと同じで、} \\ \text{文字コードが } Nq \text{ のその文字} \text{」と } Np \text{ との間に入るグルー／カーン})$$

として、前側の文字の JFM を使った時の空白（グルー／カーン）と、後側の文字の JFM を使った時のそれを求める。

gb, ga それぞれに対する $\langle ratio \rangle$ の値を d_b, d_a とする。

- ga と gb の両方が未定義であるならば、JFM 由来のグルーは挿入されず、`kanjiskip` を採用することとなる。どちらか片方のみが未定義であるならば、次のステップでその未定義の方は長さ0の kern で、 $\langle ratio \rangle$ の値は0であるかのように扱われる。
- `differentjfm` の値が `pleft`, `pright`, `paverage` のとき、 $\langle ratio \rangle$ の指定に従って比例配分を行う。JFM 由来のグルー／カーンは以下の値となる：

$$f\left(\frac{1-d_b}{2}gb + \frac{1+d_b}{2}ga, \frac{1-d_a}{2}gb + \frac{1+d_a}{2}ga\right)$$

ここで、 $f(x, y)$ は

$$f(x, y) = \begin{cases} x & \text{if } \text{differentjfm} = \text{pleft}; \\ y & \text{if } \text{differentjfm} = \text{pright}; \\ (x+y)/2 & \text{if } \text{differentjfm} = \text{paverage}; \end{cases}$$

- `differentjfm` がそれ以外の値の時は、 $\langle ratio \rangle$ の値は無視され、JFM 由来のグルー／カーンは以下の値となる：

$$f(gb, ga)$$

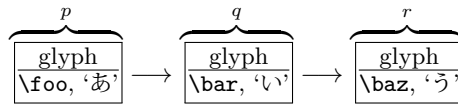
ここで、 $f(x, y)$ は

$$f(x, y) = \begin{cases} \min(x, y) & \text{if } \text{differentjfm} = \text{small}; \\ \max(x, y) & \text{if } \text{differentjfm} = \text{large}; \\ (x+y)/2 & \text{if } \text{differentjfm} = \text{average}; \\ x+y & \text{if } \text{differentjfm} = \text{both}; \end{cases}$$

例えば、

```
\jfont\foo=psft:Ryumin-Light:jfm=ujis;-kern
\jfont\bar=psft:GothicBBB-Medium:jfm=ujis;-kern
\jfont\baz=psft:GothicBBB-Medium:jfm=ujis;jfmvar=piyo;-kern
```

という 3 フォントを考え、



という 3 ノードを考える (それぞれ単独でクラスタをなす)。この場合、 p と q の間は、実フォントが異なるにもかかわらず 2. の状況となる一方で、 q と r の間は (実フォントが同じなのに) jfmvar キーの内容が異なるので 3. の状況となる。

kanjiskip [K] 上の [M] において空白が定まらなかった場合、以下で定めた量「右空白」として採用する。この段階においては、`\inhibitglue` は効力を持たないため、結果として、2 つの和文文字間には常に何らかのグルー／カーンが挿入されることとなる。

1. 両クラスタ (厳密には $Nq.tail$, $Np.head$) の中身の文字コードに対する `autospacing` パラメタが両方とも `false` だった場合は、長さ 0 の glue とする。
2. ユーザ側から見た `kanjiskip` パラメタの自然長が $\maxdimen = (2^{30} - 1)sp$ でなければ、`kanjiskip` パラメタの値を持つ glue を採用する。
3. 2. でない場合は、 Nq , Np で使われている JFM に指定されている `kanjiskip` の値を用いる。どちらか片方のクラスタだけが和文文字 (和文 A・和文 B) のときは、そちらのクラスタで使われている JFM 由来の値だけを用いる。もし両方で使われている JFM が異なった場合は、上の [M] 3. と同様の方法を用いて調整する。

■禁則用ペナルティの挿入 まず、

$a := (Nq^{*15}$ の文字に対する `postbreakpenalty` の値) + (Np^{*16} の文字に対する `prebreakpenalty` の値)

とおく。ペナルティは通常 $[-10000, 10000]$ の整数値をとり、また ± 10000 は正負の無限大を意味することになっているが、この a の算出では単純な整数の加減算を行う。

a は禁則処理用に Nq と Np の間に加えられるべきペナルティ量である。

P-normal [PN] Nq と Np の間の (a) 部分にペナルティ (`penalty_node`) があれば処理は簡単である：それらの各ノードにおいて、ペナルティ値を (± 10000 を無限大として扱いつつ) a だけ増加させればよい。また、 $10000 + (-10000) = 0$ としている。

少々困るのは、(a) 部分にペナルティが存在していない場合である。直感的に、補正すべき量 a が 0 でないとき、その値をもつ `penalty_node` を作って「右空白」の (もし未定義なら Np の) 直前に挿入……ということになるが、実際には僅かにこれより複雑である。

- 「右空白」がカーンであるとき、それは「 Nq と Np の間で改行は許されない」ことを意図している。そのため、この場合は $a \neq 0$ であってもペナルティの挿入はしない。
- そうでないときは、 $a \neq 0$ ならば `penalty_node` を作って挿入する。

13.5 その他の場合

本節の内容は表 9 にまとめてある。

■和文 A と欧文の間 Nq が和文 A で、 Np が欧文の場合、JFM グルー挿入処理は次のようにして行われる。

- 「右空白」については、まず以下に述べる Boundary-B [O_B] により空白を決定しようと試みる。それが失敗した場合は、`xkanjiskip [X]` によって定める。

*16 厳密にはそれぞれ $Nq.tail$, $Np.head$.

表 9. JFM グルーの概要

$Np \downarrow$	和文 A	和文 B	欧文	箱	glue	kern
和文 A	$\frac{M \rightarrow K}{PN}$	$\frac{O_A \rightarrow K}{PN}$	$\frac{O_A \rightarrow X}{PN}$	$\frac{O_A}{PA}$	$\frac{O_A}{PN}$	$\frac{O_A}{PS}$
和文 B	$\frac{O_B \rightarrow K}{PA}$	$\frac{K}{PS}$	$\frac{X}{PS}$			
欧文	$\frac{O_B \rightarrow X}{PA}$	$\frac{X}{PS}$				
箱	$\frac{O_B}{PA}$					
glue	$\frac{O_B}{PN}$					
kern	$\frac{O_B}{PS}$					

上の表において、 $\frac{M \rightarrow K}{PN}$ は次の意味である：

1. 「右空白」を決めるために、LuaTeX-já はまず「JFM 由来 [M]」の方法を試みる。これが失敗したら、LuaTeX-já は「kanjiskip [K]」の方法を試みる。
2. LuaTeX-já は 2 つのクラスタの間の禁則処理用のペナルティを設定するために「P-normal [PN]」の方法を採用する。

- 禁則用ペナルティも、以前述べた P-normal [PN] と同じである。

Boundary-B [O_B] 和文文字と「和文でないもの」との間に入る空白を以下によって求め、未定義でなければそれを「右空白」として採用する。JFM-origin [M] の変種と考えて良い。これによって定まる空白の典型例は、和文の閉じ括弧と欧文文字の間に入る半角アキである。

1. もし両クラスタの間で `\inhibitglue` が実行されていた場合（証として `whatsit` ノードが自動挿入される）、「右空白」は未定義。
2. そうでなければ、 Nq と「文字コードが `'jcharbdd'` の文字」との間に入るグルー／カーンとして定まる。

xkanjiskip [X] この段階では、kanjiskip [K] のときと同じように、以下で定めた量を「右空白」として採用する。`\inhibitglue` は効力を持たない。

1. 以下のいずれかの場合は、`xkanjiskip` の挿入は抑止される。しかし、実際には行分割を許容するために、長さ 0 の glue を採用する：
 - 両クラスタにおいて、それらの中身の文字コードに対する `autoxspacing` パラメタが共に `false` である。
 - Nq の中身の文字コードについて、「直後への `xkanjiskip` の挿入」が禁止されている（つまり、`jaxspmode` (or `alxspmode`) パラメタが 2 以上）。
 - Np の中身の文字コードについて、「直前への `xkanjiskip` の挿入」が禁止されている（つまり、`jaxspmode` (or `alxspmode`) パラメタが偶数）。
2. ユーザ側から見た `xkanjiskip` パラメタの自然長が $\backslash\maxdimen = (2^{30} - 1)\text{sp}$ でなければ、`xkanjiskip` パラメタの値を持つ glue を採用する。
3. 2. でない場合は、 Nq , Np (和文 A/和文 B なのは片方だけ) で使われている JFM に指定されている `xkanjiskip` の値を用いる。

■**欧文と和文 A の間** N_q が欧文で、 N_p が和文 A の場合、JFM グルー挿入処理は上の場合とほぼ同じである。和文 A のクラスタが逆になるので、Boundary-A [O_A] の部分が変わるだけ。

- 「右空白」については、まず以下に述べる Boundary-A [O_A] により空白を決定しようと試みる。それが失敗した場合は、`xkanjiskip` [X] によって定める。
- 禁則用ペナルティは、以前述べた P-normal [PN] と同じである。

Boundary-A [O_A] 「和文でないもの」と和文文字との間に入る空白を以下によって求め、未定義でなければそれを「右空白」として採用する。JFM-origin [M] の変種と考えて良い。これによって定まる空白の典型例は、欧文文字と和文の開き括弧との間に入る半角アキである。

1. もし両クラスタの間で `\inhibitglue` が実行されていた場合（証として `whatsit` ノードが自動挿入される）、次へ。
2. そうでなければ、「文字コードが `'jcharbdd'` の文字」と N_p との間に入るグルー／カーンとして定まる。

■**和文 A と箱・グルー・カーンの間** N_q が和文 A で、 N_p が箱・グルー・カーンのいずれかであった場合、両者の間に挿入される JFM グルーについては同じ処理である。しかし、そこでの行分割に対する仕様が異なるので、ペナルティの挿入処理は若干異なったものとなっている。

- 「右空白」については、既に述べた Boundary-B [O_B] により空白を決定しようと試みる。それが失敗した場合は、「右空白」は挿入されない。
- 禁則用ペナルティの処理は、後ろのクラスタ N_p の種類によって異なる。なお、 $N_p.head$ は無意味であるから、「 $N_p.head$ に対する `prebreakpenalty` の値」は 0 とみなされる。言い換えれば、

$$a := (N_q \text{ の文字に対する } \text{postbreakpenalty} \text{ の値}).$$

箱 N_p が箱であった場合は、両クラスタの間での行分割は（明示的に両クラスタの間に `\penalty10000` があった場合を除き）いつも許容される。そのため、ペナルティ処理は、後に述べる P-allow [PA] が P-normal [PN] の代わりに用いられる。

グルー N_p がグルーの場合、ペナルティ処理は P-normal [PN] を用いる。

カーン N_p がカーンであった場合は、両クラスタの間での行分割は（明示的に両クラスタの間にペナルティがあった場合を除き）許容されない。ペナルティ処理は、後に述べる P-suppress [PS] を使う。

これらの P-normal [PN], P-allow [PA], P-suppress [PS] の違いは、 N_q と N_p の間（以前の図だと (a) の部分）にペナルティが存在しない場合にのみ存在する。

P-allow [PA] N_q と N_p の間の (a) 部分にペナルティがあれば、P-normal [PN] と同様に、それらの各ノードにおいてペナルティ値を a だけ増加させる。

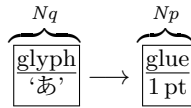
(a) 部分にペナルティが存在していない場合、`LuaTeX-ja` は N_q と N_p の間の行分割を可能にしようとする。そのために、以下の場合に a をもつ `penalty.node` を作って「右空白」の（もし未定義なら N_p の）直前に挿入する：

- 「右空白」がグルーでない（カーンか未定義）であるとき。

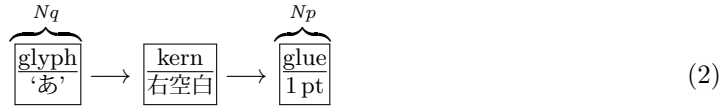
P-suppress [PS] N_q と N_p の間の (a) 部分にペナルティがあれば、P-normal [PN] と同様に、それらの各ノードにおいてペナルティ値を a だけ増加させる。

(a) 部分にペナルティが存在していない場合、 N_q と N_p の間の行分割は元々不可能のはずだったのであるが、`LuaTeX-ja` はそれをわざわざ行分割可能にはしない。そのため、「右空白」が `glue` であれば、その直前に `\penalty10000` を挿入する。

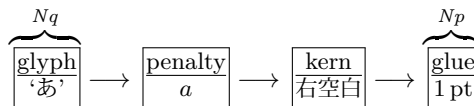
なお、「右空白」はカーンの



のような状況を考える。このとき、 a 、即ち「あ」の `postbreakpenalty` がいかなる値であっても、この 2 クラス間では最終的に



となり、 a 分のペナルティは挿入されないことに注意して欲しい。`postbreakpenalty` は (a は) 殆どの場合が非負の値と考えられ、そのような場合は (2) と



との間に差異は生じない^{*17}。

■箱・グルー・カーンと和文 A の間 Np が箱・グルー・カーンのいずれかで、 Np が和文 A であった場合は、すぐ上の (Nq と Np の順序が逆になっている) 場合と同じである。

- 「右空白」については、既に述べた Boundary-A [O_A] により空白を決定しようと試みる。それが失敗した場合は、「右空白」は挿入されない。
- 禁則用ペナルティの処理は、 Nq の種類によって異なる。 $Nq.tail$ は無意味なので、

$$a := (Np \text{ の文字に対する } \text{prebreakpenalty} \text{ の値}).$$

箱 Nq が箱の場合は、P-allow [PA] を用いる。

グルー Nq がグルーの場合は、P-normal [PN] を用いる。

カーン Nq がカーンの場合は、P-suppress [PS] を用いる。

■和文 A と和文 B の違い 先に述べたように、和文 B は hbox の中身の先頭 (or 末尾) として出現している和文文字である。リスト内に直接ノードとして現れている和文文字 (和文 A) との違いは、

- 和文 B に対しては、JFM の文字クラス指定から定まる空白 JFM-origin [M], Boundary-A [O_A], Boundary-B [O_B] の挿入は行われない。例えば、
 - 片方が和文 A, もう片方が和文 B のクラスタの場合, Boundary-A [O_A] または Boundary-B [O_B] の挿入を試み, それがダメなら `kanjiskip` [K] の挿入を行う。
 - 和文 B の 2 つのクラスタの間には, `kanjiskip` [K] が自動的に入る。
- 和文 B と箱・グルー・カーンが隣接したとき (どちらが前かは関係ない), 間に JFM グルー・ペナルティの挿入は一切しない。
- 和文 B と和文 B, また和文 B と欧文とが隣接した時は, 禁則用ペナルティ挿入処理は P-suppress [PS] が用いられる。
- 和文 B の文字に対する `prebreakpenalty`, `postbreakpenalty` の値は使われず, 0 として計算される。

次が具体例である：

^{*17} `kern`→`glue` が 1 つの行分割可能点 (行分割に伴うペナルティは 0) であるため, たとえ $a = 10000$ であっても, Nq と Np の間で行分割を禁止することはできない。

1	あ. \inhibitglue A\\	あ. A
2	\hbox{あ. }A\\	あ. A
3	あ. A	あ. A

- 1行目の\inhibitglueはBoundary-B [O_B]の処理のみを抑止するので、ピリオドと「A」の間にはxkanjiskip(四分アキ)が入ることに注意。
- 2行目のピリオドと「A」の間においては、前者が和文Bとなる(hboxの中身の末尾として登場しているから)ので、そもそもBoundary-B [O_B]の処理は行われない。よって、xkanjiskipが入ることとなる。
- 3行目では、ピリオドの属するクラスは和文Aである。これによって、ピリオドと「A」の間にはBoundary-B [O_B]由来の半角アキが入ることになる。

14 listings パッケージへの対応

`listings` パッケージが、そのままでは日本語をまともに出力できないことはよく知られている。きちんと整形して出力するために、`listings` パッケージは内部で「ほとんどの文字」をアクティブにし、各文字に対してその文字の出力命令を割り当てている ([2])。しかし、そこでアクティブにする文字の中に、和文文字がないためである。p_TE_X 系列では、和文文字をアクティブにする手法がなく、`jlisting.sty` というパッチ ([4]) を用いることで無理やり解決していた。

Lua_TE_X-ja では、`process_input_buffer` コールバックを利用することで、「各行に出現する U+0080 以降の文字に対して、それらの出力命令を前置する」という方法をとっている。出力命令としては、アクティブ文字化した U+FFFF を用いている。これにより、(入力には使用されていないかもしれない) 和文文字をもすべてアクティブ化する手間もなく、見通しが良い実装になっている。

Lua_TE_X-ja で利用される `listings` パッケージへのパッチ `lltjp-listings` は、`listings` と Lua_TE_X-ja を読み込んでおけば、`\begin{document}` の箇所において自動的に読み込まれるので、通常はあまり意識する必要はない。

14.1 注意

■**LaTeX へのエスケープ** 日本語対応を行うために `process_input_buffer` を使用したことで、`texcl, escapeinside` といった「LaTeX へのエスケープ」中では、`JAchar` を名称の一部に含む制御綴は使用不可能である。例えば次のような入力を考えよう：

```
\begin{lstlisting}[escapechar=\#]
#\\ほげ_x_びよ#
\end{lstlisting}
```

ここで、2行目は `process_input_buffer` の作用により、

```
#\\ほげ_x_びよ#
```

と変換されてから、実際の処理に回される。「#」で挟まれた「LaTeX へのエスケープ」中では U+FFFF のカテゴリーコードは 9 (*ignored*) となるので、結局「\`\`ほげ」の代わりに「\`\`」という control symbol が実行されることになる。

■**異体字セレクタの扱い** `lstlisting` 環境などの内部にある異体字セレクタを扱うため、`lltjp-listings` では `vsraw` と `vscmd` という2つのキーを追加した。しかし、`lltjp-listings` が実際にロードされるのは `\begin{document}` のところであるので、プレアンブル内ではこれらの追加キーは使用できない。

`ivsrw` は、ブール値の値をとるキーであり、標準では `false` である。

- `true` の場合は、異体字セレクタは「直前の文字に続けて」出力される。もしも IVS サポート (10.2 節) が有効になっていた場合は、以下の例 (左側は入力、右側はその出力) のようになる。

```
1 \begin{lstlisting}[vsraw=true]
2 葛0E0城市, 葛0E0飾区, 葛西           1 葛城市, 葛飾区, 葛西
3 \end{lstlisting}
```

- `false` の場合は、異体字セレクタは適当な命令によって「見える形で」出力される。どのような形で出力されるかを規定するのが `vscmd` キーであり、`lltjp-listings` の標準設定では以下の例の右側のように出力される。

```
1 \begin{lstlisting}[vsraw=false,
2   vscmd=\ltjlistingsvsstdcmd]           1 葛97城市, 葛98飾区, 葛西
3 葛0E0城市, 葛0E0飾区, 葛西
4 \end{lstlisting}
```

ちなみに、本ドキュメントでは次のようにしている：

```
1 \def\IVSA#1#2#3#4#5{%
2   \textcolor{blue}{\raisebox{3.5pt}{\tt%
3     \fboxsep=0.5pt\fbox{\tiny \oalign{0#1#2\cr cr#3#4#5\cr}}}}%
4 }
5 {\catcode\%=11
6   \gdef\IVSB#1{\expandafter\IVSA\directlua{
7     local cat_str = luatexbase.catcodetables['string']
8     tex.sprint(cat_str, string.format('%X', 0xE0EF+#1))
9   }}
10 \lstset{vscmd=\IVSB}
```

14.2 文字種

`listings` パッケージの内部では、大雑把に言うと

1. 識別子として使える文字 (“letter”, “digit”) たちを集める。
2. letter でも digit でもない文字が現れた時に、収集した文字列を (必要なら修飾して) 出力する。
3. 今度は逆に、letter でない文字たちを letter が現れるまで集める。
4. letter が出現したら集めた文字列を出力する。
5. 1. に戻る。

という処理が行われている。これにより、識別子の途中では行分割が行われなくなっている。直前の文字が識別子として使えるか否かは `\lst@ifletter` というフラグに格納されている。

さて、日本語の処理である。殆どの和文文字の前後では行分割が可能であるが、その一方で括弧類や音引きなどでは禁則処理が必要なことから、`lltjp-listings` では、直前が和文文字であるかを示すフラグ `\lst@ifkanji` を新たに導入した。以降、説明のために以下のように文字を分類する：

	Letter	Other	Kanji	Open	Close
<code>\lst@ifletter</code>	T	F	T	F	T
<code>\lst@ifkanji</code>	F	F	T	T	F
意図	識別子中の文字	その他欧文文字	殆どの和文文字	開き括弧類	閉じ括弧類

なお、本来の `listings` パッケージでの分類 “digit” は、出現状況によって、上の表の Letter と Other のどちらにもなりうる。また、Kanji と Close は `\lst@ifletter` と `\lst@ifkanji` の値が一致しているが、これは間違いではない。

例えば、Letter の直後に Open が来た場合を考える。文字種 Open は和文開き括弧類を想定しているので、Letter の直後では行分割が可能であることが望ましい。そのため、この場合では、すでに収集されている文字列を出力することで行分割を許容するようにした。

同じように、 $5 \times 5 = 25$ 通り全てについて書くと、次のようになる：

		後ろ側の文字				
		Letter	Other	Kanji	Open	Close
直	Letter	収集	_____	出力	_____	収集
前	Other	出力	収集	_____	出力	_____
文	Kanji	_____	_____	出力	_____	収集
字	Open	_____	_____	_____	収集	_____
種	Close	_____	_____	出力	_____	収集

上の表において、

- ・「出力」は、それまでに集めた文字列を出力（≡ここで行分割可能）を意味する。
- ・「収集」は、後ろ側の文字を、現在収集された文字列に追加（行分割不可）を意味する。

U+0080 以降の異体字セレクタ以外の各文字が Letter, Other, Kanji, Open, Close のどれに属するかは次によって決まる：

- ・(U+0080 以降の) **ALchar** は、すべて Letter 扱いである。
- ・**JAchar** については、以下の順序に従って文字種を決める：
 1. `prebreakpenalty` が 0 以上の文字は Open 扱いである。
 2. `postbreakpenalty` が 0 以上の文字は Close 扱いである。
 3. 上の 3 条件のどちらにも当てはまらなかった文字は、Kanji 扱いである。

なお、半角カナ (U+FF61–U+FF9F) 以外の **JAchar** は欧文文字 2 文字分の幅をとるものとみなされる。半角カナは欧文文字 1 文字分の幅となる。

これらの文字種決定は、実際に `lstlisting` 環境などの内部で文字が出てくるたびに行われる。

15 和文の行長補正方法

`luatexja-adjust` で提供される優先順位付きの行長調整の詳細を述べる。大まかに述べると、次のようになる。

- ・通常の $\text{T}_{\text{E}}\text{X}$ の行分割方法に従って、段落を行分割する。この段階では、行長に半端が出た場合、その半端分は **JAglue** (`xkanjiskip`, `kanjiskip`, JFM グルー) とそれ以外のグルーの全てで（優先順位なく）負担される。
- ・その後、`post_linebreak_filter` callback を使い、**段落中の各行ごとに**、行末文字の位置を調整したり、優先度付きの行長調整を実現するためにグルーの伸縮度を調整する。その処理においては、グルーの自然長と **JAglue** 以外のグルーの伸び量・縮み量は変更せず、必要に応じて **JAglue** の伸び量・縮み量のみを変更する設計とした。

`luatexja-adjust` の作用は、この処理を行う callback を追加するだけであり、この章の残りでは callback での処理について解説する。

■準備：合計伸縮量の計算 グルーの伸縮度（plus や minus で指定されている値）には、有限値の他に、fi, fil, fill, filll という4つの無限大レベル（後ろの方ほど大きい）があり、行の調整にfiなどの無限大レベルの伸縮度が用いられている場合は、その行に対しての処理を中止する。

よって、以降、問題にしている行の行長調整は伸縮度が有限長のグルーを用いて行われているとして良い。さらに、簡単のため、この行はグルーが広げられている（自然長で組むと望ましい行長よりの短い）場合しか扱わない。

まず、段落中の行中のグルーを

- JAglue ではないグルー
- JFM グルー（優先度^{*18}別にまとめられる）
- 和欧文間空白 (xkanjiskip)
- 和文間空白 (kanjiskip)

の $1 + 1 + 5 + 1 = 8$ つに類別し、それぞれの種別ごとに許容されている伸び量（stretch の値）の合計を計算する。また、行長と自然長との差を *total* とおく。

15.1 行末文字の位置調整

行末が文字クラス *n* の JAchar であった場合、それを動かすことによって、*total* のうち JAglue が負担する分を少なくしようとする。この行末文字の左右の移動可能量は、JFM 中にある文字クラス *n* の定義の *end_stretch*, *end_shrink* フィールドに全角単位の値として記述されている。

例えば、行末文字が句点「。」であり、そこで用いられている JFM 中に

```
[2] = {
  chars = { '。', ... }, width = 0.5, ...,
  end_stretch = 0.5, end_shrink = 0.5,
},
```

という指定があった場合、この行末の句点は

- 通常の T_EX の行分割処理で「半角以上の詰め」が行われていた場合、この行中の JAglue の負担を軽減するため、行末の句点を半角だけ右に移動する（ぶら下げ組を行う）。
- 通常の T_EX の行分割処理で「半角以上の空き」が行われていた場合、逆に行末句点を半角左に移動させる（見た目的に全角取りとなる）。
- 以上のどちらでもない場合、行末句点の位置調整は行わない。

となる。

行末文字を移動した場合、その分だけ *total* の値を引いておく。

15.2 グルーの調整

total の分だけが、行中のグルーの伸び量に応じて負担されることになる。負担するグルーの優先度は以下の順であり、できるだけ *kanjiskip* を自然長のままにすることを試みている。

- (A) JAglue 以外のグルー
- (B) 優先度 2 の JFM グルー
- (C) 優先度 1 の JFM グルー

^{*18} 6.3 節にあるように、各 JFM グルーには -2 から 2 までの優先度がついている。

- (D) 優先度 0 の JFM グルー
- (E) 優先度 -1 の JFM グルー
- (F) 優先度 -2 の JFM グルー
- (G) `xkanjiskip`
- (H) `kanjiskip`

1. 行末の和文文字を移動したことで $total = 0$ となれば、調整の必要はなく、行が格納されている `hbox` の `glue_set`, `glue_sign`, `glue_order` を再計算すればよい。以降、 $total \neq 0$ と仮定する。
2. $total$ が「**JAglue** 以外のグルーの伸び量の合計」(以下、(A) の伸び量の合計、と称す) よりも小さければ、それらのグルーに $total$ を負担させ、**JAglue** 達自身は自然長で組むことができる。よって、以下の処理を行う：
 - (1) 各 **JAglue** の伸び量を 0 とする。
 - (2) 行が格納されている `hbox` の `glue_set`, `glue_sign`, `glue_order` を再計算する。これによって、 $total$ は **JAglue** 以外のグルーによって負担される。
3. $total$ が「(A) の伸び量の合計」以上ならば、(A)–(H) のどこまで負担すれば $total$ 以上になるかを計算する。例えば、

$$total = ((A)-(B) \text{ の伸び量の合計}) + p \cdot ((C) \text{ の伸び量の合計}), \quad 0 \leq p < 1$$

であった場合、各グルーは次のように組まれる：

- (A), (B) に属するグルーは各グルーで許された伸び量まで伸ばす。
- (C) に属するグルーはそれぞれ $p \times$ (伸び量) だけ伸びる。
- (D)–(H) に属するグルーは自然長のまま。

実際には、前に述べた「設計」に従い、次のように処理している：

- (1) (C) に属するグルーの伸び量を p 倍する。
- (2) (D)–(H) に属するグルーの伸び量を 0 とする。
- (3) 行が格納されている `hbox` の `glue_set`, `glue_sign`, `glue_order` を再計算する。これによって、 $total$ は **JAglue** 以外のグルーによって負担される。
4. $total$ が (A)–(H) の伸び量の合計よりも大きい場合、どうしようもないので^^; 何もしない。

16 IVS 対応 (未完)

`luatexja.otf.enable_ivs()` を実行し、IVS 対応を有効にした状態では、`pre_linebreak_filter` や `hpack_filter` コールバックには次の 4 つが順に実行される状態となっている：

`ltj.do_ivs glyph_node p` の直後に、異体字セレクタ (を表す `glyph_node`) が連続した場合に、 p のフォントに対応したを持つ「異体字情報」に従って出力するグリフを変える。

しかし、単に $p.char$ を変更するだけでは、後から `font feature` の適用 (すぐ下) により置換される可能性がある。そのため、`\CID` や `\UTF` と同じように、`glyph_node p` の代わりに `user_id` が `char.by_cid` であるような user-defined whatsit を用いている。

(`luaotfload` による `font feature` の適用)

`ltj.otf user_id` が `char.by_cid` であるような user-defined whatsit をきちんと `glyph_node` に変換する。この処理は、`\CID`, `\UTF` や IVS による置換が、`font feature` の適用で上書きされてしまうのを防止するためである。

`ltj.main_process JAglue` の挿入処理 (13 章) と、JFM の指定に従って各 **JAchar** の「寸法を補正」することを行う。

問題は各フォントの持っている IVS 情報をどのように取得するか、である。 `luaotfload` はフォント番号 $\langle font_number \rangle$ の情報を `fonts.hashes.identifiers[$\langle font_number \rangle$]` 以下に格納している。しかし、OpenType フォントの IVS 情報は格納されていないようである^{*19}。

一方、LuaTeX 内部の `fontloader` の返すテーブルには OpenType フォントでも TrueType フォントでも IVS 情報が格納されている。具体的には……

そのため、LuaTeX-japan の IVS 対応においては、LuaTeX 内部の `fontloader` を直接用いることで、フォントの IVS 情報を取得している。20140114.0 以降でキャッシュを用いるようにした要因はここにあり、`fontloader` の呼び出しでかなり時間を消費することから、IVS 情報をキャッシュに保存することで 2 回目以降の実行時間を節約している。

17 複数フォントの「合成」(未完)

18 LuaTeX-japan におけるキャッシュ

`luaotfload` パッケージが、各 TrueType・OpenType フォントの情報をキャッシュとして保存していると同様の方法で、LuaTeX-japan もいくつかのキャッシュファイルを作成するようになった。

- 通常、キャッシュは `$TEXMFVAR/luatexja/` 以下に保存され、そこから読み込みが行われる。
- 「通常の」テキスト形式のキャッシュ (拡張子は `.lua`) 以外にも、それをバイナリ形式 (バイトコード) に変換したのもサポートしている。
 - LuaTeX と LuaJITTeX ではバイトコードの形式が異なるため、バイナリ形式のキャッシュは共有できない。LuaTeX 用のバイナリキャッシュは `.luc`、LuaJITTeX 用のは `.lub` と拡張子を変えることで対応している。
 - キャッシュを読み込む時、同名のバイナリキャッシュがあれば、テキスト形式のものよりそちらを優先して読み込む。
 - テキスト形式のキャッシュが更新/作成される際は、そのバイナリ版も同時に更新される。また、(バイナリ版が見つからず) テキスト形式のキャッシュ側が読み込まれたときは、LuaTeX-japan はバイナリキャッシュを作成する。

18.1 キャッシュの使用箇所

LuaTeX-japan では以下の 3 種類のキャッシュを使用している：

`ltj-cid-auto-adobe-japan1.lua` Ryumin-Light のような非埋め込みフォントの情報を格納しており、(それらが LuaTeX-japan の標準和文フォントなので) LuaTeX-japan の読み込み時に自動で読まれる。生成には `UniJIS2004-UTF32-H`、`Adobe-Japan1-UCS2` という 2 つの CMap が必要である。22 ページで述べたように、`cid` キーを使って非埋め込みの中国語・韓国語フォントを定義する場合、同様のキャッシュが生成される。キャッシュの名称、必要となる CMap については表 10 を参照して欲しい。

`ivs_***.lua` フォント “***” における異体字情報を格納している。構造は以下の通り：

```
return {  
  {
```

^{*19} TrueType フォントに関しては、

```
fonts.hashes.identifiers[ $\langle font\_number \rangle$ ].resources.variants[ $\langle selector \rangle$ ][ $\langle base\_char \rangle$ ]
```

に、 $\langle base_char \rangle$ 番の文字の後に異体字セレクタ ($\langle selector \rangle$) が続いた場合に出力すべきグリフが書かれてある。

表 10. cid key and corresponding files

cid key	name of the cache	used CMaps	
Adobe-Japan1-*	ltj-cid-auto-adobe-japan1.lua	UniJIS2004-UTF32-H	Adobe-Japan1-UCS2
Adobe-Korea1-*	ltj-cid-auto-adobe-korea1.lua	UniKS-UTF32-H	Adobe-Korea1-UCS2
Adobe-GB1-*	ltj-cid-auto-adobe-gb1.lua	UniGB-UTF32-H	Adobe-GB1-UCS2
Adobe-CNS1-*	ltj-cid-auto-adobe-cns1.lua	UniCNS-UTF32-H	Adobe-CNS1-UCS2

```
[10955]={ -- U+2ACB "Subset Of Above Not Equal To"
  [65024]=983879, -- <2ACB FE00>
},
[37001]={ -- U+9089 "邊"
  [0]=37001, -- <9089 E0100>
  991049, -- <9089 E0101>
  ...
},
...
},
["chksum"]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF", -- checksum of the fontfile
["version"]=4, -- version of the cache
}
```

ltj-jisx0208.{luc|lub} LuaTeX-ja 配布中の ltj-jisx0208.lua をバイナリ化したものである。これは JIS X 0208 と Unicode との変換テーブルであり、pTeX との互換目的の文字コード変換命令で用いられる。

18.2 内部命令

LuaTeX-ja におけるキャッシュ管理は、`luatexja.base` (`ltj-base.lua`) に実装しており、以下の 3 関数が公開されている。ここで、`<filename>` は保存するキャッシュのファイル名を拡張子なしで指定する。

`save_cache(<filename>, <data>)` nil でない `<data>` をキャッシュ `<filename>` に保存する。テキスト形式の `<filename>.lua` のみならず、そのバイナリ版も作成/更新される。

`save_cache_luc(<filename>, <data>[, <serialized_data>])`

`save_cache` と同様だが、バイナリキャッシュのみが更新される。第 3 引数 `<serialized_data>` が与えられた場合、それを `<data>` の文字列化表現として使用する。そのため、`<serialized_data>` は普通は指定しないことになるだろう。

`load_cache(<filename>, <outdate>)` キャッシュ `<filename>` を読み込む。`<outdate>` は 1 引数 (キャッシュの中身) をとる関数であり、その戻り値は「キャッシュの更新が必要」かどうかを示すブール値でないといけない。

`load_cache` は、まずバイナリキャッシュ `<filename>.{luc|lub}` を読みこむ。もしその内容が「新しい」、つまり `<outdate>` の評価結果が `false` なら `load_cache` はこのバイナリキャッシュの中身を返す。もしバイナリキャッシュが見つからなかったか、「古すぎる」ならばテキスト版 `<filename>.lua` を読み込み、その値を返す。

以上より、`load_cache` 自体が nil でない値を返すのは、ちょうど「新しい」キャッシュが見つかった場合である。

参考文献

- [1] Victor Eijkhout. *TeX by Topic, A T_EXnician's Reference*, Addison-Wesley, 1992.
- [2] C. Heinz, B. Moses. The Listings Package.
- [3] Takuji Tanaka. upTeX—Unicode version of pTeX with CJK extensions, TUG 2013, October 2013. http://tug.org/tug2013/slides/TUG2013_upTeX.pdf
- [4] Thor Watanabe. Listings - MyTeXpert.
<http://mytexpert.sourceforge.jp/index.php?Listings>
- [5] W3C Japanese Layout Task Force (ed). Requirements for Japanese Text Layout (W3C Working Group Note), 2011, 2012. <http://www.w3.org/TR/jlreq/>
日本語訳の書籍版：W3C 日本語組版タスクフォース（編），『W3C 技術ノート 日本語組版処理の要件』，東京電機大学出版局，2012.
- [6] 乙部巖己，min10 フォントについて。
<http://argent.shinshu-u.ac.jp/~otobe/tex/files/min10.pdf>
- [7] 日本工業規格 (Japanese Industrial Standard), JIS X 4051, 日本語文書の組版方法 (Formatting rules for Japanese documents), 1993, 1995, 2004.

付録 A Package versions used in this document

This document was typeset using the following packages:

geometry.sty	2010/09/12 v5.6 Page Geometry
ifvtex.sty	2010/03/01 v1.5 Detect VTeX and its facilities (H0)
ifxetex.sty	2010/09/12 v0.6 Provides ifxetex conditional
luatexja-adjust.sty	2013/05/14
expl3.sty	2014/01/07 v4646 L3 Experimental code bundle wrapper
l3names.sty	2014/01/04 v4640 L3 Namespace for primitives
l3bootstrap.sty	2014/01/04 v4640 L3 Experimental bootstrap code
l3basics.sty	2014/01/04 v4642 L3 Basic definitions
l3expan.sty	2014/01/04 v4642 L3 Argument expansion
l3tl.sty	2013/12/27 v4625 L3 Token lists
l3seq.sty	2013/12/14 v4623 L3 Sequences and stacks
l3int.sty	2013/08/02 v4583 L3 Integers
l3quark.sty	2013/12/14 v4623 L3 Quarks
l3prg.sty	2014/01/04 v4642 L3 Control structures
l3clist.sty	2013/07/28 v4581 L3 Comma separated lists
l3token.sty	2013/08/25 v4587 L3 Experimental token manipulation
l3prop.sty	2013/12/14 v4623 L3 Property lists
l3msg.sty	2013/07/28 v4581 L3 Messages
l3file.sty	2013/10/13 v4596 L3 File and I/O operations
l3skip.sty	2013/07/28 v4581 L3 Dimensions and skips
l3keys.sty	2013/12/08 v4614 L3 Experimental key-value interfaces
l3fp.sty	2014/01/04 v4642 L3 Floating points
l3box.sty	2013/07/28 v4581 L3 Experimental boxes
l3coffins.sty	2013/12/14 v4624 L3 Coffin code layer
l3color.sty	2012/08/29 v4156 L3 Experimental color support
l3luatex.sty	2013/07/28 v4581 L3 Experimental LuaTeX-specific functions
l3candidates.sty	2014/01/06 v4643 L3 Experimental additions to l3kernel
amsmath.sty	2013/01/14 v2.14 AMS math features
amstext.sty	2000/06/29 v2.01
amsgen.sty	1999/11/30 v2.0
amsbsy.sty	1999/11/29 v1.2d
amsopn.sty	1999/12/14 v2.01 operator names
array.sty	2008/09/09 v2.4c Tabular extension package (FMI)
tikz.sty	2013/12/13 v3.0.0 (rcs-revision 1.142)
pgf.sty	2013/12/18 v3.0.0 (rcs-revision 1.14)
pgfrcs.sty	2013/12/20 v3.0.0 (rcs-revision 1.28)
everyshi.sty	2001/05/15 v3.00 EveryShipout Package (MS)
pgfcore.sty	2010/04/11 v3.0.0 (rcs-revision 1.7)
graphicx.sty	1999/02/16 v1.0f Enhanced LaTeX Graphics (DPC,SPQR)
graphics.sty	2009/02/05 v1.0o Standard LaTeX Graphics (DPC,SPQR)
trig.sty	1999/03/16 v1.09 sin cos tan (DPC)
pgfsys.sty	2013/11/30 v3.0.0 (rcs-revision 1.47)
xcolor.sty	2007/01/21 v2.11 LaTeX color extensions (UK)
pgfcomp-version-0-65.sty	2007/07/03 v3.0.0 (rcs-revision 1.7)
pgfcomp-version-1-18.sty	2007/07/23 v3.0.0 (rcs-revision 1.1)
pgffor.sty	2013/12/13 v3.0.0 (rcs-revision 1.25)
pgfkeys.sty	
pgfmath.sty	
pict2e.sty	2014/01/12 v0.2z Improved picture commands (HjG,RN,JT)
multienum.sty	
float.sty	2001/11/08 v1.3d Float enhancements (AL)
booktabs.sty	2005/04/14 v1.61803 publication quality tables
multicol.sty	2011/06/27 v1.7a multicolumn formatting (FMI)
listings.sty	2014/03/04 1.5c (Carsten Heinz)
lstmisc.sty	2014/03/04 1.5c (Carsten Heinz)
showexpl.sty	2014/01/19 v0.3l Typesetting example code (RN)
calc.sty	2007/08/22 v4.3 Infix arithmetic (KKT,FJ)
ifthen.sty	2001/05/26 v1.1c Standard LaTeX ifthen package (DPC)
varwidth.sty	2009/03/30 ver 0.92; Variable-width minipages

hyperref.sty	2012/11/06 v6.83m Hypertext links for LaTeX
hobsub-hyperref.sty	2012/05/28 v1.13 Bundle oberdiek, subset hyperref (HO)
hobsub-generic.sty	2012/05/28 v1.13 Bundle oberdiek, subset generic (HO)
hobsub.sty	2012/05/28 v1.13 Construct package bundles (HO)
intcalc.sty	2007/09/27 v1.1 Expandable calculations with integers (HO)
etexcmds.sty	2011/02/16 v1.5 Avoid name clashes with e-TeX commands (HO)
kvsetkeys.sty	2012/04/25 v1.16 Key value parser (HO)
kvdefinekeys.sty	2011/04/07 v1.3 Define keys (HO)
pdfescape.sty	2011/11/25 v1.13 Implements pdfTeX's escape features (HO)
bigintcalc.sty	2012/04/08 v1.3 Expandable calculations on big integers (HO)
bitset.sty	2011/01/30 v1.1 Handle bit-vector datatype (HO)
uniquecounter.sty	2011/01/30 v1.2 Provide unlimited unique counter (HO)
letltxmacro.sty	2010/09/02 v1.4 Let assignment for LaTeX macros (HO)
hopatch.sty	2012/05/28 v1.2 Wrapper for package hooks (HO)
xcolor-patch.sty	2011/01/30 xcolor patch
atveryend.sty	2011/06/30 v1.8 Hooks at the very end of document (HO)
atbegshi.sty	2011/10/05 v1.16 At begin shipout hook (HO)
refcount.sty	2011/10/16 v3.4 Data extraction from label references (HO)
hycolor.sty	2011/01/30 v1.7 Color options for hyperref/bookmark (HO)
auxhook.sty	2011/03/04 v1.3 Hooks for auxiliary files (HO)
kvoptions.sty	2011/06/30 v3.11 Key value format for package options (HO)
url.sty	2013/09/16 ver 3.4 Verb mode for urls, etc.
rerunfilecheck.sty	2011/04/15 v1.7 Rerun checks for auxiliary files (HO)
bookmark.sty	2011/12/02 v1.24 PDF bookmarks (HO)
amsthm.sty	2004/08/06 v2.20
luatexja-otf.sty	2013/05/14
luatexja-ajmacros.sty	2013/05/14
lmodern.sty	2009/10/30 v1.6 Latin Modern Fonts
luatexja-preset.sty	2013/10/28 Japanese font presets
luatexja-fontspec.sty	2014/01/23 fontspec support of LuaTeX-ja
fontspec.sty	2013/05/20 v2.3c Font selection for XeLaTeX and LuaLaTeX
xparse.sty	2013/12/31 v4634 L3 Experimental document command parser
fontspec-patches.sty	2013/05/20 v2.3c Font selection for XeLaTeX and LuaLaTeX
fixltx2e.sty	2006/09/13 v1.1m fixes to LaTeX
fontspec-luatex.sty	2013/05/20 v2.3c Font selection for XeLaTeX and LuaLaTeX
fontenc.sty	
xunicode.sty	2011/09/09 v0.981 provides access to latin accents and many other characters in Unicode lower plane
amssymb.sty	2013/01/14 v3.01 AMS font symbols
amsfonts.sty	2013/01/14 v3.01 Basic AMSFonts support
metalogo.sty	2010/05/29 v0.12 Extended TeX logo macros
lltjp-fontspec.sty	2013/05/14 Patch to fontspec for LuaTeX-ja
lltjp-xunicode.sty	2013/05/14 Patch to xunicode for LuaTeX-ja
lltjp-listings.sty	2014/01/09 Patch to listings for LuaTeX-ja
epstopdf-base.sty	2010/02/09 v2.5 Base part for package epstopdf
grfext.sty	2010/08/19 v1.1 Manage graphics extensions (HO)
nameref.sty	2012/10/27 v2.43 Cross-referencing by name of section
getttitlestring.sty	2010/12/03 v1.4 Cleanup title references (HO)