The Lua\TeX-ja package

The Lua\TeX-ja project team

20190408.0 (April 8, 2019)
8 Parameters

8.1 \ltxsetparameter ........................................ 30
8.2 \ltxgetparameter ....................................... 32
8.3 Alternative Commands to \ltxsetparameter ............... 33

9 Other Commands for plain \TeX{} and \LaTeX{} 2e

9.1 Commands for compatibility with \pTeX{} .......................... 34
9.2 \inhibitglue .................................................. 34
9.3 \ltjfakeboxbdd, \ltjfakeparbegin ............................ 35
9.4 \ltjdeclarealtfont ......................................... 35

10 Commands for \LaTeX{} 2e

10.1 Loading Japanese fonts in \LaTeX{} 2e ......................... 36
10.2 Patch for NFSS2 ............................................. 36
10.3 Detail of \fontfamily{} command ............................ 38
10.4 Notes on \DeclareTextSymbol .............................. 39
10.5 \strutbox ..................................................... 39

11 Addon packages

11.1 luatexja-fontspec ......................................... 39
11.2 luatexja-off ............................................... 41
11.3 luatexja-adjust ............................................ 41
11.4 luatexja-ruby .............................................. 41
11.5 \ltjext.sty .................................................. 42
11.6 luatexja-preset ............................................ 43
11.6.1 General Options ......................................... 43
11.6.2 Presets which support multi weights .................... 45
11.6.3 Presets which do not support multi weights ............ 47
11.6.4 Presets which use HG fonts ............................ 47
11.6.5 Define/Use Custom Presets .............................. 48

III Implementations

12 Storing Parameters ........................................ 49
12.1 Used dimensions, attributes and whatsit nodes ............ 49
12.2 Stack system of Lua\TeX{}-ja ................................ 50
12.3 Lua functions of the stack system .......................... 51
12.4 Extending Parameters ...................................... 51

13 Linebreak after a Japanese Character .................... 52
13.1 Reference: behavior in \pTeX{} .............................. 52
13.2 Behavior in Lua\TeX{}-ja ..................................... 53

14 Patch for the \texttt{listings} Package ....................... 54
14.1 Notes and additional keys .................................. 54
14.2 Class of characters ........................................ 55

15 Cache Management of Lua\TeX{}-ja ......................... 56
15.1 Use of cache ............................................... 56
15.2 Internal ...................................................... 57
This documentation is far from complete. It may have many grammatical (and contextual) errors. Also, several parts are written in Japanese only.
Part I
User’s manual

1 Introduction

The LuaTEX-ja package is a macro package for typesetting high-quality Japanese documents when using LuaTEX.

1.1 Backgrounds

Traditionally, ASCII pTEX, an extension of TEX, and its derivatives are used to typeset Japanese documents in TEX. pTEX is an engine extension of TEX: so it can produce high-quality Japanese documents without using very complicated macros. But this point is a mixed blessing: pTEX is left behind from other extensions of TEX, especially $\varepsilon$TEX and pdfTEX, and from changes about Japanese processing in computers (e.g., the UTF-8 encoding).

Recently extensions of pTEX, namely upTEX (Unicode-implementation of pTEX) and $\varepsilon$-pTEX (merging of pTEX and $\varepsilon$-TEX extension), have developed to fill those gaps to some extent, but gaps still exist.

However, the appearance of LuaTEX changed the whole situation. With using Lua "callbacks", users can customize the internal processing of LuaTEX. So there is no need to modify sources of engines to support Japanese typesetting: to do this, we only have to write Lua scripts for appropriate callbacks.

1.2 Major changes from pTEX

The LuaTEX-ja package is under much influence of pTEX engine. The initial target of development was to implement features of pTEX. However, implementing all feature of pTEX is impossible, since all process of LuaTEX-ja must be implemented only by Lua and TEX macros. Hence LuaTEX-ja is not a just porting of pTEX; unnatural specifications/behaviors of pTEX were not adopted.

The followings are major changes from pTEX. For more detailed information, see Part III or other sections of this manual.

Command names  pTEX adds several primitives, such as \kanjiskip, \prebreakpenalty, and \ifydir. They can be used as follows:

\kanjiskip=10pt \dimen0=kanjiskip
\tbaselineshift=0.1zw \dimen0=tbaselineshift
\prebreakpenalty\ \\hskip=100
\ifydir ... \fi

However, we cannot use them under LuaTEX-ja. Instead of them, we have to write as the following.

\ltjsetparameter{kanjiskip=10pt} \dimen0=\ltjgetparameter{kanjiskip}
\ltjsetparameter{tbaselineshift=0.1\zw} \dimen0=\ltjgetparameter{tbaselineshift}
\ltjsetparameter{prebreakpenalty={\hskip,100}} \ifnum\ltjgetparameter{direction}=4 ... \fi

Note that pTEX adds new two useful units, namely zw and zh. As shown above, they are changed by \zw and \zh respectively, in LuaTEX-ja.

Linebreak after a Japanese character  In pTEX, a line break after Japanese character is ignored (and doesn’t yield a space), since line breaks (in source files) are permitted almost everywhere in Japanese texts. However, LuaTEX-ja doesn’t have this feature completely, because of a specification of LuaTEX. For the detail, see Section 13.
Spaces related to Japanese characters  The insertion process of glues/kerns between two Japanese characters and between a Japanese character and other characters (we refer glues/kerns of both kinds as JAglue) is rewritten from scratch.

• As LuaTeX’s internal ligature handling is node-based (e.g., `of{}` doesn’t prevent ligatures), the insertion process of JAglue is now node-based.
• Furthermore, nodes between two characters which have no effects in line break (e.g., `\special` node) and kerns from italic correction are ignored in the insertion process.
• Caution: due to above two points, many methods which did for the dividing the process of the insertion of JAglue in \TeX are not effective anymore. In concrete terms, the following two methods are not effective anymore:
  ちょ{}っと ちょ／っと

  If you want to do so, please put an empty horizontal box (hbox) between it instead:
  ちょ\hbox{}っと

• In the process, two Japanese fonts which only differ in their ”real” fonts are identified.

Directions  From version 20150420.0, LuaTeX-ja supports vertical writing. We implement this feature by using callbacks of LuaTeX; so it must not be confused with \Omega-style direction support of LuaTeX itself. Due to implementation, the dimension returned by `\wd`, `\ht`, or `\dp` depends on the content of the register only. This is major difference with \TeX.

\discretionary  Japanese characters in discretionary break (\discretionary) is not supported.

Greek and Cyrillic letters, and ISO 8859-1 symbols  By default, LuaTeX-ja uses Japanese fonts to typeset Greek and Cyrillic letters. To change this behavior, put `\ltjsetparameter{jacharrange={-2,-3}}` in the preamble. For the detailed description, see Subsection 4.1.

From version 20150906.0, characters which belongs both ISO 8859-1 and JIS X 0208, such as ¶ and §, are now typeset in alphabetic fonts. If you are using \TeX prior to 2017/01/01, these characters are not typeset correctly without the `\fontspec` (and luatexja-fontspec) package.

1.3 Notations

In this document, the following terms and notations are used:

• Characters are classified into following two types. Note that the classification can be customized by a user (see Subsection 4.1).
  -- JAchar: standing for characters which is used in Japanese typesetting, such as Hiragana, Katakana, Kanji, and other Japanese punctuation marks.
  -- ALchar: standing for all other characters like latin alphabets.

  We say alphabetic fonts for fonts used in ALchar, and Japanese fonts for fonts used in JAchar.
• A word in a sans-serif font with underline (like `\prebreakpenalty`) means an internal parameter for Japanese typesetting, and it is used as a key in `\ltjsetparameter` command.
• A word in a sans-serif font without underline (like `\fontspec`) means a package or a class of \TeX.
• In this document, natural numbers start from zero. \omega denotes the set of all natural numbers which can be used in \TeX.
1.4 About the project

Project Wiki  Project Wiki is under construction.
- [https://osdn.jp/projects/luatex-ja/wiki/FrontPage\%28en\%29](https://osdn.jp/projects/luatex-ja/wiki/FrontPage\%28en\%29) (English)

This project is hosted by OSDN.

Members

- Hironori KITAGAWA
- Yusuke KUROKI
- Tomoaki HONDA
- Kazuki MAEDA
- Noriyuki ABE
- Shuzaburo SAITO
- Takayuki YATO
- Munehiro YAMAMOTO
- MA Qiuyuan
2 Getting Started

2.1 Installation

The following packages are needed for the \texttt{Luatex-ja} package.

- \texttt{luaotfload} v2.6 (or later)
- \texttt{adobemapping} (Adobe cmap and pdfmapping files)
- \texttt{etoolbox}, \texttt{everyset} (if you want to use \texttt{Luatex-ja} with \texttt{LATEX \varepsilon})
- \texttt{filehook}, \texttt{atbegshi}
- \texttt{IPAex fonts} (http://ipafont.ipa.go.jp/)

In summary, \texttt{Luatex-ja} version 20180825.0 (or later) no longer supports \texttt{TEX Live} 2016.

Now \texttt{Luatex-ja} is available from CTAN (in the \texttt{macros/luatex/generic/luatexja} directory), and the following distributions:

- \texttt{MiKTeX} (in \texttt{luatexja.tar.lzma}); see the next subsection
- \texttt{TEX Live} (in \texttt{texmf-dist/tex/luatex/luatexja})
- \texttt{W32TEX} (in \texttt{luatexja.tar.xz})

IPAex fonts are also available in these distributions.

### Manual installation

1. Download the source, by one of the following method. At the present, \texttt{Luatex-ja} has no stable release.
   - Clone the Git repository:
     
     ```bash
     $ git clone git://git.osdn.jp/gitroot/luatex-ja/luatexja.git
     ```
   - Download the tar.gz archive of HEAD in the master branch from
     
     http://git.osdn.jp/view?p=luatex-ja/luatexja.git;a=snapshot;h=HEAD;sf=tgz.

   Note that the master branch, and hence the archive in CTAN, are not updated frequently; the forefront of development is not the master branch.

2. Extract the archive. You will see \texttt{src/} and several other sub-directories. But only the contents in \texttt{src/} are needed to work \texttt{Luatex-ja}.

3. If you downloaded this package from CTAN, you have to run following commands to generate classes and \texttt{ltj-kinsoku.lua} (the file which stores default "kinsoku" parameters):

   ```bash
   $ cd src
   $ lualatex ltjclasses.ins
   $ lualatex ltjclasses.ins
   $ lualatex ltjltxdoc.ins
   $ luatex ltj-kinsoku_make.tex
   
   Do not forget The last line (processing \texttt{ltj-kinsoku_make.tex}). \*\{dtx,ins\} and \texttt{ltj-kinsoku_make.tex} used here are not needed in regular use.

4. Copy all the contents of \texttt{src/} into one of your \texttt{TEXMF} tree. \texttt{TEXMF/tex/luatex/luatexja/} is an example location. If you cloned entire Git repository, making a symbolic link of \texttt{src/} instead copying is also good.

5. If \texttt{mktexlsr} is needed to update the file name database, make it so.
2.2 Cautions

For changes from p\TeX, see Subsection 1.2.

- The encoding of your source file must be UTF-8. Other encodings, such as EUC-JP or Shift-JIS, are not supported.

- Lua\TeX-ja is very slower than p\TeX. Generally speaking, LuaJIT\TeX processes Lua\TeX-ja about 30% faster than Lua\TeX, but not always\(^1\).

- (Outdated) note for MiKTEX users Lua\TeX-ja requires that several CMap files\(^2\) must be found from Lua\TeX. Strictly speaking, those CMaps are needed only in the first run of Lua\TeX-ja after installing or updating. But it seems that MiKTEX does not satisfy this condition, so you will encounter an error like the following:

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

If so, please execute a batch file which is written on the Project Wiki (English). This batch file creates a temporary directory, copy CMaps in it, run a test file which loads Lua\TeX-ja in this directory, and finally delete the temporary directory.

2.3 Using in plain \TeX

To use Lua\TeX-ja in plain \TeX, simply put the following at the beginning of the document:

\input luatexja.sty

This does minimal settings (like ptex.tex) for typesetting Japanese documents:

- The following 12 Japanese fonts are preloaded:

<table>
<thead>
<tr>
<th>direction</th>
<th>classification</th>
<th>font name</th>
<th>&quot;10 pt&quot;</th>
<th>&quot;7 pt&quot;</th>
<th>&quot;5 pt&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>yoko (horizontal)</td>
<td>mincho</td>
<td>IPAex Mincho</td>
<td>\tenmin</td>
<td>\sevenmin</td>
<td>\fivemin</td>
</tr>
<tr>
<td></td>
<td>gothic</td>
<td>IPAex Gothic</td>
<td>\tengt</td>
<td>\sevengt</td>
<td>\fivelt</td>
</tr>
<tr>
<td>tate (vertical)</td>
<td>mincho</td>
<td>IPAex Mincho</td>
<td>\tentmin</td>
<td>\seventmin</td>
<td>\fivetmin</td>
</tr>
<tr>
<td></td>
<td>gothic</td>
<td>IPAex Gothic</td>
<td>\tentgt</td>
<td>\seventgt</td>
<td>\fiveltg</td>
</tr>
</tbody>
</table>

- The "default" Japanese fonts (and JFMs for them) can be modified by defining \ltj@stdmcfont etc. before one inputs luatexja.sty (Subsection 7.3).
- A character in an alphabetic font is generally smaller than a Japanese font in the same size. So actual size specification of these Japanese fonts is in fact smaller than that of alphabetic fonts, namely scaled by 0.962216.

- The amount of glue that are inserted between a JAchar and an ALchar (the parameter xkanjiskip) is set to

\[(0.25 \cdot 0.962216 \cdot 10 \text{ pt})^{+1\text{ pt}} = 2.40554 \text{ pt}\]

2.4 Using in L\ATeX

Using in L\ATEX\(2\epsilon\) is basically same. To set up the minimal environment for Japanese, you only have to load luatexja.sty:

\usepackage{luatexja}

It also does minimal settings (counterparts in p\\LaTeX are p1fonts.dtx and p1defs.1tx).

- Font encodings for Japanese fonts are JY3 (for horizontal direction) and JT3 (for vertical direction).

\(^1\) LuaJIT has several limitations such as 1 GB (or 2 GB) memory limitation. So typesetting a large source by LuaJIT\TeX may cause an "out of memory" error, or failure of loading/saving font cache of luaotfload.

• Traditionally, Japanese documents use only two families: mincho (明体) and gothic (ゴシック体). mincho is used in the main text, while gothic is used in the headings or for emphasis.

<table>
<thead>
<tr>
<th>classification</th>
<th>commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho (明体)</td>
<td>\textmc{...} {mcfamily ...} \mcfamily \mcfamilydefault</td>
</tr>
<tr>
<td>gothic (ゴシック体)</td>
<td>\textgt{...} {gtfamily ...} \gtfamily \gtdefault</td>
</tr>
</tbody>
</table>

• By default, the following fonts are used for these two families.

<table>
<thead>
<tr>
<th>classification</th>
<th>family</th>
<th>\mdseries</th>
<th>\bfsseries</th>
<th>scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho (明体)</td>
<td>mc</td>
<td>IPAex Mincho</td>
<td>IPAex Gothic</td>
<td>0.962216</td>
</tr>
<tr>
<td>gothic (ゴシック体)</td>
<td>gt</td>
<td>IPAex Gothic</td>
<td>IPAex Gothic</td>
<td>0.962216</td>
</tr>
</tbody>
</table>

• Note that the bold series (series bx or b) in both family are same as the medium series of gothic family. There is no italic nor slanted shape for these mc and gt.

• From version 20181102.0, one can specifies disablejfam option at loading LuaTEX-ja. This option prevents loading a patch for \LaTeX, which are needed to support Japanese characters in math mode. Without disablejfam option, one can typeset Japanese characters in math mode as $あ$ (see Page 10) as before. Japanese characters in math mode are typeset by the font family mc.

\begin{verbatim}
\renewcommand{\kanjifamilydefault}{\gtdefault}
\end{verbatim}

However, above settings are not sufficient for Japanese-based documents. To typeset Japanese-based documents, you are better to use class files other than \article.cls, \book.cls, and so on. At the present, Lua\TeX-ja has the counterparts of jclasses (standard classes in \pLaTeX) and jsclasses (classes by Haruhiko Okumura), namely, ltjclasses\(^3\) and ltjsclasses\(^5\).

Original jsclasses use \mag primitive to set the main document font size. However, Lua\TeX beta-0.87.0 or later does not support \mag in PDF output, so ltjsclasses uses the nomag\(^6\) option by default to set the main font size. If this causes some unexpected behavior, specify nomag option in \documentclass.

geometry package and classes for vertical writing  It is well-known that the geometry package produces the following error, when classes for vertical writing is used:

\begin{verbatim}
! Incompatible direction list can’t be unboxed.
\@begindvi \rightarrow \unvbox \@begindvibox \global \let \@begindvi \empty
\end{verbatim}

Now, Lua\TeX-ja automatically applies the patch \ttjpe-paper to the geometry package, when the direction of the document is tate (vertical writing). This patch \ttjpe-paper also can be used in \platex\ for the detail, please refer \ttjpe-paper.pdf (Japanese).

3 Changing Fonts

3.1 plain \TeX and \platex 2ε

plain \TeX  To change Japanese fonts in plain \TeX, you must use the command \jfont and \tfont. So please see Subsection 7.1.\

\(^3\)When ltjclasses classes are used, or luatexjs-fontspec (or luatexjs-preset) is loaded with match option, \ttfamily changes the current Japanese font family to \ttdefault. These classes and packages also redefine \ttdefault to \gtdefault (gothic family).

\(^4\)ltjarticle.cls, ltjbook.cls, ltjreport.cls, ltjarticle.cls, ltjbook.cls, ltjreport.cls. The latter ltjt*.cls are for vertically written Japanese documents.

\(^5\)ltjsarticle.cls, ltjsbook.cls, ltjsreport.cls, ltjskiyou.cls.

\(^6\)Same effect as the BXjscls classes (by Takayuki Yato) and jsclasses. However, these classes uses only \TeX code, but ltjsclasses uses Lua code.
LaTeX (NFSS2) For LaTeX, LuaTeX-ja adopted most of the font selection system of pLaTeX (in pflfonts.dtx).

- Commands \fontfamily, \fontseries, and \fontshape can be used to change attributes of Japanese fonts.

<table>
<thead>
<tr>
<th>encoding</th>
<th>family</th>
<th>series</th>
<th>shape</th>
<th>selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabetic fonts</td>
<td>\romanencoding</td>
<td>\romanfamily</td>
<td>\romanseries</td>
<td>\romanshape \useroman</td>
</tr>
<tr>
<td>Japanese fonts</td>
<td>\kanjiencoding</td>
<td>\kanjifamily</td>
<td>\kanjiseries</td>
<td>\kanjishape \usekanji</td>
</tr>
<tr>
<td>both</td>
<td>–</td>
<td>–</td>
<td>\fontseries</td>
<td>\fontshape –</td>
</tr>
<tr>
<td>auto select</td>
<td>\fontencoding</td>
<td>\fontfamily</td>
<td>–</td>
<td>\usefont</td>
</tr>
</tbody>
</table>

\fontencoding{⟨encoding⟩} changes the encoding of alphabetic fonts or Japanese fonts depending on the argument. For example, \fontencoding{JY3} changes the encoding of Japanese fonts to JY3, and \fontencoding{T1} changes the encoding of alphabetic fonts to T1. \fontfamily also changes the current Japanese font family, the current alphabetic font family, or both. For the detail, see Subsection 10.2.

- For defining a Japanese font family, use \DeclareKanjiFamily instead of \DeclareFontFamily. (In previous version of LuaTeX-ja, using \DeclareFontFamily didn’t cause any problem. But this no longer applies the current version.)
- Defining a Japanese font shape can be done by usual \DeclareFontShape:

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

### Japanese characters in math mode

Since pLaTeX supports Japanese characters in math mode, there are sources like the following:

1. \[ f_{\text{高温}} \quad (f_{\text{high temperature}}). \]
2. \[
\begin{align*}
\frac{y=(x-1)^2+2 \quad \text{よって} \quad y>0}{y=(x-1)^2+2 \quad \text{よって} \quad y>0}
\end{align*}
\]
3. \[
\text{素} := \{ p \in \mathbb{N} : p \text{ is a prime} \}
\]

We (the project members of LuaTeX-ja) think that using Japanese characters in math mode are allowed if and only if these are used as identifiers. In this point of view,

- The lines 1 and 2 above are not correct, since “高温” in above is used as a textual label, and “よって” is used as a conjunction.

- However, the line 3 is correct, since “素” is used as an identifier.

- Hence, in our opinion, the above input should be corrected as:

We also believe that using Japanese characters as identifiers is rare, hence we don’t describe how to change Japanese fonts in math mode in this chapter. For the method, please see Subsection 7.6.

When LuaTeX-ja is loaded with \disabelfam option, one cannot write Japanese characters in math mode as \$素\$. At that case, one have to use \mbox (or \text in the amsmath package).
Table 1. Commands of luatexja-fontspec

<table>
<thead>
<tr>
<th>Japanese fonts</th>
<th>\jfontspec</th>
<th>\setmainjfont</th>
<th>\setsansjfont</th>
<th>\setmonojfont*</th>
</tr>
</thead>
<tbody>
<tr>
<td>alphabetic fonts</td>
<td>\fontspec</td>
<td>\setmainfont</td>
<td>\setsansfont</td>
<td>\setmonofont</td>
</tr>
<tr>
<td>Japanese fonts</td>
<td>\newjfontfamily</td>
<td>\renewjfontfamily&quot;</td>
<td>\setjfontfamily&quot;</td>
<td>\setjfontfamily&quot;</td>
</tr>
<tr>
<td>alphabetic fonts</td>
<td>\newfontfamily</td>
<td>\renewfontfamily&quot;</td>
<td>\setjfontfamily&quot;</td>
<td>\setjfontfamily&quot;</td>
</tr>
<tr>
<td>Japanese fonts</td>
<td>\newjfontface</td>
<td>\defaultjfontfeatures</td>
<td>\addjfontfeatures</td>
<td>\addjfontfeatures</td>
</tr>
<tr>
<td>alphabetic fonts</td>
<td>\newfontface</td>
<td>\defaultfontfeatures</td>
<td>\addfontfeatures</td>
<td>\addfontfeatures</td>
</tr>
</tbody>
</table>

*"\setmonojfont is defined if and only if match option is specified.
**\renewjfontfamily and \setjfontfamily are new commands in fontspec v2.6h (2018/7/30). Hence, luatexja-fontspec define \renewjfontfamily and \setjfontfamily only when fontspec v2.6h (or later) is used.

3.2 luatexja-fontspec package

To use the functionality of the fontspec package to Japanese fonts, it is needed to load the luatexja-fontspec package in the preamble, as follows:

\usepackage[⟨options⟩]{luatexja-fontspec}

This luatexja-fontspec package automatically loads luatexja and fontspec packages, if needed.

In the luatexja-fontspec package, several commands are defined as counterparts of original commands in the fontspec package (see Table 1):

The package option of luatexja-fontspec are the followings:

match
If this option is specified, usual family-changing commands such as \rmfamily, \textrm, \sffamily, … also change Japanese font family.

Note that \setmonojfont is defined if and only if this match option is specified.

pass=⟨opts⟩
(Obsoleted) Specify options ⟨opts⟩ which will be passed to the fontspec package.

scale=⟨float⟩
Override the ratio of the font size of Japanese fonts to that of alphabetic fonts. The default value is determined as follows:

- The value of \Cjascale is used, if this control sequence is already defined.
- It is calculated automatically from the current Japanese font at the loading of the package, if \Cjascale is not defined.

\Cjascale is defined in ltjclasses and ltjclasses.

All other options listed above are simply passed to the fontspec package. This means that two lines below are equivalent, for example.

\usepackage[no-math]{fontspec}\usepackage{luatexja-fontspec}
\usepackage[no-math]{luatexja-fontspec}

The reason that \setmonojfont is not defined by default is that it is popular for Japanese fonts that nearly all Japanese glyphs have same widths. Also note that kerning information in a font is not used (that is, kern feature is set off) by default in these seven (or eight) commands. This is because of the compatibility with previous versions of LuaTeX-ja (see 7.1).

Below is an example of \jfontspec.

1 \jfontspec[CJKShape=NLC]{KozMinPr6N-Regular}
2 JIS-X-2013:2004 →辻
3 \jfontspec[CJKShape=JIS1990]{KozMinPr6N-Regular}
4 JIS X 2013:2004 →辻
5 JIS-X-0213:2004 →辻
6 JIS X 2013:2004 →辻
7 JIS-X-0208:1990 →辻
8 JIS X 2013:2004 →辻
9 JIS-X-0213:2004 →辻
10 JIS X 2013:2004 →辻
3.3 Presets of Japanese fonts

With luatexja-preset package, one use one of "preset" to simplify Japanese font setting. For details of package options, and those of each presets, please see Subsecion 11.6. The following presets are defined:

hiragino-pro, hiragino-pron, ipa, ipa-hg, ipaex, ipaex-hg, kozuka-pr6, kozuka-pr6n, kozuka-pro, moga-mobo, moga-mobo-ex,bizud, morisawa-pr6n, morisawa-pr6, ms, ms-hg, noembed, noto-otc, noto-otf, sourcehan, sourcehan-jp, ume, yu-osx, yu-win, yu-win10

For example, this document loads luatexja-preset package by
\usepackage[kozuka-pr6n]{luatexja-preset}

which means that Kozuka Pr6N fonts will be used in this document.

3.4 CID, UTF, and macros in japanese-otf package

Under pd\TeX, japanese-otf package (developed by Shuzaburo Saito) is used for typesetting characters which is in Adobe-Japan1-6 CID but not in JIS X 0208. Since this package is widely used, Lua\TeX-ja supports some of functions in the japanese-otf package, as an external package luatexja-otf.

4 Changing Internal Parameters

There are many internal parameters in Lua\TeX-ja. And due to the behavior of Lua\TeX, most of them are not stored as internal register of \TeX, but as an original storage system in Lua\TeX-ja. Hence, to assign or acquire those parameters, you have to use commands \ltjsetparameter and \ltjgetparameter.

4.1 Range of JAchars

Lua\TeX-ja divides the Unicode codespace U+0080–U+10FFFF into character ranges, numbered 1 to 217. The grouping can be (globally) customized by \ltjdefcharrange. The next line adds whole characters in Supplementary Ideographic Plane and the character “漢” to the character range 100.
\ltjdefcharrange{100}"20000-"2FFFF,‘漢"

A character can belong to only one character range. For example, whole SIP belong to the range 4 in the default setting of Lua\TeX-ja, and if one executes the above line, then SIP will belong to the range 100 and be removed from the range 4.

The distinction between ALchar and JAchar is performed by character ranges. This can be edited by setting the jacharrange parameter. For example, the code below is just the default setting of Lua\TeX-ja, and it sets
• a character which belongs character ranges 1, 4, 5, and 8 is ALchar,
• a character which belongs character ranges 2, 3, 6, and 7 is JAchar.
\ltjsetparameter{jacharrange={-1, +2, +3, -4, -5, +6, +7, -8}}

The argument to jacharrange parameter is a list of non-zero integer. Negative integer $-n$ in the list means that “each character in the range $n$ is an ALchar”, and positive integer $+n$ means that “... is a JAchar”.

Note that characters U+0000–U+007F are always treated as an ALchar (this cannot be customized).
Table 2. Unicode blocks in predefined character range 3.

<table>
<thead>
<tr>
<th>Code</th>
<th>Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+2000–U+206F</td>
<td>General Punctuation</td>
<td>Superscripts and Subscripts</td>
</tr>
<tr>
<td>U+20A0–U+20CF</td>
<td>Currency Symbols</td>
<td>Comb. Diacritical Marks for Symbols</td>
</tr>
<tr>
<td>U+2100–U+214F</td>
<td>Letterlike Symbols</td>
<td>Number Forms</td>
</tr>
<tr>
<td>U+2190–U+21FF</td>
<td>Arrows</td>
<td>Mathematical Operators</td>
</tr>
<tr>
<td>U+2300–U+23FF</td>
<td>Miscellaneous Technical</td>
<td>Control Pictures</td>
</tr>
<tr>
<td>U+2500–U+25FF</td>
<td>Geometric Shapes</td>
<td>Miscellaneous Symbols</td>
</tr>
<tr>
<td>U+2700–U+27FF</td>
<td>Dingbats</td>
<td>Supplemental Arrows-B</td>
</tr>
</tbody>
</table>

■ Default character ranges  LuaTX-ja predefines eight character ranges for convenience. They are determined from the following data:

- Blocks in Unicode 11.0.
- The Adobe-Japan1-UCS2 mapping between a CID Adobe-Japan1-6 and Unicode.
- The PXbase bundle for upTeX by Takayuki Yato.

Now we describe these eight ranges. The superscript “J” or “A” after the number shows whether each character in the range is treated as JAchars or not by default. These settings are similar to the prefercjk settings defined in PXbase bundle. Any characters equal to or above U+0080 which does not belong to these eight ranges belongs to the character range 217.

Range 8J The intersection of the upper half of ISO 8859-1 (Latin-1 Supplement) and JIS X 0208 (a basic character set for Japanese). This character range consists of the following characters:

- § (U+00A7, Section Sign)
- ’ (U+00B4, Spacing acute)
- ¨ (U+00A8, Diaeresis)
- ¶ (U+00B6, Paragraph sign)
- “ (U+00B0, Degree sign)
- × (U+00D7, Multiplication sign)
- ′ (U+00B4, Spacing acute)
- ÷ (U+00F7, Division Sign)

Range 1A Latin characters that some of them are included in Adobe-Japan1-6. This range consists of the following Unicode ranges, except characters in the range 8 above:

- 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

Range 2J Greek and Cyrillic letters. JIS X 0208 (hence most of Japanese fonts) has some of these characters.

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

Range 3A Punctuations and Miscellaneous symbols. The block list is indicated in Table 2.

Range 4A Characters usually not in Japanese fonts. This range consists of almost all Unicode blocks which are not in other predefined ranges. Hence, instead of showing the block list, we put the definition of this range itself:

\[
\texttt{\textbackslash t/jdefcharrange(4)(\%}
\texttt{ "500\textasciitilde10FF, "1200\textasciitilde1DFF, "2440\textasciitilde245F, "27C0\textasciitilde28FF, "2A00\textasciitilde2AFF,}
\texttt{ "2C00\textasciitilde2FF, "4DC0\textasciitilde4DFF, "A4D0\textasciitildeA95F, "9800\textasciitildeABFF,}
\texttt{ "E000\textasciitildeF8FF,}
\texttt{ "F800\textasciitildeFEOF, "FE20\textasciitildeFE2F, "FE70\textasciitildeFEFF, "10000\textasciitilde1AFFF,}
\texttt{ "1B170\textasciitilde1F0FF,}
\texttt{ "1F300\textasciitilde1FFFF}\]
\texttt{\% non-Japanese}

13
Table 3. Unicode blocks in predefined character range 6.

<table>
<thead>
<tr>
<th>Range</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Enclosed Alphanumerics</td>
</tr>
<tr>
<td></td>
<td>U+2460–U+24FF</td>
</tr>
<tr>
<td></td>
<td>CJK Symbols and Punctuation</td>
</tr>
<tr>
<td></td>
<td>U+3000–U+303F</td>
</tr>
<tr>
<td></td>
<td>Katakana</td>
</tr>
<tr>
<td></td>
<td>U+30A0–U+30FF</td>
</tr>
<tr>
<td></td>
<td>Katakana Phonetic Extensions</td>
</tr>
<tr>
<td></td>
<td>U+31F0–U+31FF</td>
</tr>
<tr>
<td></td>
<td>CJK Unified Ideographs</td>
</tr>
<tr>
<td></td>
<td>U+4E00–U+9FFF</td>
</tr>
<tr>
<td></td>
<td>CJK Compatibility Forms</td>
</tr>
<tr>
<td></td>
<td>U+FE10–U+FE1F</td>
</tr>
<tr>
<td></td>
<td>CJK Compatibility Ideographs</td>
</tr>
<tr>
<td></td>
<td>U+F900–U+FAFF</td>
</tr>
<tr>
<td></td>
<td>Vertical Forms</td>
</tr>
<tr>
<td></td>
<td>U+FE30–U+FE4F</td>
</tr>
<tr>
<td></td>
<td>CJK Compatibility Forms</td>
</tr>
<tr>
<td></td>
<td>U+FE50–U+FE6F</td>
</tr>
<tr>
<td></td>
<td>Small Form Variants</td>
</tr>
<tr>
<td></td>
<td>U+FF00–U+FFEF</td>
</tr>
<tr>
<td></td>
<td>Halfwidth and Fullwidth Forms</td>
</tr>
<tr>
<td></td>
<td>U+1B000–U+1B0FF</td>
</tr>
<tr>
<td></td>
<td>Kana Supplement</td>
</tr>
<tr>
<td></td>
<td>U+1B100–U+1B12F</td>
</tr>
<tr>
<td></td>
<td>Enclosed Alphanumeric Supp.</td>
</tr>
<tr>
<td></td>
<td>U+1F200–U+1F2FF</td>
</tr>
<tr>
<td></td>
<td>Enclosed Ideographic Supp.</td>
</tr>
<tr>
<td></td>
<td>U+E0100–U+E01EF</td>
</tr>
<tr>
<td></td>
<td>Variation Selectors Supp.</td>
</tr>
</tbody>
</table>

Table 4. Unicode blocks in predefined character range 7.

<table>
<thead>
<tr>
<th>Range</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Hangul Jamo</td>
</tr>
<tr>
<td></td>
<td>U+1100–U+11FF</td>
</tr>
<tr>
<td></td>
<td>Ideographic Description Characters</td>
</tr>
<tr>
<td></td>
<td>U+2F00–U+2FFC</td>
</tr>
<tr>
<td></td>
<td>CJK Strokes</td>
</tr>
<tr>
<td></td>
<td>U+31C0–U+31EF</td>
</tr>
<tr>
<td></td>
<td>Yi Radicals</td>
</tr>
<tr>
<td></td>
<td>U+4A90–U+4AFF</td>
</tr>
<tr>
<td></td>
<td>Hanlig Syllables</td>
</tr>
<tr>
<td></td>
<td>U+AC00–U+D7AF</td>
</tr>
<tr>
<td></td>
<td>Y Radicals</td>
</tr>
<tr>
<td></td>
<td>U+A960–U+A97F</td>
</tr>
<tr>
<td></td>
<td>Hangul Jamo Extended-A</td>
</tr>
<tr>
<td></td>
<td>U+D7B0–U+D7FF</td>
</tr>
<tr>
<td></td>
<td>Hangul Jamo Extended-B</td>
</tr>
</tbody>
</table>

Range 5
Surrogates and Supplementary Private Use Areas.

Range 6
Characters used in Japanese. The block list is indicated in Table 3.

Range 7
Characters used in CJK languages, but not included in Adobe-Japan1-6. The block list is indicated in Table 4.

Notes on U+0080–U+00FF
You should treat characters in textttU+0080–U+00FF as ALchar, when you use traditional 8-bit fonts, such as the marvosym package.

For example, \frowny which is provided by the marvosym package has the same codepoint as § (U+00A7). Hence, as previous versions of LuaTeX-ja, if these characters are treated as JAchars, then \frowny produces “ § ” (in a Japanese font).

To avoid such situations, the default setting of LuaTeX-ja is changed in version 20150906.0 so that all characters U+0080–U+00FF are treated as ALchar.

If you want to output a character as ALchar and JAchar regardless the range setting, you can use \ltjalchar and \ltjjachar respectively, as the following example.

```
\textfamily\large % default, ALchar, JAchar
\frowny, \ltjalchar\frowny, \ltjjachar\frowny\  % default: ALchar
\alpha, \ltjalchar\alpha, \ltjjachar\alpha\  % default: JAchar
```

4.2 kanjiskip and xkanjiskip

JAgue is divided into the following three categories:

- Glues/kerns specified in JFM. If \inhibitglue is issued around a JAchar, this glue will not be inserted at the place.
- The default glue which inserted between two JAchars (kanjiskip).
- The default glue which inserted between a JAchar and an ALchar (xkanjiskip).

The value (a skip) of kanjiskip or xkanjiskip can be changed as the following. Note that only their values at the end of a paragraph or a hbox are adopted in the whole paragraph or the whole hbox.

```
\ltjsetparameter{kanjiskip={0pt plus 0.4pt minus 0.4pt},
                xkanjiskip={0.25\zw plus 1pt minus 1pt}}
```
Here \zw is an internal dimension which stores fullwidth of the current Japanese font. This \zw can be used as the unit \zw in \texttt{p\TeX}.

The value of these parameter can be get by \texttt{\ljgetparameter}. Note that the result by \texttt{\ljgetparameter} is not the internal quantities, but a string (hence \texttt{\the} cannot be prefixed).

\begin{verbatim}
kanjiskip: \ljgetparameter{kanjiskip},
\texttt{xkanjiskip: \ljgetparameter{xkanjiskip}}
\end{verbatim}

\begin{verbatim}
kanjiskip: 0.0pt plus 0.4pt minus 0.5pt,
\texttt{xkanjiskip: 2.40555pt plus 1.0pt minus 1.0pt}
\end{verbatim}

It may occur that JFM contains the data of "ideal width of kanjiskip" and/or "ideal width of xkanjiskip". To use these data from JFM, set the value of \texttt{kanjiskip} or \texttt{xkanjiskip} to \texttt{\maxdimen} (these "ideal width" cannot be retrieved by \texttt{\ljgetparameter}).

4.3 Insertion setting of \texttt{xkanjiskip}

It is not desirable that \texttt{xkanjiskip} is inserted into every boundary between \texttt{JAchars} and \texttt{ALchars}. For example, \texttt{xkanjiskip} should not be inserted after opening parenthesis (e.g., compare “(あ)” and “(あ)”). \texttt{Lua\TeX-ja} can control whether \texttt{xkanjiskip} can be inserted before/after a character, by changing \texttt{jaxspmode} for \texttt{JAchars} and \texttt{alxspmode} parameters \texttt{ALchars} respectively.

\begin{verbatim}
\texttt{\ljsetparameter{jaxspmode={`あ,preonly},
\texttt{alxspmode=`!,postonly}}}
\end{verbatim}

One can use also numbers to specify these two parameters (see Subsection 8.1).

If you want to enable/disable all insertions of \texttt{kanjiskip} and \texttt{xkanjiskip}, set \texttt{autospacing} and \texttt{autoxspacing} parameters to \texttt{true}/false, respectively.

4.4 Shifting the baseline

To make a match between a Japanese font and an alphabetic font, sometimes shifting of the baseline of one of the pair is needed. In \texttt{p\TeX}, this is achieved by setting \texttt{\ybaselineshift} (or \texttt{\tbaselineshift}) to a non-zero length (the baseline of \texttt{ALchar} is shifted below). However, for documents whose main language is not Japanese, it is good to shift the baseline of Japanese fonts, but not that of alphabetic fonts. Because of this, \texttt{Lua\TeX-ja} can independently set the shifting amount of the baseline of alphabetic fonts and that of Japanese fonts.

Here the horizontal line in the below example is the baseline of a line.

\begin{verbatim}
\vrule width 150pt height 0.2pt depth 0.2pt \hskip-120pt
\end{verbatim}

\begin{verbatim}
\texttt{\ljsetparameter{yalbaselineshift=0pt,
yjbaselineshift=0pt}}
\end{verbatim}

There is an interesting side-effect: characters in different size can be vertically aligned center in a line, by setting two parameters appropriately. The following is an example (beware the value is not well tuned):
Note that setting positive \texttt{yalbaselineshift} or \texttt{talbaselineshift} parameters does not increase the depth of one-letter syllable $p$ of Alchar, if its left-protrusion (\texttt{lrcode}) and right-protrusion (\texttt{rrcode}) are both non-zero. This is because

- These two parameters are implemented by setting \texttt{yoffset} field of a glyph node, and this does not increase the depth of the glyph.
- To cope with the above situation, Lua\TeX\-ja automatically supplies a rule in every syllable.
- However, we cannot use this “supplying a rule” method if a syllable comprises just one letter whose \texttt{lrcode} and \texttt{rrcode} are both non-zero.

This problem does not apply for \texttt{yjabaselineshift} nor \texttt{tjabaselineshift}, because a JAchar is encapsulated by a horizontal box if needed.

### 4.5 kinsoku parameters and OpenType features

Among parameters which related to Japanese word-wrapping process (kinsoku shori), \texttt{jaxspmode}, \texttt{alxspmode}, \texttt{prebreakpenalty}, \texttt{postbreakpenalty} and \texttt{kcatcode} are stored by each character codes.

OpenType font features are ignored in these parameters. For example, a fullwidth katakana “ア” on line 10 in the below input is replaced to its halfwidth variant “ぁ”, by \texttt{hwid} feature. However, the penalty inserted after it is 10 which is the \texttt{postbreakpenalty} of “ア”, not 20.

\begin{verbatim}
\ltjsetparameter{postbreakpenalty={`ア', 10}}
\ltjsetparameter{postbreakpenalty={`ぁ', 20}}
\newcommand\showpostpena[1]{% 
\leavevmode\setbox0=\hbox{#1\hbox{}}}\% 
\unhbox0\setbox0=\lastbox\the\lastpenalty\% 
\showpostpena{ア}, \showpostpena{ぁ}, \showpostpena{ア}, \showpostpena{ぁ} \{\addjfontfeatures{CharacterWidth=Half}\showpostpena{ア}}
\end{verbatim}

\begin{verbatim}
ア 10, {	extasciitilde} 10, {	extasciitilde} 20, {	extasciitilde} 10
\end{verbatim}
Part II

Reference

5 \catcode in Lua\TeX-ja

5.1 Preliminaries: \kcatcode in p\TeX and up\TeX

In p\TeX and up\TeX, the value of \catcode determines whether a Japanese character can be used in a control word. For the detail, see Table 5.

\catcode can be set by a row of JIS X 0208 in p\TeX, and generally by a Unicode block\footnote{up\TeX divides U+FF00–U+FFEF (Halfwidth and Fullwidth Forms) into three subblocks, and \catcode can be set by a subblock.} in up\TeX. So characters which can be used in a control word slightly differ between p\TeX and up\TeX.

5.2 Case of Lua\TeX-ja

The role of \catcode in p\TeX and up\TeX can be divided into the following four kinds, and Lua\TeX-ja can control these four kinds separately:

- Distinction between JAchar or ALchar is controlled by the character range, see Subsection 4.1.
- Whether the character can be used in a control word is controlled by setting \catcode to 11 (enabled) or 12 (disabled), as usual.
- Whether \texttt{icharwidowpenalty} can be inserted before the character is controlled by the lowermost bit of the \catcode parameter.
- Linebreak after a JAchar does not produce a space.

Default setting of \catcode of Unicode characters are located in

plain Lua\TeX: \texttt{luatex-unicode-letters.tex}, which is based on \texttt{unicode-letters.tex} (for X\TeX).

Lua\TeX now included in \TeX kernel as \texttt{unicode-letters.def}.

However, the default setting of \catcode differs between X\TeX and Lua\TeX, by the following reasons:

- (plain format) \texttt{luatex-unicode-letters.tex} is based on old \texttt{unicode-letters.tex}.
- The latter half of \texttt{unicode-letters.tex} and \texttt{unicode-letters.def} sets \catcode of several characters to 11, via setting \texttt{\XeTeXcharclass}. However, this latter half does not exist (plain case), or not executed (\TeX case) in Lua\TeX.

In other words,

plain Lua\TeX Kanji nor kana characters cannot be used in a control word, in the default setting of plain Lua\TeX.

Lua\TeX In recent (2015-10-01 or later) Lua\TeX, Kanji and kana characters in a control word is supported (these \catcode are 11), but not fullwidth alphanumerics and several other characters.

This would be inconvenient for p\TeX users to shifting to Lua\TeX-ja, since several control words containing Kanji or other fullwidth characters, such as \texttt{\year} or \texttt{\year} are used in p\TeX. Hence, Lua\TeX-ja have a counterpart of \texttt{unicode-letters.tex} for Lua\TeX, to match the \catcode setting with that of X\TeX.
5.3 Non-kanji characters in a control word

Because the engine differ, so non-kanji JIS X 0208 characters which can be used in a control word differ in \TeX, in up\TeX, and in Lua\TeX-ja. Table 6 shows the difference. Except for four characters "・", "゛", "゜", "゠", Lua\TeX-ja admits more characters in a control word than up\TeX.

Difference becomes larger, if we consider non-kanji JIS X 0213 characters. For the detail, see https://github.com/h-kitagawa/kct.

6 Directions

Lua\TeX supports four Ω-style directions: TLT, TRT, RTT and LTL. However, neither directions are not well-suited for typesetting Japanese vertically, hence we implemented vertical writing by rotating TLT-box by 90 degrees.

Lua\TeX-ja supports four directions, as shown in Table 7. The second column (yoko direction) is just horizontal writing, and the third column (tate direction) is vertical writing. The fourth column (dtou direction) is actually a hidden feature of \TeX. We implemented this for debugging purpose. The fifth column (utod direction) corresponds the "tate (math) direction" of \TeX.

Directions can be changed by \yoko, \tate, \dtou, \utod, only when the current list is null. These commands cannot be executed in unrestricted horizontal modes, nor math modes. The direction of a math formula is changed to utod, when the direction outside the math formula is tate (vertical writing).

6.1 Boxes in different direction

As in \TeX, one can use boxes of different direction in one document. The below is an example.
Table 7. Directions supported by LuaTeX-ja

<table>
<thead>
<tr>
<th>Commands</th>
<th>horizontal (yoko direction)</th>
<th>vertical (tate direction)</th>
<th>dtou direction</th>
<th>utod direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beginning of the page</td>
<td>\yoko</td>
<td>\tate</td>
<td>\dtou</td>
<td>\utod</td>
</tr>
<tr>
<td>Beginning of the line</td>
<td>Left</td>
<td>Top</td>
<td>Bottom</td>
<td>Top</td>
</tr>
<tr>
<td>Used Japanese font</td>
<td>horizontal (\jfont)</td>
<td>vertical (\tfont)</td>
<td>horizontal (90° rotated)</td>
<td></td>
</tr>
</tbody>
</table>

Example

\begin{center}
\begin{tabular}{c}
\begin{tikzpicture}
\draw[->](0,0)--(0,1); \\
\draw[->](0,0)--(1,0); \\
\draw[->](0,0)--(0,-1); \\
\draw[->](0,0)--(-1,0);
\end{tikzpicture}
\end{tabular}
\end{center}

(Notation used in \(\Omega\))

TLT RTR, RTT LBL RTR

Table 8 shows how a box is arranged when the direction inside the box and that outside the box differ.

\textbf{\texttt{\textbackslash wd and direction} } In \LaTeX, \texttt{\textbackslash wd}, \texttt{\textbackslash ht}, \texttt{\textbackslash dp} means the dimensions of a box register \textit{with respect to the current direction}. This means that the value of \texttt{\textbackslash wd0} etc. might differ when the current direction is different, even if \texttt{\textbackslash box0} stores the same box. However, this no longer applies in LuaTeX-ja.

\begin{verbatim}
\setbox0=\hbox to 20pt{foo} \the\wd0, \hbox{\tate\vrule\the\wd0}
\setbox0=\hbox to 20pt{foo} \the\wd0=100pt \the\wd0, \hbox{\tate \the\wd0}
\end{verbatim}

To access box dimensions \textit{with respect to current direction}, one have to use the following commands instead of \texttt{\textbackslash wd} \texttt{\textbackslash wtc}.

\texttt{\ltjgetwd(num), \ltjgetht(num), \ltjgetdp(num)}

These commands return an \textit{internal dimension} of \texttt{\box(num)} with respect to the current direction. One can use these in \texttt{\dimexpr} primitive, as the followings.

\texttt{\dimexpr 2*\ltjgetwd42-3pt\relax, \the\ltjgetwd1701}

The following is an example.

\begin{verbatim}
\parindent0pt
\setbox0=\hbox{\yokoよこぐみ} \setbox0=\hbox{\copy32767}
\vbox{\hsize=20mm\yoko YOKO \the\ltjgetwd32767, \the\ltjgetht32767, \the\ltjgetdp32767.}
\vbox{\hsize=20mm\tate TATE \the\ltjgetwd32767, \the\ltjgetht32767, \the\ltjgetdp32767.}
\vbox{\hsize=20mm\dtou DTOU \the\ltjgetwd32767, \the\ltjgetht32767, \the\ltjgetdp32767.}
\end{verbatim}

YOKO 38.48pt, 38.48pt, 1.15pt.

よこぐみ 38.48pt, 38.48pt, 1.15pt.

TATE 38.48pt, 38.48pt, 1.15pt.

DTOU 38.48pt, 38.48pt, 1.15pt.
Table 8. Boxes in different direction

<table>
<thead>
<tr>
<th>typeset in yoko direction</th>
<th>typeset in tate or utod direction</th>
<th>typeset in dtou direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_y = h_y + d_y )</td>
<td>( W_T = h_y + d_y )</td>
<td>( W_D = h_y + d_y )</td>
</tr>
<tr>
<td>( H_y = w_T )</td>
<td>( H_T = w_T )</td>
<td>( H_D = w_T )</td>
</tr>
<tr>
<td>( D_y = 0 \text{ pt} )</td>
<td>( D_T = w_T/2 )</td>
<td>( D_T = h_D )</td>
</tr>
</tbody>
</table>

\ltjsetwd(num)=(dimen), \ltjsetht(num)=(dimen), \ltjsetdp(num)=(dimen)

These commands set the dimension of \\texttt{\box{num}}. One does not need to group the argument \( \texttt{\langle num\rangle} \); four calls of \\ltjsetwd below have the same meaning.

\ltjsetwd42 20pt, \ltjsetwd42=20pt, \ltjsetwd=42 20pt, \ltjsetwd=42=20pt

6.2 Getting current direction

The \texttt{direction} parameter returns the current direction, and the \texttt{boxdir} parameter (with the argument \( \langle num\rangle \)) returns the direction of a box register \\texttt{\box{num}}. The returned value of these parameters are a \texttt{string}:

<table>
<thead>
<tr>
<th>Direction</th>
<th>yoko</th>
<th>tate</th>
<th>dtou</th>
<th>utod</th>
<th>(empty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned value</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>

\leavevmode\def\DIR{\ltjgetparameter{direction}}
\hbox{\yoko \DIR}, \hbox{\tate\DIR}, \hbox{\dtou\DIR}, \hbox{\utod\DIR}, \hbox{\tate\tate math: \DIR$\$}
\setbox2=\hbox{\tate}\ltjgetparameter{boxdir}{2}

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
Table 9. Differences between horizontal JFMs shipped with LuaTEX-ja

<table>
<thead>
<tr>
<th>JFM Name</th>
<th>Japan Language</th>
<th>Latin Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>jfm-ujis.lua</td>
<td>ある日モモちゃんのがお使いで迷惑子になって泣きました。</td>
<td>ちょっと!!頼 らず つ</td>
</tr>
<tr>
<td>jfm-jis.lua</td>
<td>ある日モモちゃんのがお使いで迷惑子になって泣きました。</td>
<td>ちょっと!!頼 らず つ</td>
</tr>
<tr>
<td>jfm-min.lua</td>
<td>ある日モモちゃんのがお使いで迷惑子になって泣きました。</td>
<td>ちょっと!!頼 らず つ</td>
</tr>
</tbody>
</table>

(Blue: jfm-ujis.lua, Black: jfm-jis.lua, Red: jfm-min.lua)

6.3 Overridden box primitives

To cope with multiple directions, the following primitives are overridden by LuaTEX-ja, using \protected\def.

\unhbox⟨num⟩, \unvbox⟨num⟩, \unhcopy⟨num⟩, \unvcopy⟨num⟩
\vadjust{⟨material⟩}
\insert⟨number⟩{(⟨material⟩)}
\lastbox
\raise⟨dimen⟩⟨box⟩, \lower⟨dimen⟩⟨box⟩ etc., \vcenter
\vcenter

7 Font Metric and Japanese Font

7.1 \jfont

To load a font as a Japanese font (for horizontal direction), you must use the \jfont instead of \font, while \jfont admits the same syntax used in \font. LuaTEX-ja automatically loads luaotfload package, so TrueType/OpenType fonts with features can be used for Japanese fonts:

```
\jfont\tradgt={file:KozMinPr6N-Regular.otf:script=latn;\%
+trad;-kern;jfm=ujis} at 14pt
\tradgt 当／體／醫／區
```

Note that the defined control sequence (\tradgt in the example above) using \jfont is not a font_def token, but a macro. Hence the input like \fontname\tradgt causes an error. We denote control sequences which are defined in \jfont by ⟨jfont_cs⟩.

JFM A JFM has measurements of characters and glues/kerns that are automatically inserted for Japanese typesetting. The structure of JFM will be described in the next subsection. At the calling of \jfont, you must specify which JFM will be used for this font by the following keys:

\jfm=(name)

Specify the name of (horizontal) JFM. If specified JFM has not been loaded, LuaTEX-ja search and load a file named jfm–(name).lua.

The following horizontal JFMs are shipped with LuaTEX-ja:

\jfm-ujis.lua A standard horizontal JFM in LuaTEX-ja. This JFM is based on upnmlminr-h.tfm, a metric for UTF/OTF package that is used in upTEX. When you use the luatexja-otf package, you should use this JFM.
\begin{figure}[ht]
\centering
\begin{multicols}{2}
\begin{itemize}
\item \texttt{\newcommand\test{\vrule \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}} \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}
\end{itemize}
\end{multicols}
\end{figure}

\begin{figure}[ht]
\centering
\begin{multicols}{2}
\begin{itemize}
\item \texttt{\newcommand\test{\vrule \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}} \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}
\end{itemize}
\end{multicols}
\end{figure}

\begin{figure}[ht]
\centering
\begin{multicols}{2}
\begin{itemize}
\item \texttt{\newcommand\test{\vrule \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}} \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}
\end{itemize}
\end{multicols}
\end{figure}

\begin{figure}[ht]
\centering
\begin{multicols}{2}
\begin{itemize}
\item \texttt{\newcommand\test{\vrule \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}} \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}
\end{itemize}
\end{multicols}
\end{figure}

\begin{figure}[ht]
\centering
\begin{multicols}{2}
\begin{itemize}
\item \texttt{\newcommand\test{\vrule \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}} \begin{multicols}{2} \jfont\test \jfont\test \jfont\test \jfont\test \end{multicols}}
\end{itemize}
\end{multicols}
\end{figure}

\begin{quote}
\textbf{jfm-jis.lua} A counterpart for \texttt{jis.tfm}, "JIS font metric" which is widely used in \LaTeX. A major difference between \texttt{jfm-ujis.lua} and this \texttt{jfm-jis.lua} is that most characters under \texttt{jfm-ujis.lua} are square-shaped, while that under \texttt{jfm-jis.lua} are horizontal rectangles.

\textbf{jfm-min.lua} A counterpart for \texttt{min10.tfm}, which is one of the default Japanese font metric shipped with \TeX.

The difference among these three JFMs is shown in Table 9.
\end{quote}

\begin{itemize}
\item \textbf{Using kerning information in a font} Some fonts have information for inter-glyph spacing. \LaTeX\-ja 20140324.0 or later treats kerning spaces like an italic correction; any glue and/or kern from the JFM and a kerning space can coexist. See Figure 2 for detail.

Note that in \texttt{\setmainjfont} etc. which are provided by \texttt{luatexja-fontspec} package, kerning option is set off (\texttt{Kerning=Off}) by default, because of the compatibility with previous versions of \LaTeX\-ja.

\item \textbf{extend and slant} The following setting can be specified as OpenType font features:

\begin{verbatim}
extend=(extend) expand the font horizontally by (extend).
\end{verbatim}
\end{itemize}
slant=⟨slant⟩ slant the font.

Note that LuaTeX-ja doesn’t adjust JFMs by these extend and slant settings; you have to write new JFMs on purpose. For example, the following example uses the standard JFM jfm-ujis.lua, hence letter-spacing and the width of italic correction are not correct:

\jfont\E=KozMinPr6N-Regular:extend=1.5;jfm=ujis;-kern
\E あいうえお
\jfont\S=KozMinPr6N-Regular:slant=1;jfm=ujis;-kern
\S あいう\ABC

\ltjksp  kanjiskip_natural, kanjiskip_stretch, kanjiskip_shrink keys (Page ??) makes that LuaTeX-ja inserts not only a glue which is specified by a JFM, and also the natural width/stretch part/shrink part of \kanjiskip.

This functionality is disabled by -ltjksp specification.

\leavemode
\ltjsetparameter{kanjiskip=Opt plus 3\zw}
\vrule\hbox to 15\zw{あ「い」う, えお}\vrule
\jfont\G=file:KozMinPr6N-Regular.otf\%
\G\leavemode\%
\vrule\hbox to 15\zw{あ「い」う, えお}\vrule

7.2 $\texttt{\textbackslash tfont}$

$\texttt{\textbackslash tfont}$ loads a font as a Japanese font for vertical direction. This command admits the same syntax used in $\texttt{\textbackslash font}$ and $\texttt{\textbackslash jfont}$. A font defined by $\texttt{\textbackslash tfont}$ differs the following points from that by $\texttt{\textbackslash jfont}$:

• OpenType Feature vrt2\(^8\) is automatically activated, unless vert and/or vrt2 are explicitly activated or deactivated (as the second line in the example below).

$\texttt{\textbackslash tfont}\S=file:KozMinPr6N-Regular.otf\%jfm=ujisv \% vrt2 is automatically activated
$\texttt{\textbackslash tfont}\T=file:KozMinPr6N-Regular.otf\%jfm=ujisv;-vert \% vert and vrt2 are not activated
$\texttt{\textbackslash tfont}\U=file:ipaexm.ttf\%jfm=ujisv
    \% vert is automatically activated, since this font does not have vrt2

• Sometimes vert and/or vrt2 are not activated while one specified activation of these feature. This is because the font does not define these features in current combination of script tag and language system identifier.

In this situation, LuaTeX-ja performs all replacements which is defined in vert feature for some scripts for some languages.

• Furthermore, a glyph is automatically rotated 90 degrees, if it is not replaced by vert feature for any script for any language, and if it is marked as ‘r” or “Tr” in UAX #50.

• One have to specify the name of vertical JFM in jfm=⟨name⟩. LuaTeX-ja ships following vertical JFMs:

jfm-ujisv.lua  A standard vertical JFM in LuaTeX-ja. This JFM is based on upnmlminv-v.tfm, a metric for UTF/OTF package that is used in upTeX.

jfm-tmin.lua  A counterpart for tmin10.tfm, which is one of the default Japanese font metric shipped with pTeX.

\(^8\)If the font does not define vrt2 feature, use vert instead.
7.3 Default Japanese fonts and JFMs

If following commands are defined at loading Lua\TeX-ja package, these change default Japanese fonts and JFMs for them:

\ltj@stdmcfont The default Japanese font for the mincho family.
\ltj@stdgtfont The default Japanese font for the gothic family.
\ltj@stdyokojfm The default JFM for horizontal direction.
\ltj@stdyokojfm The default JFM for vertical direction.

For example,
\begin{verbatim}
\def\ltj@stdmcfont{IPAMincho}
\def\ltj@stdgtfont{IPAGothic}
\end{verbatim}
makes that IPA Mincho and IPA Gothic will be used as default Japanese fonts, instead of IPAex Mincho and IPAex Gothic.

This feature is intended for classes which use special JFMs. It is recommended to use \texttt{\luatexja-preset} or \texttt{\luatexja-fontspec} package to select standard fonts in ordinary \LaTeX sources.

For compatibility with earlier versions, Lua\TeX-ja reads \texttt{luatexja.cfg} automatically if it is found by Lua\TeX. One should not overuse this \texttt{luatexja.cfg}; it will overwrite the definition of \ltj@stdmcfont and others.

7.4 Prefix \texttt{psft}

Besides "file:" and "name:" prefixes which are introduced in the \texttt{luaotfload} package, Lua\TeX-ja adds "psft:" prefix in \texttt{\jfont} (and \texttt{\font}), to specify a "name-only" Japanese font which will not be embedded to PDF. Note that these non-embedded fonts under current Lua\TeX has Identity-H encoding, and this violates the standard ISO32000-1:2008 ([10]).

OpenType font features, such as "+jp90", have no meaning in name-only fonts using "psft:" prefix, because we can’t expect what fonts are actually used by the PDF reader. Note that extend and slant settings (see above) are supported with \texttt{psft} prefix, because they are only simple linear transformations.

\textbf{cid key} The default font defined by using \texttt{psft}: prefix is for Japanese typesetting; it is Adobe-Japan1-7 CID-keyed font. One can specify \texttt{cid} key to use other CID-keyed non-embedded fonts for Chinese or Korean typesetting.

\begin{verbatim}
\jfont\testJ={psft:Ryumin-Light:cайл=Adobe-Japan1-7;jfm=jis} % Japanese
\jfont\testD={psft:Ryumin-Light:jfm=jis} % default value is Adobe-Japan1-7
\jfont\testC={psft:AdobeMingStd-Light:cайл=Adobe-CNS1-7;jfm=jis} % Traditional Chinese
\jfont\testR={psft:SimSun:cайл=Adobe-GB1-5;jfm=jis} % Simplified Chinese
\jfont\testK={psft:Batang:cайл=Adobe-Korea1-2;jfm=jis} % Korean
\jfont\testKR={psft:SourceHanSerifAKR9:cайл=Adobe-KR-9;jfm=jis} % Korean
\end{verbatim}

Note that the code above specifies \texttt{jfm-jis.lua}, which is for Japanese fonts, as JFM for Chinese and Korean fonts.

At present, Lua\TeX-ja supports only 5 values written in the sample code above. Specifying other values, e.g.,
\begin{verbatim}
\jfont\test={psft:Ryumin-Light:cайл=Adobe-Japan2;jfm=jis}
\end{verbatim}
produces the following error:
\begin{verbatim}
! Package luatexja Error: bad cid key `Adobe-Japan2'.
\end{verbatim}
\begin{verbatim}
See the luatexja package documentation for explanation.
Type H <return> for immediate help.
\end{verbatim}

\footnote{This is because commands has \texttt{\_} in their names.}

24
I couldn't find any non-embedded font information for the CID 'Adobe-Japan2'. For now, I'll use 'Adobe-Japan1-6'. Please contact the LuaTeX-ja project team.

7.5 Structure of a JFM file

A JFM file is a Lua script which has only one function call:

\texttt{luatexja.jfont.define_jfm \{ ... \}}

Real data are stored in the table which indicated above by \{ ... \}. So, the rest of this subsection are devoted to describe the structure of this table. Note that all lengths in a JFM file are floating-point numbers in design-size unit.

\texttt{version=(version) (optional, default value is 1)}

The version JFM. Currently 1, 2, and, 3 are supported

\texttt{dir=(direction) (required)}

The direction of JFM. 'yoko' (horizontal) or 'tate' (vertical) are supported.

\texttt{zw=(length) (required)}

The amount of the length of the "full-width".

\texttt{zh=(length) (required)}

The amount of the "full-height" (height + depth).

\texttt{kanjiskip=\{natural\}, \{stretch\}, \{shrink\}} (optional)

This field specifies the "ideal" amount of \texttt{kanjiskip}. As noted in Subsection 4.2, if the parameter \texttt{kanjiskip} is \texttt{\maxdimen}, the value specified in this field is actually used (if this field is not specified in JFM, it is regarded as 0 pt). Note that \{stretch\} and \{shrink\} fields are in design-size unit too.

\texttt{xkanjiskip=\{natural\}, \{stretch\}, \{shrink\}} (optional)

Like the \texttt{kanjiskip} field, this field specifies the "ideal" amount of \texttt{xkanjiskip}.

■ Character classes

Besides from above fields, a JFM file have several sub-tables those indices are natural numbers. The table indexed by \(i \in \omega\) stores information of character class \(i\). At least, the character class 0 is always present, so each JFM file must have a sub-table whose index is [0]. Each sub-table (its numerical index is denoted by \(i\)) has the following fields:

\texttt{chars=\{character\}, ...} (required except character class 0)

This field is a list of characters which are in this character type \(i\). This field is optional if \(i = 0\), since all JAchar which do not belong any character classes other than 0 are in the character class 0 (hence, the character class 0 contains most of JAchars). In the list, character(s) can be specified in the following form:

- a Unicode code point
- the character itself (as a Lua string, like ‘あ’)
- a string like ‘あ*’ (the character followed by an asterisk)
- several “imaginary” characters (We will describe these later.)
Consider a Japanese character node which belongs to a character class whose the align field is 'middle'.

- The black rectangle is the imaginary body of the node. Its width, height, and depth are specified by JFM.
- Since the align field is 'middle', the 'real' glyph is centered horizontally (the green rectangle) first.
- Furthermore, the glyph is shifted according to values of fields left and down. The ultimate position of the real glyph is indicated by the red rectangle.

<table>
<thead>
<tr>
<th>Direction of JFM</th>
<th>'yoko' (horizontal)</th>
<th>'tate' (vertical)</th>
</tr>
</thead>
<tbody>
<tr>
<td>width field</td>
<td>the width of the &quot;real&quot; glyph</td>
<td></td>
</tr>
<tr>
<td>height field</td>
<td>the height of the &quot;real&quot; glyph</td>
<td></td>
</tr>
<tr>
<td>depth field</td>
<td>the depth of the &quot;real&quot; glyph</td>
<td></td>
</tr>
<tr>
<td>italic field</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 10. Default values of width field and other fields

In most cases, left and down fields are 0, while it is not uncommon that the align field is 'middle' or 'right'. For example, setting the align field to 'right' is practically needed when the current character class is the class for opening delimiters.'
full stop “.” and a fullwidth middle dot “・” is three-fourth of fullwidth, namely halfwidth from the ideographic full stop, and quarter-width from the fullwidth middle dot. In this case, we specify \langle ratio \rangle to \(0.25/(0.5 + 0.25) = 1/3\).

In case of glue, one can specify following additional keys in each \[ j \] subtable:

- priority=\langle priority \rangle An integer in \([-4, 3]\) (treated as 0 if omitted), or a pair of these integers \{\langle stretch \rangle, \langle shrink \rangle\} (version 2 or later). This is used only in line adjustment with priority by luatexja-adjust (see Subsection 11.3). Higher value means the glue is easy to stretch, and is also easy to shrink.

- kanjiskip\_natural=\langle num \rangle, kanjiskip\_stretch=\langle num \rangle, kanjiskip\_shrink=\langle num \rangle

These keys specifies the amount of the natural width of kanjiskip (the stretch/shrink part, respectively) which will be inserted in addition to the original JFM glue. Default values of them are all 0.

As an example, in jfm-ujis.lua, the standard JFM in horizontal writing, we have:

- Between an ordinal letter “あ” and an ideographic opening bracket, we have a glue whose natural part and shrink part are both half-width, while its stretch part is zero. However, this glue also can be stretched as much as the stretch part of kanjiskip times the value of kanjiskip\_stretch key (1 in this case).
- Between an ideographic closing brackets (the ideographic comma “，” is included) and an ordinal letter, we have the same glue. Again, this glue also can be stretched as much as the stretch part of kanjiskip times the value of kanjiskip\_stretch key (1 in this case).
- Between an ideographic opening bracket and an ordinal letter and between an ordinal letter and an ideographic closing bracket, we have a glue whose natural part and stretch part are both zero, while its shrink part as much as the shrink part of kanjiskip.

Hence we have the following result:

\begin{verbatim}
\leavevmode
\ltjsetparameter{kanjiskip=0pt plus 3\zw}
\vrule\hbox to 15\zw{あ「い」う,えお}\vrule
\vrule\hbox{あ「い」う,えお}\vrule\par
\ltjsetparameter{kanjiskip=0pt minus \zw}
\vrule\hbox to 6.5\zw{あ「い」う,えお}\vrule
\end{verbatim}

\begin{verbatim}
end\_stretch=(kern), end\_shrink=(kern) (optional, version 1 only)
end\_adjust={⟨kern⟩, ⟨kern⟩, ⟨kern⟩, ...} (optional, version 2 or later)
\end{verbatim}

■ Character to character classes We explain how the character class of a character is determined, using jfm-test.lua which contains the following:

\begin{verbatim}
[0] = {
    chars = { '漢' },
    align = 'left', left = 0.0, down = 0.0,
    width = 1.0, height = 0.88, depth = 0.12, italic=0.0,
},
[2000] = {
    chars = { '。', '・' },
    align = 'left', left = 0.0, down = 0.0,
    width = 0.5, height = 0.88, depth = 0.12, italic=0.0,
},
\end{verbatim}

Now consider the following input/output:

\begin{verbatim}
\jfont\a=file:KozMinPr6N-Regular.otf:jfm=test;+hwid
\setbox0\hbox{あ と 漢}
\the\ud0
\end{verbatim}

Now we look why the above source outputs 15 pt.
1. The character “ヒ” is converted to its half width form “ヒ” by \textit{hwid} feature.

2. According to the JFM, the character class of “ヒ” is 2000, hence its width is halfwidth.

3. The character class of “漢” is zero, hence its width is fullwidth.

4. Hence the width of \texttt{\hbox} equals to 15 pt.

This example shows that the character class of a character is generally determined after applying font features by \textit{luaotfload}.

However, if the class determined by the glyph after application of features is zero, Lua\TeX-ja adopts the class determined by the glyph before application of features. The following input is an example.

```
\jfont\a=file:KozMinPr6N-Regular.otf:jfm=test;+vert
\a 漢。\inhibitglue 漢
```

Here, the character class of the ideographic full stop “。” (U+3002) is determined as follows:

1. As the case of “ヒ”, the ideographic full stop “。” is converted to its vertical form “︒” (U+FE12) by \textit{vert} feature.

2. The character class of “︒”, according to the JFM is zero.

3. However, Lua\TeX-ja remembers that this “︒” is obtained from “。” by font features. The character class of “。” is non-zero value, namely, 2000.

4. Hence the ideographic full stop “。” in above belongs the character class 2000.

\section*{Imaginary characters}

As described before, you can specify several imaginary characters in \texttt{chars} field. The most of these characters are regarded as the characters of class 0 in \texttt{p\LaTeX}. As a result, Lua\TeX-ja can control typesetting finer than p\LaTeX. The following is the list of imaginary characters:

- \texttt{‘boxbdd’}
  The beginning/ending of a hbox, and the beginning of a noindented (i.e., began by \texttt{\noindent}) paragraph.

- \texttt{‘parbdd’}
  The beginning of an (indented) paragraph.

- \texttt{‘jcharbdd’}
  A boundary between \texttt{JChar} and anything else.

- \texttt{‘alchar’, ‘nox_alchar’}
  (version 3 or later) A boundary between \texttt{JChar} and \texttt{ALchar}.

- \texttt{‘glue’}
  (version 3 or later) A boundary between \texttt{JChar}, and, a glue or kern.

- -1 The left/right boundary of an inline math formula.

\section*{Porting JFM from p\LaTeX}

See Japanese version of this manual.

\section*{7.6 Math font family}

\LaTeX handles fonts in math formulas by 16 font families\textsuperscript{10}, and each family has three fonts: \texttt{\textfont}, \texttt{\scriptfont} and \texttt{\scriptscriptfont}.

Lua\TeX-ja’s handling of Japanese fonts in math formulas is similar; \texttt{Table 11} shows counterparts to \LaTeX’s primitives for math font families. There is no relation between the value of \texttt{\fam} and that of \texttt{\jfam}; with appropriate settings, you can set both \texttt{\fam} and \texttt{\jfam} to the same value. Here (\texttt{ifont_cs}) in the argument of \texttt{jatextfont} etc. is a control sequence which is defined by \texttt{\jfont}, i.e., a horizontal Japanese font.

\textsuperscript{10} Omega, Aleph, Lua\TeX and \texttt{e-\LaTeX} can handles 256 families, but an external package is needed to support this in plain \LaTeX and \TeX.\TeX.
Table 11. Commands for Japanese math fonts

<table>
<thead>
<tr>
<th>Japanese fonts</th>
<th>alphabetic fonts</th>
</tr>
</thead>
<tbody>
<tr>
<td>\jfam ∈ [0, 256)</td>
<td>\fam</td>
</tr>
<tr>
<td>jatextfont={\jfam}, {\jfont}cs</td>
<td>textfont{\fam}=(font_cs)</td>
</tr>
<tr>
<td>jascriptfont={\jfam}, {\jfont}cs</td>
<td>scriptfont{\fam}=(font_cs)</td>
</tr>
<tr>
<td>jascriptscriptfont={\jfam}, {\jfont}cs</td>
<td>\scriptscriptfont{\fam}=(font_cs)</td>
</tr>
</tbody>
</table>

7.7 Callbacks

LuaTeX-ja also has several callbacks. These callbacks can be accessed via \texttt{luatexbase.add_to_callback} function and so on, as other callbacks.

\texttt{luatexja.load_jfm} callback

With this callback you can overwrite JFMs. This callback is called when a new JFM is loaded.

\begin{verbatim}
: function (<table> jfm_info, <string> jfm_name)
: return <table> new_jfm_info
: end
\end{verbatim}

The argument jfm_info contains a table similar to the table in a JFM file, except this argument has chars field which contains character codes whose character class is not 0.

An example of this callback is the \texttt{ltjarticle} class, with forcefully assigning character class 0 to 'parbdi' in the JFM \texttt{jfm-min.lua}.

\texttt{luatexja.define_jfont} callback

This callback and the next callback form a pair, and you can assign characters which do not have fixed code points in Unicode to non-zero character classes. This \texttt{luatexja.define_font} callback is called just when new Japanese font is loaded.

\begin{verbatim}
: function (<table> jfont_info, <number> font_number)
: return <table> new_jfont_info
: end
\end{verbatim}

jfont_info has the following fields, \textit{which may not overwritten by a user}:

- \texttt{size} The font size specified at \texttt{\jfont} in scaled points (1 sp = \texttt{2^{−16} pt}).
- \texttt{zw, zh, kanjiskip, xkanjiskip} These are scaled value of those specified by the JFM, by the font size.
- \texttt{jfm} The internal number of the JFM.
- \texttt{var} The value of jfmvar key, which is specified at \texttt{\jfont}. The default value is the empty string.
- \texttt{chars} The mapping table from character codes to its character classes.
  The specification \texttt{[i].chars={\texttt{character}}, ...}} in the JFM will be stored in this field as \texttt{chars={\texttt{\{character\}=i, ...}}}.
- \texttt{char_type} For \(i \in \omega\), \texttt{char_type[\[i\]}} is information of characters whose class is \(i\), and has the following fields:
  - \texttt{width}, \texttt{height}, \texttt{depth}, \texttt{italic}, \texttt{down}, \texttt{left} are just scaled value of those specified by the JFM, by the font size.
  - \texttt{align} is a number which is determined from align field in the JFM:
    \begin{align*}
    \{0 & \text{'left' (default)} \\
    0.5 & \text{'middle'} \\
    1 & \text{'right'}
    \end{align*}

For \(i, j \in \omega, \texttt{char_type[i]}[\[j\]} stores a kern or a glue which will be inserted between character class \(i\) and class \(j\).
The returned table `new_jfont_info` also should include these fields, but you are free to add more fields (to use them in the `luatexja.find_char_class` callback). The `font_number` is a font number.

A good example of this and the next callbacks is the `luatexja-otf` package, supporting "AJ1-xxx" form for Adobe-Japan1 CID characters in a JFM. This callback doesn’t replace any code of LuaTeX-ja.

**luatexja.find_char_class callback**

This callback is called just when LuaTeX-ja is trying to determine which character class a character `chr_code` belongs. A function used in this callback should be in the following form:

```lua
function (number) char_class, (table) jfont_info, (number) chr_code)
  if char_class~=0 then return char_class
  else
    ....
  return (<number> new_char_class or 0)
end
end
```

The argument `char_class` is the result of LuaTeX-ja's default routine or previous function calls in this callback, hence this argument may not be 0. Moreover, the returned `new_char_class` should be as same as `char_class` when `char_class` is not 0, otherwise you will overwrite the LuaTeX-ja's default routine.

**luatexja.set_width callback**

This callback is called when LuaTeX-ja is trying to encapsulate a JAchar `glyph_node`, to adjust its dimension and position.

```lua
function (table) shift_info, (table) jfont_info, (table) char_type)
  return (table) new_shift_info
end
```

The argument `shift_info` and the returned `new_shift_info` have down and left fields, which are the amount of shifting down/left the character in a scaled point.

A good example is test/valign.lua. After loading this file, the vertical position of glyphs is automatically adjusted; the ratio (height : depth) of glyphs is adjusted to be that of letters in the character class 0. For example, suppose that

- The setting of the JFM: (height) = 88x, (depth) = 12x (the standard values of Japanese OpenType fonts);
- The value of the real font: (height) = 28y, (depth) = 5y (the standard values of Japanese TrueType fonts).

Then, the position of glyphs is shifted up by

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

## 8 Parameters

### 8.1 \ltjsetparameter

As described before, \ltjsetparameter and \ltjgetparameter are commands for accessing most parameters of LuaTeX-ja. One of the main reason that LuaTeX-ja didn’t adopted the syntax similar to that of pTeX (e.g., \prebreakpenalty = 10000) is the position of hpack_filter callback in the source of LuaTeX, see Section 12.

\ltjsetparameter and \ltjglobalsetparameter are commands for assigning parameters. These take one argument which is a (key)=(value) list. The difference between these two commands is the scope of assignment; \ltjsetparameter does a local assignment and \ltjglobalsetparameter does a global one. They also obey the value of \globaldefs, like other assignments.

The following is the list of parameters which can be specified by the \ltjsetparameter command. [\cs] indicates the counterpart in pTeX, and symbols beside each parameter has the following meaning:
• “＊” : values at the end of a paragraph or a hbox are adopted in the whole paragraph or the whole hbox.
• “†” : assignments are always global.

\jcharwidowpenalty=(penalty)* [\jcharwidowpenalty]
Penalty value for suppressing orphans. This penalty is inserted just after the last \jAchar which is not regarded as a (Japanese) punctuation mark.

\kcatcode=(\chr_code, (\natural number))*
An additional attributes which each character whose character code is \chr_code has. At version 20120506.0 or later, the lowermost bit of \natural_number indicates whether the character is considered as a punctuation mark (see the description of \jcharwidowpenalty above).

\prebreakpenalty=(\chr_code, (penalty))* [\prebreakpenalty]
Set a penalty which is inserted automatically before the character \chr_code, to prevent a line starts from this character. For example, a line cannot started with one of closing brackets “” ”, so Lua\TeX-ja sets \ltxjsetparameter{prebreakpenalty={`,10000}} by default.

\p\TeX has following restrictions on \prebreakpenalty and \postbreakpenalty, but they don’t exist in Lua\TeX-ja:
• Both \prebreakpenalty and \postbreakpenalty cannot be set for the same character.
• We can set \prebreakpenalty and \postbreakpenalty up to 256 characters.

\postbreakpenalty=(\chr_code, (penalty))* [\postbreakpenalty]
Set a penalty which is inserted automatically after the character \chr_code, to prevent a line ends with this character.

\jatextfont=(\jfam, (\jfont_cs))* [\textfont in \TeX]
\jascriptfont=(\jfam, (\jfont_cs))* [\scriptfont in \TeX]
\jascriptscriptfont=(\jfam, (\jfont_cs))* [\scriptscriptfont in \TeX]
\yjabaselineshift=(\dimen)
\yalbaselineshift=(\dimen) [\ybaselineshift]
\tjabaselineshift=(\dimen)
\talbaselineshift=(\dimen) [\tbaselineshift]
\jxspmode=(\chr_code, (\mode))*
Set whether inserting \xkanjiskip is allowed before/after a \jAchar whose character code is \chr_code. The followings are allowed for \mode:
0, inhibit Insertion of \xkanjiskip is inhibited before the character, nor after the character.
1, preonly Insertion of \xkanjiskip is allowed before the character, but not after.
2, postonly Insertion of \xkanjiskip is allowed after the character, but not before.
3, allow Insertion of \xkanjiskip is allowed both before the character and after the character. This is the default value.

This parameter is similar to the \inhibitxspcode primitive of \p\TeX, but not compatible with \inhibitxspcode.

\alxspmode=(\chr_code, (\mode))* [\xspcode]
Set whether inserting \xkanjiskip is allowed before/after a \ALchar whose character code is \chr_code. The followings are allowed for \mode:
0, inhibit Insertion of \texttt{xkanjiskip} is inhibited before the character, nor after the character.

1, preonly Insertion of \texttt{xkanjiskip} is allowed before the character, but not after.

2, postonly Insertion of \texttt{xkanjiskip} is allowed after the character, but not before.

3, allow Insertion of \texttt{xkanjiskip} is allowed before the character and after the character. This is the default value.

Note that parameters \texttt{jaxspmode} and \texttt{alxspmode} share a common table, hence these two parameters are synonyms of each other.

\begin{verbatim}
autospacing=(bool) [\autospacing]
autoxspacing=(bool) [\autoxspacing]
kanjiskip=(skip)* [\kanjiskip]

The default glue which inserted between two JAchars. Changing current Japanese font does not alter this parameter, as \texttt{pTX}.

If the natural width of this parameter is \texttt{\maxdimen}, Lua\TeX-ja uses the value which is specified in the JFM for current Japanese font (See Subsection 7.5).

xkanjiskip=(skip)* [\xkanjiskip]

The default glue which inserted between a JAchar and an ALchar. Changing current font does not alter this parameter, as \texttt{pTX}.

As \texttt{kanjiskip}, if the natural width of this parameter is \texttt{\maxdimen}, Lua\TeX-ja uses the value which is specified in the JFM for current Japanese font (See Subsection 7.5).

differentjfm=(mode)†

Specify how glues/kerns between two JAchars whose JFM (or size) are different. The allowed arguments are the followings:

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

The default value is paverage. ...

jacharrange=(ranges)\n
kansujichar={⟨digit⟩, ⟨chr_code⟩}* [\kansujichar]

direction=(dir) (always local)

Assigning to this parameter has the same effect as \texttt{\yoko} (if \texttt{⟨dir⟩} = 4), \texttt{\tate} (if \texttt{⟨dir⟩} = 3), \texttt{\dtou} (if \texttt{⟨dir⟩} = 1) or \texttt{\utod} (if \texttt{⟨dir⟩} = 11). If the argument \texttt{⟨dir⟩} is not one of 4, 3, 1 nor 11, the behavior of this assignment is undefined.

8.2 \texttt{\ltjgetparameter}

\texttt{\ltjgetparameter} is a control sequence for acquiring parameters. It always takes a parameter name as first argument.

\begin{verbatim}
\ltjgetparameter(differentjfm),
\ltjgetparameter(autospacing),
\ltjgetparameter(kanjiskip),
\ltjgetparameter(prebreakpenalty){`}).
\end{verbatim}

The return value of \texttt{\ltjgetparameter} is always a string, which is outputted by \texttt{tex.write()}. Hence any character other than space “ “ (U+0020) has the category code 12 (other), while the space has 10 (space).

• If first argument is one of the following, no additional argument is needed.

  jcharwidowpenalty, yjabaselineshift, yalbaselineshift, autospacing, autoxspacing, kanjiskip, xkanjiskip, differentjfm, direction

\end{verbatim}
Note that \ltjgetparameter{autospacing} and \ltjgetparameter{autoxspacing} returns 1 or 0, not true nor false.

- If first argument is one of the following, an additional argument—a character code, for example—is needed.
  - kcatcode, prebreakpenalty, postbreakpenalty, jaxspmode, alxspmode
  \ltjgetparameter{jaxspmode}{...} and \ltjgetparameter{alxspmode}{...} returns 0, 1, 2, or 3, instead of preonly etc.

- \ltjgetparameter{jacharrange}{⟨range⟩} returns 0 if "characters which belong to the character range ⟨range⟩ are J\textbf{A}char", 1 if "... are A\textbf{L}char". Although there is no character range −1, specifying −1 to ⟨range⟩ does not cause an error (returns 1).

- For an integer ⟨digit⟩ between 0 and 9, \ltjgetparameter{kansujichar}{⟨digit⟩} returns the character code of the result of \kansuji{⟨digit⟩}.

- \ltjgetparameter{adjustdir} returns an integer which represents the direction of the surrounding vertical list. As direction, the return value 1 means down-to-up direction, 3 means tate direction (vertical typesetting), and 4 means yoko direction (horizontal typesetting).

- For an integer ⟨reg_num⟩ between 0 and 65535, \ltjgetparameter{boxdim}{⟨reg_num⟩} returns the direction of \box{⟨reg_num⟩}. If this box register is void, the returned value is zero.

- The following parameter names cannot be specified in \ltjgetparameter.
  - jatextfont, jascriptfont, jascriptscriptfont, jacharrange

- \ltjgetparameter{chartorange}{⟨chr_code⟩} returns the range number which ⟨chr_code⟩ belongs to (although there is no parameter named "chartorange").
  If ⟨chr_code⟩ is between 0 and 127, this ⟨chr_code⟩ does not belong to any character range. In this case, \ltjgetparameter{chartorange}{⟨chr_code⟩} returns −1.
  Hence, one can know whether ⟨chr_code⟩ is J\textbf{A}char or not by the following:
  \ltjgetparameter{jacharrange}{\ltjgetparameter{chartorange}{⟨chr_code⟩}}
  \%
  0 if \textbf{J}\textbf{A}char, 1 if \textbf{A}\textbf{L}char

- Because the returned value is string, the following conditionals do not work if \kanjiskip (or \xkanjiskip) has the stretch part or the shrink part.
  \ifdim\ltjgetparameter{kanjiskip}\z@ \... \fi
  \ifdim\ltjgetparameter{xkanjiskip}\z@ \... \fi

  The correct way is using a temporary register.
  \@tempskipa=\ltjgetparameter{kanjiskip} \ifdim\@tempskipa\z@ \... \fi
  \@tempskipa=\ltjgetparameter{xkanjiskip} \ifdim\@tempskipa\z@ \... \fi

8.3 Alternative Commands to \ltjsetparameter

The basic method to set parameters of Lua\TeX-ja is to use \ltjsetparameter or \ltjglobalsetparameter. However, these commands are slow, because they parse a key-value list, so several alternative commands are used in Lua\TeX-ja. This subsection is not for general Lua\TeX-ja users.

### Setting \texttt{kanjiskip} or \texttt{xkanjiskip}

In \ltjclasses, every size-changing command such as \texttt{\Large} changes \texttt{\kanjiskip} and \texttt{\xkanjiskip}. But a simple implementation, as the code below, is slow since two key-value lists are parsed by \texttt{\ltjsetparameter}:
Hence, LuaTeX-ja defines more primitive commands, namely \texttt{\textbackslash ltj@setpar@global}, \texttt{\textbackslash ltjsetkanjiskip}, and \texttt{\textbackslash ltjsetxkanjiskip}. Here

\texttt{\textbackslash ltj@setpar@global\textbackslash ltjsetkanjiskip 10pt}

and \texttt{\textbackslash ltjsetparameter{kanjiskip=10pt}} has the same effect. The actual code of ltjsclasses is shown below:

\begin{verbatim}
\texttt{\ltj@setpar@global}\texttt{\ltjsetkanjiskip\z@ plus .1\zw minus .01\zw}\texttt{\@tempskipa=\ltjgetparameter{xkanjiskip}}\texttt{\ifdim\@tempskipa>\z@}\texttt{\if@slide}\texttt{\ltjsetxkanjiskip.1em}\texttt{\else}\texttt{\ltjsetxkanjiskip.25em plus .15em minus .06em}\texttt{\fi}\texttt{\fi}
\end{verbatim}

Note that using \texttt{\ltjsetkanjiskip} or \texttt{\ltjsetxkanjiskip} alone, that is, without executing \texttt{\ltj@setpar@global} in advance, is not supported.

9 Other Commands for plain \LaTeX and \LaTeXe

9.1 Commands for compatibility with p\LaTeX

The following commands are implemented for compatibility with p\LaTeX. Note that the former five commands don’t support JIS X 0213, but only JIS X 0208. The last \texttt{\kansuji} converts an integer into its Chinese numerals.

\texttt{\kuten, \jis, \euc, \sjis, \ucs, \kansuji}

These six commands takes an internal integer, and returns a string.

\begin{verbatim}
\newcount\hoge
\hoge=\texttt{2423}\% \hoge=9251, \texttt{\textbackslash jis\\textbackslash hoge}\texttt{\\textbackslash jis\hoge}\texttt{\\jis\hoge}\texttt{\\char\jis\hoge}\texttt{\textbackslash kansuji1701}
\end{verbatim}

To change characters of Chinese numerals for each digit, set \texttt{kansujichar} parameter:

\begin{verbatim}
\texttt{\ltjsetparameter{\textbackslash kansujichar={1,`壹}}}\texttt{\ltjsetparameter{\textbackslash kansujichar={7,`漆}}}\texttt{\ltjsetparameter{\textbackslash kansujichar={0,`零}}}\texttt{\kansuji1701}
\end{verbatim}

9.2 \texttt{\inhibitglue}

\texttt{\inhibitglue} suppresses the insertion of JAglue. The following is an example, using a special JFM that there will be a glue between the beginning of a box and “あ”, and also between “あ” and “ウ”.

\begin{verbatim}
\end{verbatim}
With the help of this example, we remark the specification of `\inhibitglue`:

- The call of `\inhibitglue` in the (internal) vertical mode is simply ignored.
- The call of `\inhibitglue` in the (restricted) horizontal mode is only effective on the spot; does not get over boundary of paragraphs. Moreover, `\inhibitglue` cancels ligatures and kernings, as shown in the last line of above example.
- The call of `\inhibitglue` in math mode is just ignored.

9.3 `\ltjfakeboxbdd`, `\ltjfakeparbegin`

Sometimes 'parbdd' and 'boxbdd' specifications look like "fail", especially in paragraphs inside list environments. This is because `\everypar` inserts some nodes such as boxes and kerns, so the "first letter" in a paragraph is in fact not the first letter.

\begin{verbatim}
\parindent1\zw
\noindent ああああああああ\par % for comparison
「ああああああ\par % normal paragraph ああああああああ
\everypar{null}
「ああああああ\par % ???
\end{verbatim}

`\ltjfakeboxbdd` and `\ltjfakeparbegin` primitives resolve this situation.

- `\ltjfakeparbegin` creates a node which indicates "beginning of an indented paragraph" to the insertion process of `JAglue`.
- `\ltjfakeboxbdd` creates a node which indicates "beginning/ending of a box" to the insertion process of `JAglue`.

As an example, the example above can be improved as follows:

\begin{verbatim}
\parindent1\zw
\noindent ああああああああ\par % for comparison
「ああああああ\par % normal paragraph ああああああああ
\everypar{null}\ltjfakeparbegin}
「ああああああ\par
\end{verbatim}

9.4 `\ltjdeclarealtfont`

Using `\ltjdeclarealtfont`, one can "compose" more than one Japanese fonts. This `\ltjdeclarealtfont` uses in the following form:

\begin{verbatim}
\ltjdeclarealtfont{base_font_cs}{alt_font_cs}{\langle range\rangle}
\end{verbatim}

where `\langle base_font_cs\rangle` and `\langle alt_font_cs\rangle` are defined by `\jfont`. Its meaning is

If the current Japanese font is `\langle base_font_cs\rangle`, characters which belong to `\langle range\rangle` is typeset by another Japanese font `\langle alt_font_cs\rangle`, instead of `\langle base_font_cs\rangle`.

Here `\langle range\rangle` is a comma-separated list of character codes, but also accepts negative integers: \(-n\ (n \geq 1)\) means that all characters of character classes \(n\), with respect to JFM used by `\langle base_font_cs\rangle`. Note that characters which do not exist in `\langle alt_font_cs\rangle` are ignored.

For example, if `\hoge` uses `jfm-ujis.lua`, the standard JFM of Lua\TeX\-ja, then
\ltjdeclarealtfont\hoge\piyo{"3000-30FF, {-1}--{-1}}
does
If the current Japanese font is \hoge, \texttt{U+3000–U+30FF} and characters in class 1 (ideographic opening brackets) are typeset by \piyo.

10 Commands for \LaTeX\

10.1 Loading Japanese fonts in \LaTeX\

From version 20190107, \LaTeX-ja does not load Japanese fonts for horizontal direction and that for vertical direction at the same time, to reduce the number of loaded fonts. This will save time for typesetting and memory consumption of Lua side ([11]).

- \selectfont loads (and chooses) only the Japanese font for current direction, and does not load the Japanese font for other direction (LuaTEX-ja only detects its size and JFM, to calculate the amount of shifting the baseline).

- Direction changing commands (\yoko, \tate, \dtou, \utod) are patched to include the following process:
  If the Japanese font for new direction is not loaded, LuaTEX-ja loads it automatically.

Original commands are saved as \ltj@@orig\yoko etc.

- Specifying Japanese font command which is defined by \jfont, \tfont, or \DeclareFixedFont directly actually loads (and selects) the Japanese font. For example, \texttt{JAchars} in \texttt{box0} will be typeset in \texttt{HOGE}, in the following code:

  \begin{verbatim}
  \% in horizontal direction \yoko
  \DeclareFixedFont\HOGE{JT3}{gt}{m}{n}{12} \% JT3: for vertical direction
  \HOGE
  \setbox0=\hbox{\tate あいう}
  \end{verbatim}

10.2 Patch for NFSS2

Japanese patch for NFSS2 in LuaTEX-ja is based on \texttt{plfonts.dtx} which plays the same role in \texttt{p\LaTeX}. We will describe commands which are not described in Subsection 3.1.

additonal dimensions
Like \texttt{pl\LaTeX}, LuaTEX-ja defines the following dimensions for information of current Japanese font:

- \cht (height), \cdp (depth), \chT (sum of former two),
- \cwd (width), \csvs (lineskip), \chs (equals to \cwd)

and its \texttt{\normalsize} version:

- \ChT (height), \Cdp (depth), \Cwd (width),
- \Cvs (equals to \baselineskip), \Chs (equals to \Cwd).

Note that \cwd and \chT may differ from \texttt{\zw} and \texttt{\zh} respectively. On the one hand the former dimensions are determined from the character "あ", but on the other hand \texttt{\zw} and \texttt{\zh} are specified by JFM.

\begin{verbatim}
\DeclareYokoKanjiEncoding{(encoding)}{(text-settings)}{(math-settings)}
\DeclareTateKanjiEncoding{(encoding)}{(text-settings)}{(math-settings)}
\end{verbatim}

In NFSS2 under LuaTEX-ja, distinction between alphabetic fonts and Japanese fonts are only made by their encodings. For example, encodings OT1 and T1 are encodings for alphabetic fonts, and Japanese fonts cannot have these encodings. These command define a new encoding scheme for Japanese font families.
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、……

Figure 5. An example of \DeclareAlternateKanjiFont

\texttt{\setrelationfont\{JY3\}\{\k@family\}\{m\}\{n\}\{TU\}\{lmss\}\{m\}\{n\}}

\% \k@family: current Japanese font family

\texttt{\fontfamily{\{family\}}}

As in \TeX\, \texttt{\selectfont} changes current font family (alphabetic, Japanese, or both) to \texttt{\{family\}}. See Subsection 10.3 for detail.

\texttt{\fontfamily{\{family\}}}
• In \ltjdeclarealtfont, the base font and the alternate font must be already defined. But this \DeclareAlternateKanjiFont is not so. In other words, \DeclareAlternateKanjiFont is effective only after current Japanese font is changed, or only after \selectfont is executed.

• ...

Furthermore, LuaTeX-ja applies patches which enables NFSS2 commands, such as \DeclareSymbolFont and \SetSymbolFont, to specify Japanese fonts as math fonts.

Specifying disablejfam option in \usepackage prevents applying these patches. Hence one cannot write Japanese Characters in math mode directly if disablejfam option is specified. The code below does not work either:

\DeclareSymbolFont{mincho}{JY3}{mc}{m}{n}
\DeclareSymbolFontAlphabet{\mathmc}{mincho}

10.3 Detail of \fontfamily command

In this subsection, we describe when \fontfamily{family} changes current Japanese/alphabetic font family. Basically, current Japanese font family is changed to \fontfamily{family} if it is recognized as a Japanese font family, and similar with alphabetic font family. There is a case that current Japanese/alphabetic font family are both changed to \fontfamily{family}, and another case that \fontfamily{family} isn’t recognized as a Japanese/alphabetic font family either.

■ Recognition as Japanese font family

First, Whether Japanese font family will be changed is determined in following order. This order is very similar to \fontfamily in p\TeX\ \varepsilon, but we re-implemented in Lua. We use an auxiliary list \textit{N}_J.

1. If the family \textit{family} has been defined already by \DeclareKanjiFamily, \textit{family} is recognized as a Japanese font family. Note that \textit{family} need not be defined under current Japanese font encoding.

2. If the family \textit{family} has been listed in a list \textit{N}_J, this means that \textit{family} is not a Japanese font family.

3. If the luatexja-fontspec package is loaded, we stop here, and \textit{family} is not recognized as a Japanese font family.

   If the luatexja-fontspec package is not loaded, now LuaTeX-ja looks whether there exists a Japanese font encoding \textit{enc} such that a font definition named \textit{enc}\textit{family}.fd (the file name is all lowercase) exists. If so, \textit{family} is recognized as a Japanese font family (the font definition file won’t be loaded here). If not, \textit{family} is not a Japanese font family, and \textit{family} is appended to the list \textit{N}_J.

■ Recognition as alphabetic font family

Next, whether alphabetic font family will be changed is determined in following order. We use auxiliary lists \textit{F}_A and \textit{N}_A.

1. If the family \textit{family} has been listed in a list \textit{F}_A, \textit{family} is recognized as an alphabetic font family.

2. If the family \textit{family} has been listed in a list \textit{N}_A, this means that \textit{family} is not an alphabetic font family.

3. If there exists an alphabetic font encoding such that the family \textit{family} has been defined under it, \textit{family} is recognized as an alphabetic font family, and to memorize this, \textit{family} is appended to the list \textit{F}_A.

4. Now LuaTeX-ja looks whether there exists an alphabetic font encoding \textit{enc} such that a font definition named \textit{enc}\textit{family}.fd (the file name is all lowercase) exists. If so, current alphabetic font family will be changed to \textit{family} (the font definition file won’t be loaded here). If not, current alphabetic font family won’t be changed, and \textit{family} is appended to the list \textit{N}_A.

Also, each call of \DeclareFontFamily after loading of LuaTeX-ja makes the second argument (family) is appended to the list \textit{F}_A.

The above order is very similar to \fontfamily in p\TeX\ \varepsilon, but more complicated (clause 3.). This is because p\TeX\ \varepsilon is a format however LuaTeX-ja is not, hence LuaTeX-ja does not know calls of \DeclareFontFamily before itself is loaded.
Table 12. strut

<table>
<thead>
<tr>
<th>box</th>
<th>direction</th>
<th>width</th>
<th>height</th>
<th>depth</th>
<th>user command</th>
</tr>
</thead>
<tbody>
<tr>
<td>\ystrutbox</td>
<td>yoko</td>
<td>0</td>
<td>0.7\baselineskip</td>
<td>0.3\baselineskip</td>
<td>\ystrut</td>
</tr>
<tr>
<td>\tstrutbox</td>
<td>tate, utod</td>
<td>0</td>
<td>0.5\baselineskip</td>
<td>0.5\baselineskip</td>
<td>\tstrut</td>
</tr>
<tr>
<td>\dstrutbox</td>
<td>dou</td>
<td>0</td>
<td>0.7\baselineskip</td>
<td>0.3\baselineskip</td>
<td>\dstrut</td>
</tr>
<tr>
<td>\zstrutbox</td>
<td>—</td>
<td>0</td>
<td>0.7\baselineskip</td>
<td>0.3\baselineskip</td>
<td>\zstrut</td>
</tr>
</tbody>
</table>

**Remarks** Of course, there is a case that ⟨family⟩ is not recognized as a Japanese font family, nor an alphabetic font family. In this case, Lua\TeX-ja treats "the argument ⟨family⟩ is wrong", so set both current alphabetic and Japanese font family to ⟨family⟩, to use the default family for font substitution.

10.4 Notes on \DeclareTextSymbol

From \TeX 2017/01/01, the standard encoding of Lua\TeX is changed to the \TU encoding. This means that symbols defined by T1 and TS1 encodings can be used without loading any package. To produce these symbols in alphabetic fonts in Lua\TeX-ja, Lua\TeX-ja patches \DeclareTextSymbol, and reloads tuenc.def.

Under original definition of \DeclareTextSymbol, internal commands which is defined by \DeclareTextSymbol (such as \T1\textquotedblleft) are chardef tokens. However, this no longer holds in Lua\TeX-ja; for example, the meaning of \TU\textquotedblleft is \ltjalchar8220L-

10.5 \strutbox

As p\TeX (2017/04/08 or later), \strutbox is a macro which is expanded to one of \ystrutbox, \tstrutbox, and \dstrutbox (all of them are shown in Table 12), according to the current direction. Similarly, \strut now uses one of these boxes.

11 Addon packages

Lua\TeX-ja has several addon packages. These addons are written as \TeX packages, but luatexja-otf and luatexja-adjust can be loaded in plain Lua\TeX by \input.

11.1 luatexja-fontspec

As described in Subsection 3.2, this optional package provides the counterparts for several commands defined in the fontspec package (requires fontspec v2.4). In addition to OpenType font features in the original fontspec, the following "font features" specifications are allowed for the commands of Japanese version:

\[ \text{CID=⟨name⟩, JFM=⟨name⟩, JFM-var=⟨name⟩} \]

These 3 keys correspond to cid, jfm and jfmmvar keys for \jfont and \tfont respectively. See Subsections 7.1 and 7.4 for details of cid, jfm and jfmmvar keys.

The CID key is effective only when with NoEmbed described below. The same JFM cannot be used in both horizontal Japanese fonts and vertical Japanese fonts, hence the JFM key will be actually used in YokoFeatures and TateFeatures keys.

NoEmbed

By specifying this key, one can use "name-only" Japanese font which will not be embedded in the output PDF file. See Subsection 7.4.

Kanjiskip=⟨bool⟩
\jfontspec[  \ YokoFeatures={{Color=007F00}}, TateFeatures={{Color=00007F},  \ TateFont=KozGoPr6N-Regular}  \{KozMinPr6N-Regular}  \hbox{{yoko 横組のテスト}}\hbox{{tate 縦組のテスト}}  \addjfontfeatures{{Color=FF0000}}  \hbox{{yoko 横組}}\hbox{{tate 縦組}}

Figure 6. An example of TateFeatures etc.

\jfontspec[  \ AltFont={{  \ {Range="4E00-"67FF, Color=007F00},  \ {Range="6800-"9EFF, Color=0000FF},  \ {Range="3040-"306F, Font=KozGoPr6N-Regular},  \ }}  \{KozMinPr6N-Regular}  \japanese

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、われらとわれらの子孫のために、
諸国民との協和による成果と、わが国全土にわたって自由のもたらす恵沢を確保し、……

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、われらとわれらの子孫のために、
諸国民との協和による成果と、わが国全土にわたって自由のもたらす恵沢を確保し、……

Figure 7. An example of AltFont

TateFeatures={{(features)}} TateFont=(font)

The TateFeatures key specifies font features which are only turned on in vertical writing, such as
Style=VerticalKana (vkna feature). Similarly, the TateFont key specifies the Japanese font which
will be used only in vertical writing. A demonstration is shown in Figure 6.

YokoFeatures={{(features)}}

The YokoFeatures key specifies font features which are only turned on in horizontal writing. A
demonstration is shown in Figure 6.

AltFont

As \ltjdeclarealtfont (Subsection 9.4) and \DeclareAlternateKanjiFont (Subsection 10.2),
with this key, one can typeset some Japanese characters by a different font and/or using different features.
The AltFont feature takes a comma-separated list of comma-separated lists, as the following:

AltFont = {  
  ...  
  { Range=(range), (features) },  
  { Range=(range), Font=(font name), (features) },  
  { Range=(range), Font=(font name) },  
  ...  
}

Each sublist should have the Range key (sublist which does not contain Range key is simply ignored).
A demonstration is shown in Figure 7.

Remark on AltFont, YokoFeatures, TateFeatures keys

In AltFont, YokoFeatures, TateFeatures keys, one cannot specify per-shape settings such as
BoldFeatures. For example,

AltFont = {  
  { Font=HogeraMin-Light, BoldFont=HogeraMin-Bold,  
    Range="3000-"30FF, BoldFeatures={{Color=007F00}} }  
}
does not work. Instead, one have to write

\begin{verbatim}
UprightFeatures = {
  AltFont = { { Font=HogeraMin-Light, Range="3000-"30FF, } },
},
BoldFeatures = {
  AltFont = { { Font=HogeraMin-Bold, Range="3000-"30FF, Color=007F00 } },
}
\end{verbatim}

On the other hand, YokoFeatures, TateFeatures and TateFont keys can be specified in each list in the AltFont key. Also, one can specify AltFont inside YokoFeatures, TateFeatures keys.

Note that features which are specified in YokoFeatures and TateFeatures are always interpreted after other "direction-independent" features. This explains why \addjfontfeatures at line 6 in Figure 6 has no effect, because a color specification is already done in YokoFeatures and TateFeatures keys.

11.2 luatexja-otf

This optional package supports typesetting glyphs by specifying a CID number. The package luatexja-otf offers the following 2 low-level commands:

\begin{verbatim}
\CID{(number)}
\end{verbatim}

Typeset a glyph whose CID number is \textit{(number)}. If the Japanese font is neither Adobe-Japan1, Adobe-GB1, Adobe-CNS1, Adobe-Korea1, nor Adobe-KR CID-keyed font, \LaTeX-ja treats that \textit{(number)} is a CID number of Adobe-Japan1 character collection, and tries to typeset a "most suitable glyph".

\begin{verbatim}
\UTF{(hex_number)}
\end{verbatim}

Typeset a character whose character code is \textit{(hex_number)} (in hexadecimal). This command is similar to \char"\textit{(hex_number)}", but please remind remarks below.

This package automatically loads \texttt{luatexja-ajmacros.sty}, which is slightly modified version of \texttt{ajmacros.sty}\footnote{Useful macros by INOUE Koich!, for the \texttt{japanese-otf} package.}. Hence one can use macros which are defined in \texttt{ajmacros.sty}, such as \texttt{aj\ 半角}.

\textbf{Remarks} Characters by \texttt{\CID{}} and \texttt{\UTF{}} commands are different from ordinary characters in the following points:

\begin{itemize}
  \item Always treated as \texttt{JAchar}.
  \item Processing codes for supporting OpenType features (e.g., glyph replacement and kerning) by the \texttt{luaotfload} package is not performed to these characters.
\end{itemize}

\textbf{Additional syntax of JFM} The package \texttt{luatexja-otf} extends the syntax of JFM; the entries of \texttt{chars} table in JFM now allows a string in the form 'AJ1-xxx', which stands for the character whose CID number in Adobe-Japan1 is xxx.

This extended notation is used in the standard JFM \texttt{jfm-ujis.lua} to typeset halfwidth Hiragana glyphs (CID 516–598) in halfwidth.

11.3 luatexja-adjust

(see Japanese version of this manual)

11.4 luatexja-ruby

This addon package provides functionality of "ruby" (furigana) annotations using callbacks of \LaTeX-ja. There is no detailed manual of \texttt{luatexja-ruby.sty} in English. (Japanese manual is another PDF file, \texttt{luatexja-ruby.pdf}.)
以上の原理は、「包除原理」とよく呼ばれるが

Group-ruby  By default, ruby characters (the second argument of \ruby) are attached to base characters (the first argument), as one object. This type of ruby is called group-ruby.

As the above example, ruby hangover is allowed on the Hiragana before/after its base characters.

Mono-ruby To attach ruby characters to each base characters (mono-ruby), one should use \ruby multiple times:

Jukugo-ruby Vertical bar | denotes a boundary of groups.

If there are multiple groups in one \ruby call, A linebreak between two groups is allowed.

If the width of ruby characters are longer than that of base characters, \ruby automatically selects the appropriate form among the line-head form, the line-middle form, and the line-end form.

11.5 lltjext.sty

pdflatX supplies additional macros for vertical writing in the plext package. The lltjext package which we want to describe here is the LuaTEx-ja counterpart of the plext package.

tabular, array, minipage environments
These environments are extended by <dir>, which specifies the direction, as follows:

\begin{tabular}<dir>[pos]{table spec} ... \end{tabular}
\begin{array}<dir>[pos]{table spec} ... \end{array}
\begin{minipage}<dir>[pos]{width} ... \end{minipage}
This option permits one of the following five values. If none of them is specified, the direction inside the environment is same as that outside the environment.

- \textit{y} yoko direction (horizontal writing)
- \textit{t} tate direction (vertical writing)
- \textit{z} utod direction if direction outside the env. is tate.
- \textit{d} dtou direction
- \textit{u} utod direction

\texttt{\parbox<\langle dir\rangle>\{(\langle width\rangle)\{(\langle contents\rangle)\}}\parbox command is also extended by \langle dir\rangle.

\texttt{\pbox<\langle dir\rangle>\{(\langle width\rangle)\{(\langle pos\rangle)\{(\langle contents\rangle)\}}\pbox command is also extended by \langle dir\rangle, as follows:

\begin{verbatim}
\begin{picture}\langle dir\rangle\{(x_size, y_size)\}(x_offset,y_offset)
...
\end{picture}
\end{verbatim}

\texttt{\rensuiji\{(\langle pos\rangle)\{(\langle contents\rangle)\}}, \rensujiskip}

\texttt{\Kanji\{(\langle counter_name\rangle)\}}

\texttt{\kasen\{(\langle contents\rangle)\}}, \bout\{(\langle contents\rangle)\}, \bouenchar}

参照番号

11.6 \texttt{luatexja-preset}

As described in Subsection 3.3. One can load the \texttt{luatexja-preset} package to use several “presets” of Japanese fonts. This package provides functions in a part of \texttt{japanese-otf} package (changing fonts) and a part of \texttt{PXchfon} package (presets) by Takayuki Yato.

Options which are given in \texttt{\usepackage} but not described in this subsection are simply passed to the \texttt{luatexja-fontspec} package. For example, the line 5 in below example is equivalent to lines 1–3.

\begin{verbatim}
\usepackage{no-math}{fontspec}
\usepackage{match}{luatexja-fontspec}
\usepackage[kozuka-pr6n]{luatexja-preset}
\%
\usepackage{no-math,match,kozuka-pr6n}{luatexja-preset}
\end{verbatim}

11.6.1 General Options

\texttt{fontspec} (enabled by default)

With this option, Japanese fonts are selected using functionality of the \texttt{luatexja-fontspec} package. This means that the fontspec package is automatically loaded by this package.

If you need to pass some options to fontspec, you can load fontspec manually before \texttt{luatexja-preset}:

\begin{verbatim}
\usepackage{no-math}{fontspec}
\usepackage[...]{luatexja-preset}
\end{verbatim}

\footnote{If \texttt{nfssonly} option is \textit{not} specified; in this case these options are simply ignored.}
**nfssonly**

With this option, selecting Japanese fonts won’t be performed using the functionality of the fontspec package, but only standard NFSS2 (hence without \addfontfeatures etc.). This option is ignored when luatexja-fontspec package is loaded.

When this option is specified, fontspec and luatexja-fontspec are *not* loaded by default. Nevertheless, the package fontspec can coexist with the option, as the following:

\usepackage{fontspec}
\usepackage[hiragino-pron,nfssonly]{luatexja-preset}

In this case, one can use \setmainfont etc. to select *alphabetic* fonts.

**match**

If this option is specified, usual family-changing commands such as \rmfamily, \textrm, \sffamily, ... also change Japanese font family. This option is passed to luatexja-fontspec, if fontspec option is specified.

**nodeluxe** *(enabled by default)*

The negation of deluxe option. Use one-weighted *mincho* and *gothic* font families. This means that \mcfamily\bfseries, \gfcfamily\bfseries and \gfcfamily\mdseries use the same font.

**deluxe**

Use the mincho family with three weights (light, medium, and bold), the gothic family with three weights (medium, bold, and extra bold), and *rounded gothic*. Mincho light and gothic extra bold can be by \mcfamily\ltseries and \gfcfamily\ebseries, respectively.

- Some presets do not have the light weight of mincho. In this case, we substitute the medium weight for the light weight.
- luatexja-preset does not produce an error (only produces a warning), even if (one of) fonts for \mcfamily\ltseries, \gfcfamily\ebseries, \mgfamily do not exist.

**expert**

Use horizontal/vertical kana alternates, and define a command \rubyfamily to use kana characters designed for ruby.

**bold**

Substitute bold series of gothic for bold series of mincho. If nodeluxe option is enabled, medium series of gothic is also changed, since we use same font for both series of gothic.

**jis90, 90jis**

Use JIS X 0208:1990 glyph variants if possible.

**jis2004, 2004jis**

Use JIS X 0213:2004 glyph variants if possible.

**jfm_yoko=(jfm)**

Use the JFM jfm-(jfm).lua for horizontal direction, instead of jfm-ujis.lua (default JFM).

**jfm_tate=(jfm)**

Use the JFM jfm-(jfm).lua for vertical direction, instead of jfm-ujisv.lua (default JFM).

**jis**

Same as jfm_yoko=jis.

Note that jis90, 90jis, jis2004 and 2004jis only affect with mincho, gothic (and, possibly rounded gothic) families defined by this package. We didn’t taken account of when more than one options among them are specified.

---

13Provided by \mgfamily and \textmg, because "rounded gothic" is called *maru gothic* (丸ゴシック) in Japanese.
### 11.6.2 Presets which support multi weights

Besides bizud, morisawa-pro, and morisawa-pr6n presets, fonts are specified by font name, not by file name. In following tables, starred fonts (e.g. KozGoPro-Regular*) are used for medium series of gothic, if and only if deluxe option is specified.

**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.

Kozuka Pro/Pr6N fonts are bundled with Adobe’s software, such as Adobe InDesign. There is not rounded gothic family in Kozuka fonts.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>kozuka-pro</th>
<th>kozuka-pr6</th>
<th>kozuka-pr6n</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mincho</strong></td>
<td>light</td>
<td>KozMinPro-Light</td>
<td>KozMinProVI-Light</td>
<td>KozMinPr6N-Light</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>KozMinPro-Regular</td>
<td>KozMinProVI-Regular</td>
<td>KozMinPr6N-Regular</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>KozMinPro-Bold</td>
<td>KozMinProVI-Bold</td>
<td>KozMinPr6N-Bold</td>
</tr>
<tr>
<td><strong>gothic</strong></td>
<td>medium</td>
<td>KozGoPro-Regular*</td>
<td>KozGoProVI-Regular*</td>
<td>KozGoPr6N-Regular*</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>KozGoPro-Bold</td>
<td>KozGoProVI-Bold</td>
<td>KozGoPr6N-Bold</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>KozGoPro-Heavy</td>
<td>KozGoProVI-Heavy</td>
<td>KozGoPr6N-Heavy</td>
</tr>
<tr>
<td><strong>rounded gothic</strong></td>
<td></td>
<td>KozGoPro-Heavy</td>
<td>KozGoProVI-Heavy</td>
<td>KozGoPr6N-Heavy</td>
</tr>
</tbody>
</table>

**hiragino-pro** Hiragino Pro (Adobe-Japan1-5) fonts.
**hiragino-pron** Hiragino ProN (Adobe-Japan1-5, JIS04-savvy) fonts.

Hiragino fonts (except Hiragino Mincho W2) are bundled with Mac OS X 10.5 or later. Some editions of a Japanese word-processor "一太郎 2012" includes Hiragino ProN fonts. Note that fonts for gothic extra bold (HiraKakuStd[N]-W8) only contains characters in Adobe-Japan1-3 character collection, while others contains those in Adobe-Japan1-5 character collection.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>hiragino-pro</th>
<th>hiragino-pron</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mincho</strong></td>
<td>light</td>
<td>Hiragino Mincho Pro W2</td>
<td>Hiragino Mincho ProN W2</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>Hiragino Mincho Pro W3</td>
<td>Hiragino Mincho ProN W3</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Hiragino Mincho Pro W6</td>
<td>Hiragino Mincho ProN W6</td>
</tr>
<tr>
<td><strong>gothic</strong></td>
<td>medium</td>
<td>Hiragino Kaku Gothic Pro W3*</td>
<td>Hiragino Kaku Gothic ProN W3*</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Hiragino Kaku Gothic Pro W6</td>
<td>Hiragino Kaku Gothic ProN W6</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>Hiragino Kaku Gothic Std W8</td>
<td>Hiragino Kaku Gothic StdN W8</td>
</tr>
<tr>
<td><strong>rounded gothic</strong></td>
<td></td>
<td>Hiragino Maru Gothic Pro W4</td>
<td>Hiragino Maru Gothic ProN W4</td>
</tr>
</tbody>
</table>

**bizud** BIZ UD fonts (by Morisawa Inc.) bundled with Windows 10 October 2018 Update.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mincho</strong></td>
<td>BIZ-UDMinchoM.ttc</td>
</tr>
<tr>
<td><strong>gothic</strong></td>
<td>BIZ-UDGothicR.ttc</td>
</tr>
<tr>
<td></td>
<td>BIZ-UDGothicB.ttc</td>
</tr>
<tr>
<td></td>
<td>BIZ-UDGothicB.ttc</td>
</tr>
<tr>
<td><strong>rounded gothic</strong></td>
<td></td>
</tr>
</tbody>
</table>
morisawa-pro Morisawa Pro (Adobe-Japan1-4) fonts.
morisawa-pr6n Morisawa Pr6N (Adobe-Japan1-6, JIS04-savvy) fonts.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>morisawa-pro</th>
<th>morisawa-pr6n</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>medium</td>
<td>A-OTF-RyuminPro-Light.otf</td>
<td>A-OTF-RyuminPr6N-Light.otf</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>A-OTF-FutoMinA101Pro-Bold.otf</td>
<td>A-OTF-FutoMinA101Pr6N-Bold.otf</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>A-OTF-GothicBBBPro-Medium.otf</td>
<td>A-OTF-GothicBBBPr6N-Medium.otf</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>A-OTF-FutoGoB101Pro-Bold.otf</td>
<td>A-OTF-FutoGoB101Pr6N-Bold.otf</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>A-OTF-MidashiGoPro-MB31.otf</td>
<td>A-OTF-MidashiGoPr6N-MB31.otf</td>
</tr>
<tr>
<td>rounded gothic</td>
<td>bold</td>
<td>A-OTF-Jun101Pro-Bold.otf</td>
<td>A-OTF-ShinMGoPr6N-Bold.otf</td>
</tr>
</tbody>
</table>

yu-win Yu fonts bundled with Windows 8.1.
yu-win10 Yu fonts bundled with Windows 10.
yu-osx Yu fonts bundled with OSX Mavericks.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>yu-win</th>
<th>yu-win10</th>
<th>yu-osx</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>light</td>
<td>YuMincho-Light</td>
<td>YuMincho-Light</td>
<td>(YuMincho Medium)</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>YuMincho-Regular</td>
<td>YuMincho-Regular</td>
<td>YuMincho Medium</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>YuMincho-DemiBold</td>
<td>YuMincho-DemiBold</td>
<td>YuMincho Demibold</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>YuGothic-Regular*</td>
<td>YuGothic-Regular*</td>
<td>YuGothic Medium*</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic Bold</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic Bold</td>
</tr>
<tr>
<td>rounded gothic</td>
<td>bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic Bold</td>
</tr>
</tbody>
</table>

moga-mobo MogaMincho, MogaGothic, and MoboGothic.
moga-mobo-ex MogaExMincho, MogaExGothic, and MoboExGothic.

These fonts can be downloaded from [http://yozvox.web.fc2.com/](http://yozvox.web.fc2.com/).

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>default, 90jis option</th>
<th>jis2004 option</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>medium</td>
<td>Moga90Mincho</td>
<td>MogaMincho</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Moga90Mincho Bold</td>
<td>MogaMincho Bold</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>Moga90Gothic</td>
<td>MogaGothic</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Moga90Gothic Bold</td>
<td>MogaGothic Bold</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>Moga90Gothic Bold</td>
<td>MogaGothic Bold</td>
</tr>
<tr>
<td>rounded gothic</td>
<td>bold</td>
<td>Moba90Gothic</td>
<td>MobaGothic</td>
</tr>
</tbody>
</table>

When moga-mobo-ex is specified, the font "MogaEx90Mincho" etc. are used.

ume Ume Mincho and Ume Gothic.

Source Han Serif and Source Han Sans fonts (Language-specific OTF or OTC)
Source Han Serif JP and Source Han Sans JP fonts (Region-specific Subset OTF)

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>sourcehan</th>
<th>sourcehan-jp</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>light</td>
<td>Source Han Serif Light</td>
<td>Source Han Serif JP Light</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>Source Han Serif Regular</td>
<td>Source Han Serif JP Regular</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Source Han Serif Bold</td>
<td>Source Han Serif JP Bold</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>Source Han Sans Regular*</td>
<td>Source Han Sans JP Regular*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source Han Sans Medium</td>
<td>Source Han Sans JP Medium</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Source Han Sans Bold</td>
<td>Source Han Sans JP Bold</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>Source Han Sans Heavy</td>
<td>Source Han Sans JP Heavy</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>Source Han Sans Heavy</td>
<td>Source Han Sans JP Heavy</td>
</tr>
</tbody>
</table>

Noto SerifCJK and Noto SansCJK fonts (OTC)
Noto SerifCJK and Noto SansCJK fonts (Language-specific OTF)

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>noto-otc</th>
<th>noto-otf</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>light</td>
<td>Noto Serif CJK Light</td>
<td>Noto Serif CJK JP Light</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>Noto Serif CJK Regular</td>
<td>Noto Serif CJK JP Regular</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Noto Serif CJK Bold</td>
<td>Noto Serif CJK JP Bold</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>Noto Sans CJK Regular*</td>
<td>Noto Sans CJK JP Regular*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Noto Sans CJK Medium</td>
<td>Noto Sans CJK JP Medium</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Noto Sans CJK Bold</td>
<td>Noto Sans CJK JP Bold</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>Noto Sans CJK Black</td>
<td>Noto Sans CJK JP Black</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>Noto Sans CJK Black</td>
<td>Noto Sans CJK JP Black</td>
</tr>
</tbody>
</table>

11.6.3 Presets which do not support multi weights

Next, we describe settings for using only single weight.

<table>
<thead>
<tr>
<th>noembed</th>
<th>ipa</th>
<th>ipaex</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>Ryumin-Light (non-embedded)</td>
<td>IPA Mincho</td>
<td>IPAex Mincho</td>
</tr>
<tr>
<td>gothic</td>
<td>GothicBBB-Medium (non-embedded)</td>
<td>IPA Gothic</td>
<td>IPAex Gothic</td>
</tr>
</tbody>
</table>

11.6.4 Presets which use HG fonts

We can use HG fonts bundled with Microsoft Office for realizing multiple weights. In the table below, starred fonts (e.g., IPA Gothic*) are used only if jis2004 or nodeluxe option is specified.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>ipa-hg</th>
<th>ipaex-hg</th>
<th>ms-hg</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>medium</td>
<td>IPA Mincho</td>
<td>IPAex Mincho</td>
<td>MS Mincho</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>HG Mincho E</td>
<td>HG Mincho E</td>
<td></td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>IPA Gothic*</td>
<td>IPAex Gothic*</td>
<td>MS Gothic*</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>HG Gothic M</td>
<td>HG Gothic M</td>
<td></td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>HG Soei Kaku Gothic UB</td>
<td>HG Soei Kaku Gothic UB</td>
<td></td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>HG Maru Gothic M PRO</td>
<td>HG Maru Gothic M PRO</td>
<td></td>
</tr>
</tbody>
</table>

Note that HG Mincho E, HG Gothic E, HG Soei Kaku Gothic UB, and HG Maru Gothic PRO are internally specified by:

default by font name (HGMinchoE, etc.).
jis90, 90jis by file name (hgrme.ttc, hgrge.ttc, hgrsgu.ttc, hgrsmp.ttf).
11.6.5 Define/Use Custom Presets

From version 20170904.0, one can define new presets using $\texttt{\LaTeX} \texttt{newpreset}$, and use them by $\texttt{\LaTeX} \texttt{applypreset}$. These two commands can only be used in the preamble.

$\texttt{\LaTeX} \texttt{newpreset}\{\langle \text{name} \rangle\}\{\langle \text{specification} \rangle\}$

Define new preset $\langle \text{name} \rangle$. This $\langle \text{name} \rangle$ cannot be same as other presets, options described in Sub-subsection 11.6.1, nor following nine strings:

$\langle \text{specification} \rangle$ is a comma-separated list which consists of other presets and/or the following keys:

- $\text{mc-l}=$ $\langle \text{font} \rangle$ mincho light
- $\text{mc-m}=$ $\langle \text{font} \rangle$ mincho medium
- $\text{mc-bx}=$ $\langle \text{font} \rangle$ mincho bold
- $\text{gt-m}=$ $\langle \text{font} \rangle$ gothic medium
- $\text{gt-bx}=$ $\langle \text{font} \rangle$ gothic bold
- $\text{gt-eb}=$ $\langle \text{font} \rangle$ gothic extra bold
- $\text{mg-m}=$ $\langle \text{font} \rangle$ rounded gothic
- $\text{mc}=$ $\langle \text{font} \rangle$ Same as $\text{mc-l}=$ $\langle \text{font} \rangle$, $\text{mc-m}=$ $\langle \text{font} \rangle$, $\text{mc-bx}=$ $\langle \text{font} \rangle$
- $\text{gt}=$ $\langle \text{font} \rangle$ Same as $\text{gt-m}=$ $\langle \text{font} \rangle$, $\text{gt-bx}=$ $\langle \text{font} \rangle$, $\text{gt-eb}=$ $\langle \text{font} \rangle$

If $\texttt{deluxe}$ is not specified at loading the package, only $\text{mc}$ and $\text{gt}$ keys (among above 9 keys) have a meaning.

$\texttt{\LaTeX} \texttt{newpreset*}\{\langle \text{name} \rangle\}\{\langle \text{specification} \rangle\}$

Almost same as $\texttt{\LaTeX} \texttt{newpreset}$. However, if $\langle \text{name} \rangle$ matches a preset which already defined, this command simply overwrite it.

$\texttt{\LaTeX} \texttt{applypreset}\{\langle \text{name} \rangle\}$

Set Japanese font families using preset $\langle \text{name} \rangle$.

Note that $\texttt{\LaTeX} \texttt{newpreset}$ does not “expand” the definition to define a preset. This means that one can write as the following:

$\texttt{\LaTeX} \texttt{newpreset}\{\texttt{hoge}\}\{\texttt{piyo,mc-bx=HiraMinProN-W6}\}$
$\texttt{\LaTeX} \texttt{newpreset}\{\texttt{piyo}\}\{\texttt{mg-m=HiraMaruProN-W4}\}$
$\texttt{\LaTeX} \texttt{applypreset}\{\texttt{hoge}\}$

**Restrictions** Presets which are defined by $\texttt{\LaTeX} \texttt{newpreset}$ have following restrictions:

- One cannot specify non-embedded fonts (such as Ryumin-Light).
- Some presets, such as $\texttt{ipa-hg}$, have a feature that fonts are changed according to whether $\texttt{90jis}$ or $\texttt{jis2004}$ is specified. This feature is not usable in presets which are defined by $\texttt{\LaTeX} \texttt{newpreset}$. 

48
Part III
Implementations

12 Storing Parameters

12.1 Used dimensions, attributes and whatsit nodes

Here the following is the list of dimensions and attributes which are used in Lua\TeX-ja.

\jQ (dimension) \jQ is equal to 1 Q = 0.25 mm, where "Q" (also called "級") is a unit used in Japanese phototypesetting. So one should not change the value of this dimension.

\jH (dimension) There is also a unit called “歯” which equals to 0.25 mm and used in Japanese phototypesetting. This \jH is the same \dimen register as \jQ.

\lj\zw (dimension) A temporal register for the “full-width” of current Japanese font. The command \zw sets this register to the correct value, and “return” this register itself.

\lj\zh (dimension) A temporal register for the “full-height” (usually the sum of height of imaginary body and its depth) of current Japanese font. The command \zh sets this register to the correct value, and “return” this register itself.

\jfam (attribute) Current number of Japanese font family for math formulas.

\lj\j@curjfnt (attribute) If this attribute is a positive number, it stores the font number of current Japanese font for horizontal direction. If this attribute is negative, it means that the Japanese font for horizontal direction is not loaded–Lua\TeX-ja only knows its size and JFM.

\lj\j@curtfnt (attribute) Similar to \lj\j@curjfnt, but with current Japanese font for vertical direction.

\lj\j@charclass (attribute) The character class of a JAchar. This attribute is only set on a glyph_node which contains a JAchar.

\lj\j@yablshift (attribute) The amount of shifting the baseline of alphabetic fonts in scaled point (2^{-16} pt).

\lj\j@ykblshift (attribute) The amount of shifting the baseline of Japanese fonts in scaled point (2^{-16} pt).

\lj\j@tablshift (attribute)
\lj\j@tkblshift (attribute)

\lj\j@autospc (attribute) Whether the auto insertion of kanjiskip is allowed at the node.

\lj\j@autoxspc (attribute) Whether the auto insertion of xkanjiskip is allowed at the node.

\lj\j@icflag (attribute) An attribute for distinguishing "kinds" of a node. One of the following value is assigned to this attribute:

italic (1) Kerns from italic correction (\), or from kerning information of a Japanese font. These kerns are “ignored” in the insertion process of JAgue, unlike explicit \kern.

packed (2)
kinsoku (3) Penalties inserted for the word-wrapping process (kinsoku shori) of Japanese characters.

(from_jfm − 2)−(from_jfm + 2) (4−8) Glues/kerns from JFM.
kanji_skip (9), kanji_skip_jfm (10) Glues from kanjiskip.
xkanji_skip (11), xkanji_skip_jfm (12) Glues from xkanjiskip.
processed (13) Nodes which is already processed by ....
ic_processed (14) Glues from an italic correction, but already processed in the insertion process of JAglues.
boxbdl (15) Glues/kerns that inserted just the beginning or the ending of an hbox or a paragraph.
\ltj@kcat $i$ (attribute) Where \( i \) is a natural number which is less than 7. These 7 attributes store bit vectors indicating which character block is regarded as a block of JAchars.
\ltj@dir (attribute) \texttt{dir\_node\_auto} (128)
\texttt{dir\_node\_manual} (256)
\ltjlineendcomment (counter)

Furthermore, LuaTeX-ja uses several user-defined whatsit nodes for internal processing. All those nodes except \texttt{direction} whatsits store a natural number (hence its type is 100). \texttt{direction} whatsits store a node list, hence its type is 110. Their \texttt{user\_id} (used for distinguish user-defined whatsits) are allocated by \texttt{luatexbase.newuserwhatsitid}.

\texttt{inhibitglue} Nodes for indicating that \texttt{\nofollow{\texttt{\textbackslash inhibitglue}}} is specified. The \texttt{value} field of these nodes doesn’t matter.

\texttt{stack\_marker} Nodes for LuaTeX-ja’s stack system (see the next subsection). The \texttt{value} field of these nodes is current group level.
\texttt{char\_by\_cid} Nodes for JAchar which the callback process of luaotfload won’t be applied, and the character code is stored in the \texttt{value} field. Each node of this type are converted to a \texttt{glyph\_node after} the callback process of luaotfload. Nodes of this type is used in \texttt{CID}, \texttt{UTF} and IVS support.
\texttt{replace\_vs} Similar to \texttt{char\_by\_cid} whatsits above. These nodes are for ALchar which the callback process of luaotfload won’t be applied.

\texttt{begin\_par} Nodes for indicating beginning of a paragraph. A paragraph which is started by \texttt{\item} in list-like environments has a horizontal box for its label before the actual contents. So ...

\texttt{direction}

These whatsits will be removed during the process of inserting JAglues.

12.2 Stack system of LuaTeX-ja

\textbf{Background} LuaTeX-ja has its own stack system, and most parameters of LuaTeX-ja are stored in it. To clarify the reason, imagine the parameter \texttt{kanjiskip} is stored by a skip, and consider the following source:
\begin{verbatim}
1 \ltjsetparameter{kanjiskip=0pt}ふがふが.
2 \setbox0=\hbox{%
3 \ltjsetparameter{kanjiskip=5pt}ほげほげ%
4 \box0.ぴよぴよ

50
\end{verbatim}

As described in Subsection 8.1, the only effective value of \texttt{kanjiskip} in an hbox is the latest value, so the value of \texttt{kanjiskip} which applied in the entire hbox should be 5pt. However, by the implementation method of LuaTeX, this “5pt” cannot be known from any callbacks. In the \texttt{tex/packaging.w}, which is a file in the source of LuaTeX, there are the following codes:
\begin{verbatim}
122 void package(int c)  
123 {  
124   scaled h; /* height of box */
125   halfword p; /* first node in a box */
126   scaled d; /* max depth */
127   int grp;  
128   grp = cur_group;  
129   d = box_max_depth;   
130   unsave();
\end{verbatim}
save_ptr -= 4;
if (cur_list.mode_field == -hmode) {
    cur_box = filtered_hppack(cur_list.head_field,
    cur_list.tail_field, saved_value(1),
    saved_level(1), grp, saved_level(2));
    subtype(cur_box) = HLIST_SUBTYPE_HBOX;
}

Notice that unsave() is executed before filtered_hppack(), where hpack_filter callback is executed) here. So “5 pt” in the above source is orphaned at unsave(), and hence it can’t be accessed from hpack_filter callback.

Implementation  The code of stack system is based on that in a post of Dev-luatex mailing list.14

These are two \TeX count registers for maintaining information: \texttt{\@stack} for the stack level, and \texttt{\@group@level} for the \TeX’s group level when the last assignment was done. Parameters are stored in one big table named \texttt{charprop_stack_table}, where \texttt{charprop_stack_table[i]} stores data of stack level \texttt{i}. If a new stack level is created by \texttt{\setparameter}, all data of the previous level is copied.

To resolve the problem mentioned in above paragraph “Background”, Lua\TeX-ja uses another trick. When the stack level is about to be increased, a whatsit node whose type, subtype and value are \texttt{44 (user defined), stack_marker} and the current group level respectively is appended to the current list (we refer this node by \texttt{stack_flag}). This enables us to know whether assignment is done just inside a hbox.

Suppose that the stack level is \texttt{s} and the \TeX’s group level is \texttt{t} just after the hbox group, then:

- If there is no \texttt{stack_flag} node in the list of the contents of the hbox, then no assignment was occurred inside the hbox. Hence values of parameters at the end of the hbox are stored in the stack level \texttt{s}.
- If there is a \texttt{stack_flag} node whose value is \texttt{t + 1}, then an assignment was occurred just inside the hbox group. Hence values of parameters at the end of the hbox are stored in the stack level \texttt{s + 1}.
- If there are \texttt{stack_flag} nodes but all of their values are more than \texttt{t + 1}, then an assignment was occurred in the box, but it is done in more internal group. Hence values of parameters at the end of the hbox are stored in the stack level \texttt{s}.

Note that to work this trick correctly, assignments to \texttt{\@stack} and \texttt{\@group@level} have to be local always, regardless the value of \texttt{\globaldefs}. To solve this problem, we use another trick: the assignment \texttt{\directlua{tex.globaldefs=0}} is always local.

12.3 Lua functions of the stack system

In this subsection, we will see how a user use Lua\TeX-ja’s stack system to store some data which obeys the grouping of \TeX.

The following function can be used to store data into a stack:
\texttt{luatexja.stack.set_stack_table(index, \texttt{\textless any\texttt{\textgreater} data})}

Any values which except \texttt{nil} and NaN are usable as \texttt{index}. However, a user should use only negative integers or strings as \texttt{index}, since natural numbers are used by Lua\TeX-ja itself. Also, whether \texttt{data} is stored locally or globally is determined by \texttt{luatexja.isglobal} (stored globally if and only if \texttt{luatexja.isglobal == ’global’}).

Stored data can be obtained as the return value of
\texttt{luatexja.stack.get_stack_table(index, \texttt{\textless any\texttt{\textgreater} default, \texttt{\textless number\texttt{\textgreater} level})}

where \texttt{level} is the stack level, which is usually the value of \texttt{\@stack}, and \texttt{default} is the default value which will be returned if no values are stored in the stack table whose level is \texttt{level}.

12.4 Extending Parameters

Keys for \texttt{\setparameter} and \texttt{\getparameter} can be extended, as in luatex-adj.
Figure 9. Definition of parameter setting commands

**Setting parameters**  Figure 9 shows the *most outer* definition of two commands, \ltjsetparameter and \ltjglobalsetparameter. Most important part is the last \setkeys, which is offered by the xkeyval package.

Hence, to add a key in \ltjsetparameter, one only have to add a key whose prefix is ltj and whose family is japaram, as the following.
\define@key[ltj]{japaram}{...}{...}
\ltjsetparameter and \ltjglobalsetparameter automatically sets luatexja.isglobal. Its meaning is the following.

\[
luatexja.isglobal = \begin{cases} 
  \text{'global'}, & \text{global} \\
  \text{'local'}, & \text{local} 
\end{cases}
\]

This is determined not only by command name (\ltjsetparameter or \ltjglobalsetparameter), but also by the value of \globaldefs.

**Getting parameters**  \ltjgetparameter is implemented by a Lua script.

For parameters that do not need additional arguments, one only have to define a function in the table luatexja.unary_pars. For example, with the following function, \ltjgetparameter{hoge} returns a *string* 42.

```latex
1 \begin{verbatim}
2 function luatexja.unary_pars.hoge (t)
3  return 42
4 end
5\end{verbatim}
```

Here the argument of luatexja.unary_pars.hoge is the stack level of LuaTeX-ja's stack system (see Subsection 12.2).

On the other hand, for parameters that need an additional argument (this must be an integer), one have to define a function in luatexja.binary_pars first. For example,

```latex
1 \begin{verbatim}
2 function luatexja.binary_pars.fuga (c, t)
3  return tostring(c) .. ' , ' .. tostring(42)
4 end
5\end{verbatim}
```

Here the first argument \(t\) is the stack level, as before. The second argument \(c\) is just the second argument of \ltjgetparameter.

For parameters that need an additional argument, one also have to execute the \TeX code like \ltj@@decl@array@param{fuga} to indicate that "the parameter fuga needs an additional argument".

### 13 Linebreak after a Japanese Character

#### 13.1 Reference: behavior in \TeX

In \TeX, a line break after a Japanese character doesn’t emit a space, since words are not separated by spaces in Japanese writings. However, this feature isn’t fully implemented in LuaTeX-ja due to the specification of
• 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 \TeX, 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.

Figure 10. State transitions of p\TeX’s input processor

callbacks in Lua\TeX. To clarify the difference between p\TeX and Lua\TeX, We briefly describe the handling of a line break in p\TeX, in this subsection.
p\TeX’s input processor can be described in terms of a finite state automaton, as that of \TeX in Section 2.5 of [1]. The internal states are as follows:

• State N: new line
• State S: skipping spaces
• State M: middle of line
• State K: after a Japanese character

The first three states—N, S, and M—are as same as \TeX’s input processor. State K is similar to state M, and is entered after Japanese characters. The diagram of state transitions are indicated in Figure 10. Note that p\TeX doesn’t leave state K after “beginning/ending of a group” characters.

13.2 Behavior in Lua\TeX-ja

States in the input processor of Lua\TeX is the same as that of \TeX, and they can’t be customized by any callbacks. Hence, we can only use process_input_buffer and token_filter callbacks for to suppress a space by a line break which is after Japanese characters.

However, token_filter callback cannot be used either, since a character in category code 5 (end-of-line) is converted into an space token in the input processor. So we can use only the process_input_buffer callback. This means that suppressing a space must be done just before an input line is read.

Considering these situations, handling of an end-of-line in Lua\TeX-ja are as follows:

A character whose character code is \texttt{\textbackslash l\textbackslash t}j\texttt{lineendcomment} is appended to an input line, before Lua\TeX actually process it, if and only if the following three conditions are satisfied:

1. The category code of \texttt{\textbackslash endlinechar} is 5 (end-of-line).

\footnote {Its default value is **FFFFF, so U+FFFFF is used. The category code of U+FFFFF is set to 14 (comment) by Lua\TeX-ja.}

\footnote {Usually, it is (return) (whose character code is 13).}
2. The category code of `\ltjlineendcomment` itself is 14 (comment).
3. The input line matches the following "regular expression":
   \( (\text{any char}(\land \text{catcode = 1}) \lor (\text{catcode = 2}))^* \)

**Remark** The following example shows the major difference from the behavior of p\TeX.

```latex
\begin{lstlisting}
葛城 市 , 葛飾 区 , 葛西
\end{lstlisting}
```

It is not strange that “あ” does not printed in the above output. This is because \TeX Gyre Termes does not contain “あ”, and because “あ” in line 3 is considered as an \textbf{AChar}.

Note that there is no space before “y” in the output, but there is a space before “u”. This follows from following reasons:

- When line 3 is processed by \texttt{process_input_buffer} callback, “あ” is considered as an \textbf{JAchar}. Since line 3 ends with an \textbf{JAchar}, the comment character (whose character code is `\ltjlineendcomment`) is appended to this line, and hence the linebreak immediately after this line is ignored.

- When line 4 is processed by \texttt{process_input_buffer} callback, “い” is considered as an \textbf{ALchar}. Since line 4 ends with an \textbf{ALchar}, the linebreak immediately after this line emits a space.

### 14 Patch for the listings Package

It is well-known that the listings package outputs weird results for Japanese input. The listings package makes most of letters active and assigns output command for each letter ([2]). But Japanese characters are not included in these activated letters. For p\TeX series, there is no method to make Japanese characters active; a patch \texttt{jlisting.sty} ([4]) resolves the problem forcibly.

In Lua\TeX-ja, the problem is resolved by using the \texttt{process_input_buffer} callback. The callback function inserts the output command (active character `\ltjlineendcomment`) before each letter above \texttt{U+0080}. This method can omits the process to make all Japanese characters active (most of the activated characters are not used in many cases).

If the listings package and Lua\TeX-ja were loaded, then the patch \texttt{lltjp-listings} is loaded automatically at \texttt{\begin{document}}.

#### 14.1 Notes and additional keys

**Variation selectors** \texttt{lltjp-listings} add two keys, namely \texttt{vsraw} and \texttt{vscmd}, which specify how variation selectors are treated in \texttt{lstlisting} or other environments. Note that these additional keys are not usable in the preamble, since \texttt{lltjp-listings} is loaded at \texttt{\begin{document}}.

\texttt{vsraw} is a key which takes a boolean value, and its default value is false.

- If the \texttt{vsraw} key is true, then variation selectors are "combined" with the previous character.

```latex
\begin{lstlisting}[vsraw=true]
葛城 市 , 葛飾 区 , 葛西
\end{lstlisting}
```

- If the \texttt{vsraw} key is false, then variation selectors are typeset by an appropriate command, which is specified by the \texttt{vscmd} key. The default setting of the \texttt{vscmd} key produces the following.

```latex
\begin{lstlisting}[vsraw=false,
  vscmd=\ltjlistingsvsstdcmd]
葛城 市 , 葛飾 区 , 葛西
\end{lstlisting}
```
For example, the following code is the setting of the `vscmd` key in this document.

\begin{verbatim}
\def\IVSA#1#2#3#4#5{%
\textcolor{blue}\raisbox{3.5pt}{\tt
\fboxsep=0.5pt\fbox{\tiny \oalign{0#1#2\crcr#3#4#5\crcr}}}%
,}
{\catcode`%=11
\gdef\IVSB#1{\expandafter\IVSA\directlua{
local cat_str = luatexbase.catcodetables['string']
tex.sprint(cat_str, string.format('%X', 0xE00EF+#1))}}}
\lstset{vscmd=\IVSB}
\end{verbatim}

The default output command of variation selectors is stored in `\ltjlistingsvsstdcmd`.

\begin{itemize}
  \item The **doubleletterspace** key \hspace{1em} Even the column format is fixed, sometimes characters are not vertically aligned. The following example is typeset with `basewidth=2em`, and you’ll see the leftmost ”H” are not vertically aligned.
  \begin{verbatim}
  : H :
  : H H H H :
  \end{verbatim}

  lltjp-listing adds the **doubleletterspace** key (not activated by default, for compatibility) to improve the situation, namely doubles inter-character space in each output unit. With this key, the above input now produces better output.
  \begin{verbatim}
  : H :
  : H H H H :
  \end{verbatim}
\end{itemize}

\section{Class of characters}

Roughly speaking, the listings package processes input as follows:

1. Collects letters and digits, which can be used for the name of identifiers.
2. When reading an other, outputs the collected character string (with modification, if needed).
3. Collects others.
4. When reading a letter or a digit, outputs the collected character string.
5. Turns back to 1.

By the above process, line breaks inside of an identifier are blocked. A flag `\lst@ifletter` indicates whether the previous character can be used for the name of identifiers or not.

For Japanese characters, line breaks are permitted on both sides except for brackets, dashes, etc. Hence the patch lltjp-listings introduces a new flag `\lst@ifkanji`, which indicates whether the previous character is a Japanese character or not. For illustration, we introduce following classes of characters:

\begin{tabular}{|c|c|}
\hline
Letter & Other \\
\hline
`\lst@ifletter` & T & F \\
`\lst@ifkanji` & F & F \\
\hline
\textbf{Meaning} & char in an identifier & other alphabet \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|}
\hline
Kanji & Open & Close \\
\hline
`\lst@ifletter` & T & F \\
`\lst@ifkanji` & T & F \\
\hline
\textbf{Meaning} & most of Japanese char & opening brackets & closing brackets \\
\hline
\end{tabular}
Note that digits in the listings package can be Letter or Other according to circumstances.

For example, let us consider the case an Open comes after a Letter. Since an Open represents Japanese open brackets, it is preferred to be permitted to insert line break after the Letter. Therefore, the collected character string is output in this case.

The following table summarizes $5 \times 5 = 25$ cases:

<table>
<thead>
<tr>
<th>Prev</th>
<th>Next</th>
<th>Letter</th>
<th>Other</th>
<th>Kanji</th>
<th>Open</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>collects</td>
<td>outputs</td>
<td></td>
<td>collects</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>outputs</td>
<td>collects</td>
<td>outputs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the above table,

- “outputs” means to output the collected character string (i.e., line breaking is permitted there).
- “collects” means to append the next character to the collected character string (i.e., line breaking is prohibited there).

Characters above or equal to U+0080 except Variation Selectors are classified into above 5 classes by the following rules:

- **ALchars** above or equal to U+0080 are classified as Letter.
- **JAchars** are classified in the order as follows:
  1. Characters whose prebreakpenalty is greater than or equal to 0 are classified as Open.
  2. Characters whose postbreakpenalty is greater than or equal to 0 are classified as Close.
  3. Characters that don’t satisfy the above two conditions are classified as Kanji.

The width of halfwidth kana (U+FF61–U+FF9F) is same as the width of **ALchar**; the width of the other **JAchars** is double the width of **ALchar**.

This classification process is executed every time a character appears in the **lstlisting** environment or other environments/commands.

### 15 Cache Management of LuaTEx-ja

LuaTEx-ja creates some cache files to reduce the loading time. in a similar way to the luaotfload package:

- Cache files are usually stored in (and loaded from) `$TEXMFVAR/luatexja/`.
- In addition to caches of the text form (the extension is "\.lua"), caches of the binary, precompiled form are supported.
  - We cannot share same binary cache for LuaTEx and LuaJITTEx. Hence we distinguish them by their extension, "\.luc" for LuaTEx and "\.lub" for LuaJITTEx.
  - In loading a cache, the binary cache precedes the text form.
  - When LuaTEx-ja updates a cache hoge\.lua, its binary version is also updated.

#### 15.1 Use of cache

LuaTEx-ja uses the following cache:
Table 13. cid key and corresponding files

<table>
<thead>
<tr>
<th>cid key</th>
<th>name of the cache</th>
<th>used CMaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adobe-Japan1-*</td>
<td>ltj-cid-auto-adobe-japan1.lua</td>
<td>UniJIS2004-UTF32-* Adobe-Japan1-UCS2</td>
</tr>
<tr>
<td>Adobe-Korea1-*</td>
<td>ltj-cid-auto-adobe-korea1.lua</td>
<td>UniKS-UTF32-* Adobe-Korea1-UCS2</td>
</tr>
<tr>
<td>Adobe-GB1-*</td>
<td>ltj-cid-auto-adobe-gb1.lua</td>
<td>UniGB-UTF32-* Adobe-GB1-UCS2</td>
</tr>
<tr>
<td>Adobe-CNS1-*</td>
<td>ltj-cid-auto-adobe-cns1.lua</td>
<td>UniCNS-UTF32-* Adobe-CNS1-UCS2</td>
</tr>
</tbody>
</table>

ltj-cid-auto-adobe-japan1.lua
The font table of a CID-keyed non-embedded Japanese font. This is loaded in every run. It is created from three CMaps, UniJIS2004-UTF32-{H,V} and Adobe-Japan1-UCS2, and this is why these two CMaps are needed in the first run of LuaTeX-ja.

Similar caches are created as Table 13, if you specified cid key in \jfont to use other CID-keyed non-embedded fonts for Chinese or Korean, as in Page 24.

extra/***..lua
This file stores the table which stores the following.

- unicode variants in a font "***"
- vertical width of glyphs, if it is not equal to the sum of the height of ascender and the depth of descender
- vertical variants

The following is the structure of the that table.

```lua
return {
  [10955]={ -- U+2ACB "Subset Of Above Not Equal To"
    [65024]=983879, -- <2ACB FE00>
    ["vwidth"]=0.98, -- vertical width
  },
  [37001]={ -- U+9089 "邉"
    [0]=37001, -- <9089 E0100>
    991049, -- <9089 E0101>
    ... "vform"=995025, -- vertical variant
  },
  ["unicodes"]={
    ["aj102.pe.vert"]=984163, -- glyph name to unicode
    ...
  },
  ["chksum"]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF", -- checksum of the fontfile
  ["version"]=11, -- version of the cache
}
```

ltj-jisx0208.{luc|lub}
The binary version of ltj-jisx0208.lua. This is the conversion table between JIS X 0208 and Unicode which is used in Kanji-code conversion commands for compatibility with \pTeX.

15.2 Internal
Cache management system of LuaTeX-ja is stored in \luatexja.base (ltj-base.lua). There are three public functions for cache management in \luatexja.base, where (filename) stands for the file name without suffix:

```lua
save_cache((filename), (data))
```
Save a non-nil table (data) into a cache (filename). Both the text form (filename).lua and its binary version are created or updated.
save_cache_luc((filename), (data), (serialized_data))

Same as save_cache, except that only the binary cache is updated. The third argument (serialized_data) is not usually given. But if this is given, it is treated as a string representation of (data).

load_cache((filename), (outdate))

Load the cache (filename). (outdate) is a function which takes one argument (the contents of the cache), and its return value is whether the cache is outdated.

load_cache first tries to read the binary cache (filename).luc|lub. If its contents is up-to-date, load_cache returns the contents. If the binary cache is not found or its contents is outdated, load_cache tries to read the text form (filename).lua. Hence, the return value of load_cache is non-nil, if and only if the updated cache is found.
References


   http://tug.org/tug2013/slides/TUG2013_upTeX.pdf


[6] 乙部厳己. 「min10 フォントについて」


[8] 濱野尚人, 田村明史, 倉沢良一. 「\TeX{} の出版への応用—縦組み機能の組み込み—」.
   .../texmf-dist/doc/ptex/base/ptxdoc.pdf

