The LuaTeX-ja package

The LuaTeX-ja project team

20221006.0 (October 6, 2022)
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 \texttt{pTeX}, an extension of \texttt{TeX}, and its derivatives are used to typeset Japanese documents in \texttt{TeX}. \texttt{pTeX} is an engine extension of \texttt{TeX}: so it can produce high-quality Japanese documents without using very complicated macros. But this point is a mixed blessing: \texttt{pTeX} is left behind from other extensions of \texttt{TeX}, especially \texttt{eTeX} and pdf\texttt{TeX}, and from changes about Japanese processing in computers (e.g., the UTF-8 encoding).

Recently extensions of \texttt{pTeX}, namely up\texttt{TeX} (Unicode-implementation of \texttt{pTeX}) and \texttt{e-pTeX} (merging of \texttt{pTeX} and \texttt{e-TeX} extension), have developed to fill those gaps to some extent, but gaps still exist.

However, the appearance of Lua\texttt{TeX} changed the whole situation. With using Lua “callbacks”, users can customize the internal processing of Lua\texttt{TeX}. 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 \texttt{pTeX}

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

The followings are major changes from \texttt{pTeX}. For more detailed information, see Part III or other sections of this manual.

\begin{itemize}
  \item \textbf{Command names} \texttt{pTeX} adds several primitives, such as \texttt{\kanjiskip}, \texttt{\prebreakpenalty}, and \texttt{\ifydir}. They can be used as follows:

  \begin{verbatim}
\kanjiskip=10pt \dimen0=\kanjiskip
\tbaselineshift=0.1zw
\dimen0=\tbaselineshift
\prebreakpenalty=100
\ifydir...i
  \end{verbatim}

  However, we cannot use them under Lua\texttt{TeX}-ja. Instead of them, we have to write as the following.

  \begin{verbatim}
\ltjsetparameter{\kanjiskip=10pt} \dimen0=\ltjgetparameter{\kanjiskip}
\ltjsetparameter{\tbaselineshift=0.1\zw}
\dimen0=\ltjgetparameter{\tbaselineshift}
\ltjsetparameter{\prebreakpenalty={`,100}}
\ifnum\ltjgetparameter{\direction}=4...i
  \end{verbatim}

  Note that \texttt{pTeX} adds new two useful units, namely \texttt{zw} and \texttt{zh}. As shown above, \textit{they are changed to \texttt{\zw} and \texttt{\zh} respectively in Lua\texttt{TeX}-ja.\footnote{Lua\texttt{TeX}-ja 20200127.0 introduces \texttt{\ltj@zw} and \texttt{\ltj@zh}, which are copy of \texttt{\zw} and \texttt{\zh}.}}

  \item \textbf{Linebreak after a Japanese character} In \texttt{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, Lua\texttt{TeX}-ja doesn’t have this feature completely, because of a specification of Lua\texttt{TeX}. For the detail, see Section 15.
\end{itemize}
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 \LaTeX's internal ligature handling is node-based (e.g., of{}fice 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, Lua\TeX-ja supports vertical writing. We implement this feature by using callbacks of Lua\TeX; so it must not be confused with \Omega-style direction support of Lua\TeX 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, Lua\TeX-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.

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 sens-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 Qiyuan
2 Getting Started

2.1 Installation

The following packages are needed for the LuaTeX-ja package.

- Lua\TeX\ 1.10.0 (or later) (DVI output (\outputmode=8 is not supported.)
- recent luaotfload (v3.1 or later recommended)
- adobemapping (Adobe cmap and pdfmapping files)
- E\TeX\ 2\nu\ 2020-02-02 patch level 5 or later (if you want to use E\TeX\-ja with E\TeX\)
- etoolbox (if you want to use Lua\TeX\-ja with E\TeX\)
- everytex (only for E\TeX\ 2\nu\ 2020-02-02 and 2020-10-01)
- filehook, atbegshi (only for E\TeX\ 2\nu\ 2020-02-02)
- ltxcmds, pdftexcmds
- fonts\texttt{e} v2.7c (or later)
- Harano Aji fonts (https://github.com/trueroad/HaranoAjiFonts)
  More specifically, HaranoAjiMincho-Regular and HaranoAjiGothic-Medium.

Now Lua\TeX\-ja is available from CTAN (in the macros/luatex/generic/luatexja directory), and the following distributions:

- \TeX\ Live (in texmf-dist/tex/luatex/luatexja)
- W32\TeX\ (in luatexja.tar.xz)
- MiK\TeX\ (in luatexja.tar.xz)

Harano Aji fonts are also available in these distributions (haranoaji in \TeX\ Live and MiK\TeX, and luatexja.tar.xz in W32\TeX).

■ HarfBuzz and Lua\TeX\-ja Using Lua\TeX\-ja with LuaHB\TeX\(\texttt{e}\)(Lua\TeX\ integrated with HarfBuzz) is not well tested. Maybe documents can typeset without an error, but with unwanted results (especially, vertical typesetting and \CID).

Especially, We don’t recommend defining a Japanese font with HarfBuzz, by specifying \texttt{Renderer=Harfbuzz} etc. (fonts\texttt{e}) or \texttt{mode=harf} (otherwise).

■ Manual installation

1. Download the source, by one of the following method. At the present, Lua\TeX\-ja has no stable release.

- Clone the Git repository by
  $ \texttt{git}\ clone \texttt{git://git.osdn.jp/gitroot/luatex-ja/luatexja.git}$

- Download the tar.gz archive of HEAD in the master branch from
  \texttt{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 src/ and several other sub-directories. But only the contents in src/ are needed to work Lua\TeX\-ja.
3. If you downloaded this package from CTAN, you have to run following commands to generate classes:

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

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

5. If `maketexlr` 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`, and uses a lot of memory.
- (Outdated) note for MiK\TeX 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 MiK\TeX does not satisfy this condition, so you will encounter an error like the following:

```latex
! Lua\TeX error ...les (x86)/MiKTeX 2.9/tex/luatex/luatexja/l\tj-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.

- Note that when Lua\TeX-ja is loaded in plain Lua\TeX, we cannot use color specification on font loading, such as

```latex
\font\hoge=lmroman10-regular.otf:color=FF0000 \fontprimitive
```

This is because codes for shifting baseline in math mode (Lua\TeX-ja) collide with and prevents loading codes for font color (luaotfload) in these environments. **We recommend to use \LaTeX 2020-02-02 (or later)**, since we can avoid this collision in there.

### 2.3 Using in plain \TeX

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

```latex
\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>“10 pt”</th>
<th>“7 pt”</th>
<th>“5 pt”</th>
</tr>
</thead>
<tbody>
<tr>
<td>yoko</td>
<td>mincho</td>
<td>HaranoAjiMincho-Regular</td>
<td>\tenmin \sevenmin \fivemin</td>
<td>\tengt \sevengt \fivetgt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gothic</td>
<td>HaranoAjiGothic-Medium</td>
<td>\tenmin \sevenmin \fivemin</td>
<td>\tengt \sevengt \fivetgt</td>
<td></td>
</tr>
<tr>
<td>tate</td>
<td>mincho</td>
<td>HaranoAjiMincho-Regular</td>
<td>\tenmin \sevenmin \fivemin</td>
<td>\tengt \sevengt \fivetgt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>gothic</td>
<td>HaranoAjiGothic-Medium</td>
<td>\tenmin \sevenmin \fivemin</td>
<td>\tengt \sevengt \fivetgt</td>
<td></td>
</tr>
</tbody>
</table>

– The “default” Japanese fonts (and JFMs for them) can be modified by defining \jstcmd{\stdmcfont etc. before one inputs luatexja.sty (Subsection 8.3).

– 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 \texttt{JAchar} and an \texttt{ALchar} (the parameter \xkanjiskip) is set to

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

2.4 Using in \LaTeX

Using in \LaTeX is basically same. To set up the minimal environment for Japanese, you only have to load luatexja.sty:

\begin{verbatim}
\usepackage{luatexja}
\end{verbatim}

It also does minimal settings (counterparts in \PLaTeX are plfonts.dtx and pldefs.ltx).

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

- Traditionally, Japanese documents use only two families: \textit{mincho} (\明朝体) and \textit{gothic} (\ゴシック体). \textit{mincho} is used in the main text, while \textit{gothic} is used in the headings or for emphasis.

<table>
<thead>
<tr>
<th>classification (Japanese counterpart for typewriter font)</th>
<th>commands</th>
<th>family</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{\textmc{...}} \texttt{\mcfamily ...} \texttt{\mcdefault}</td>
<td>\texttt{\textmc{...}} \texttt{\mcfamily ...} \texttt{\mcdefault}</td>
<td>\texttt{\mcdefault}</td>
</tr>
<tr>
<td>\texttt{\textgt{...}} \texttt{\gtfamily ...} \texttt{\gtdefault}</td>
<td>\texttt{\textgt{...}} \texttt{\gtfamily ...} \texttt{\gtdefault}</td>
<td>\texttt{\gtdefault}</td>
</tr>
</tbody>
</table>

Here \jttdefault specifies the Japanese font family in \verb or \verbatim environment, and its default value is \mcdefault (mincho family).\footnote{When \ltjclasses classes are used, or luatexja-fontspec (or luatexja-preset) is loaded with match option, \ttfamily changes the current Japanese font family to \jttdefault. These classes and packages also redefine \jttdefault to \gtdefault (gothic family).}

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

<table>
<thead>
<tr>
<th>classification</th>
<th>family</th>
<th>\texttt{\mdseries}</th>
<th>\texttt{\bfseries}</th>
<th>scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{mincho} (\明朝体)</td>
<td>mc</td>
<td>HaranoAjiMincho-Regular</td>
<td>HaranoAjiGothic-Medium</td>
<td>0.962216</td>
</tr>
<tr>
<td>\textit{gothic} (\ゴシック体)</td>
<td>gt</td>
<td>HaranoAjiGothic-Medium</td>
<td>HaranoAjiGothic-Medium</td>
<td>0.962216</td>
</tr>
</tbody>
</table>

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

- From version 20181102.0, one can specifies \texttt{disablejfam} option at loading \LaTeX-ja. This option prevents loading a patch for \LaTeX, which are needed to support Japanese characters in math mode. Without \texttt{disablejfam} option, one can typeset Japanese characters in math mode as \$\texttt{\textasciitilde}$ (see Page 11) as before. Japanese characters in math mode are typeset by the font family \texttt{mc}.

- If you use the beamer class with the default font theme (which uses sans serif fonts) and with \LaTeX-ja, you might want to change default Japanese fonts to the gothic family. The following line changes the default Japanese font family to it:

\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, LuaTeX-ja has the counterparts of `jclasses` (standard classes in pLaTeX) and `jsclasses` (classes by Haruhiko Okumura), namely, `ltjclasses`\(^4\) and `ltjsclasses`\(^5\).

Original `jsclasses` use \texttt{\textbackslash mag} primitive to set the main document font size. However, LuaTeX does not support \texttt{\textbackslash mag} in PDF output, so `ltjsclasses` uses the \texttt{nomag*} option\(^6\) by default to set the main font size. If this causes some unexpected behavior, specify \texttt{nomag} option in \texttt{\documentclass}.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{encoding} & \textbf{family} & \textbf{series} & \textbf{shape} & \textbf{selection} \\
\hline
\textbf{Alphabetic fonts} & \texttt{\textbackslash romanencoding} & \texttt{\textbackslash romanfamily} & \texttt{\textbackslash romanseries} & \texttt{\textbackslash romanshape} & \texttt{\textbackslash userroman} \\
\textbf{Japanese fonts} & \texttt{\textbackslash kanjiencoding} & \texttt{\textbackslash kanjifamily} & \texttt{\textbackslash kanjiseries} & \texttt{\textbackslash kanjishape} & \texttt{\textbackslash usekanji} \\
both & - & - & \texttt{\fontseries} & \texttt{\fontshape*} & - \\
\textbf{auto select} & \texttt{\fontencoding} & \texttt{\fontfamily} & - & - & \texttt{\usefont} \\
\hline
\end{tabular}
\caption{Font selection system in LuaTeX-ja.}
\end{table}

\begin{itemize}
\item \texttt{\fontfamily}, \texttt{\fontseries}, and \texttt{\fontshape} try to change attributes of Japanese fonts, as well as those of alphabetic fonts. Of course, \texttt{\selectfont} is needed to select current text fonts.
\end{itemize}

Note that \texttt{\fontshape} always changes current alphabetic font shape, but it does not change current Japanese font shape if the target shape is unavailable for current Japanese encoding/family/series. For the detail, see Subsection 11.2.

\begin{itemize}
\item \texttt{\fontencoding{\langle encoding \rangle}} changes the encoding of alphabetic fonts or Japanese fonts depending on the argument. For example, \texttt{\fontencoding{\texttt{\textbackslash JY3}}} changes the encoding of Japanese fonts to JY3, and \texttt{\fontencoding{\texttt{T1}}} changes the encoding of alphabetic fonts to T1. \texttt{\fontfamily} also changes the current Japanese font family, the current alphabetic font family, or both. For the detail, see Subsection 11.2.
\end{itemize}

\begin{itemize}
\item For defining a Japanese font family, use \texttt{\DeclareKanjiFamily} instead of \texttt{\DeclareFontFamily}. (In previous version of LuaTeX-ja, using \texttt{\DeclareFontFamily} didn’t cause any problem. But this no longer applies the current version.)
\end{itemize}

\begin{itemize}
\item Defining a Japanese font shape can be done by usual \texttt{\DeclareFontShape}:
\end{itemize}

\begin{verbatim}
\DeclareFontShape{\texttt{\textbackslash JY3}}{\texttt{\textbackslash mc}}{\texttt{\textbackslash b}}{\texttt{\textbackslash n}}{\texttt{\textless \textgreater \texttt{s*HaranoAjiMincho--Bold:jfm=ujis;\textbackslash korn}\textgreater}}{}
% Harano Aji Mincho Bold
\end{verbatim}

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

\(^5\)ltjsarticle.cls, ltjsbook.cls, ltjsreport.cls, ltjsarticle.cls, ltjsbook.cls, ltjsreport.cls. The latter ltjskiyou.cls.\(^6\)

\(^6\)Same effect as the BX\texttt{\textbackslash jcls} classes (by Takayuki Yato) and \texttt{\textbackslash jsclasses}. However, these classes uses only \texttt{\textbackslash BX} code, but \texttt{\textbackslash jclasses} uses Lua code.
<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>
</tbody>
</table>

Japanese fonts are used for Japanese characters, while Alphabetic fonts are used for alphabetic characters.

### Japanese characters in math mode

- **Since \TeX supports Japanese characters in math mode,** there are sources like the following:

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

- **We (the project members of Lua\TeX-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 “\text{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 “\text{素}” is used as an identifier.

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

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

- **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 8.6.

*When Lua\TeX-ja is loaded with \texttt{disablejfam} option, one cannot write Japanese characters in math mode as \texttt{素}. At that case, one have to use \texttt{\mbox} (or \texttt{\text} in the amsmath package).*

### 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:

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

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 \texttt{\rmfamily, \textrm, \sffamily, ...} also change Japanese font family.
- **pass=⟨options⟩**

\texttt{(Obsoleted) Specify options ⟨options⟩ which will be passed to the fontspec package.}
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 \ltjsclasses.

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}

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 Lua\TeX-ja (see 8.1).

Below is an example of \jfontspec.

\begin{verbatim}
1 \jfontspec[CJKShape=NLC]{HaranoAjiMincho-Regular}
2 JIS~X~0213:2004 →辻鯵
3 \jfontspec[CJKShape=JIS1990]{HaranoAjiMincho-Regular}
4 JIS~X~0208-1990 →辻鯵
5 \jfontspec[CJKShape=JIS1978]{HaranoAjiMincho-Regular}
6 JIS~C~6226-1978 →辻鰺
\end{verbatim}

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 Subsection 13.6. The following presets are defined:

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

For example, this document loads luatexja-preset package by
\usepackage[haranoaji]{luatexja-preset}
which means that Harano Aji fonts will be used in this document.

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

Under p\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.
Table 2. Characters in predefined character range 8.

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>§ (U+00A7)</td>
<td>Section Sign</td>
</tr>
<tr>
<td>' (U+00B0)</td>
<td>Degree sign</td>
</tr>
<tr>
<td>' (U+00B4)</td>
<td>Spacing acute</td>
</tr>
<tr>
<td>× (U+00D7)</td>
<td>Multiplication sign</td>
</tr>
<tr>
<td>- (U+00F7)</td>
<td>Division Sign</td>
</tr>
</tbody>
</table>

Table 3. Unicode blocks in predefined character range 1.

<table>
<thead>
<tr>
<th>Code</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+0080–U+00FF</td>
<td>Latin-1 Supplement</td>
</tr>
<tr>
<td>U+0100–U+017F</td>
<td>Latin Extended-A</td>
</tr>
<tr>
<td>U+0250–U+02AF</td>
<td>IPA Extensions</td>
</tr>
<tr>
<td>U+0300–U+036F</td>
<td>Combining Diacritical Marks</td>
</tr>
<tr>
<td>U+0180–U+017F</td>
<td>Latin Extended-B</td>
</tr>
<tr>
<td>U+02B0–U+02FF</td>
<td>Spacing Modifier Letters</td>
</tr>
<tr>
<td>U+1E00–U+1EFF</td>
<td>Latin Extended Additional</td>
</tr>
</tbody>
</table>

4.1 Range of JAchars

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

\laptops{charrange}{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 LuaTeX-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 LuaTeX-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, 7, and 9 is JAchar.

\laptops{setparameter}{jacharrange={-1, +2, +3, -4, -5, +6, +7, -8, +9}}

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).

Default character ranges LuaTeX-ja predefines nine character ranges for convenience. They are determined from the following data:

- Blocks in Unicode 12.0.0.
- The Adobe-Japan1-UCS2 mapping between a CID Adobe-Japan1- and Unicode.
- The PXbase bundle for upTeX by Takayuki Yato.

Now we describe these nine ranges. The superscript “J” or “A” after the number shows whether each character in the range is treated as JAchar 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 8

The intersection of the upper half of ISO 8859-1 (Latin-1 Supplement) and JIS X 0208 (a basic character set for Japanese). The character list is indicated in Table 2.

Range 1

Latin characters that some of them are included in Adobe-Japan1-7. This range consists of the Unicode ranges indicated in Table 3, except characters in the range 8 above.

Range 2

Greek and Cyrillic letters. JIS X 0208 (hence most of Japanese fonts) has some of these characters.
Table 4. Unicode blocks in predefined character range 3.

<table>
<thead>
<tr>
<th>Block Range</th>
<th>Block Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+2070–U+209F</td>
<td>Superscripts and Subscripts</td>
</tr>
<tr>
<td>U+20A0–U+20CF</td>
<td>Currency Symbols</td>
</tr>
<tr>
<td>U+2100–U+214F</td>
<td>Letterlike Symbols</td>
</tr>
<tr>
<td>U+2190–U+21FF</td>
<td>Arrows</td>
</tr>
<tr>
<td>U+2200–U+22FF</td>
<td>Mathematical Operators</td>
</tr>
<tr>
<td>U+2300–U+23FF</td>
<td>Miscellaneous Technical</td>
</tr>
<tr>
<td>U+2400–U+243F</td>
<td>Control Pictures</td>
</tr>
<tr>
<td>U+2500–U+257F</td>
<td>Box Drawing</td>
</tr>
<tr>
<td>U+2580–U+259F</td>
<td>Block Elements</td>
</tr>
<tr>
<td>U+25A0–U+25FF</td>
<td>Geometric Shapes</td>
</tr>
<tr>
<td>U+2600–U+26FF</td>
<td>Miscellaneous Symbols</td>
</tr>
<tr>
<td>U+2700–U+27BF</td>
<td>Dingbats</td>
</tr>
<tr>
<td>U+2900–U+297F</td>
<td>Supplemental Arrows-A</td>
</tr>
<tr>
<td>U+2980–U+29FF</td>
<td>Misc. Math Symbols-B</td>
</tr>
</tbody>
</table>

Table 5. Characters in predefined character range 9.

<table>
<thead>
<tr>
<th>Character</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+2002</td>
<td>En space</td>
</tr>
<tr>
<td>U+200A</td>
<td>Non-breaking hyphen</td>
</tr>
<tr>
<td>U+2014</td>
<td>Em dash</td>
</tr>
<tr>
<td>U+2016</td>
<td>Double vertical line</td>
</tr>
<tr>
<td>U+2019</td>
<td>Right single quotation mark</td>
</tr>
<tr>
<td>U+201C</td>
<td>Left double quotation mark</td>
</tr>
<tr>
<td>U+201E</td>
<td>Double low-9 quotation mark</td>
</tr>
<tr>
<td>U+2021</td>
<td>Double dagger</td>
</tr>
<tr>
<td>U+2025</td>
<td>Two dot leader</td>
</tr>
<tr>
<td>U+202E</td>
<td>Per mille sign</td>
</tr>
<tr>
<td>U+2033</td>
<td>Double prime</td>
</tr>
<tr>
<td>U+203A</td>
<td>Single right-pointing angle quot.</td>
</tr>
<tr>
<td>U+203C</td>
<td>Double exclamation mark</td>
</tr>
<tr>
<td>U+203F</td>
<td>Undertie</td>
</tr>
<tr>
<td>U+2044</td>
<td>Fraction slash</td>
</tr>
<tr>
<td>U+2048</td>
<td>Question exclamation mark</td>
</tr>
<tr>
<td>U+2051</td>
<td>Two asterisks aligned vertically</td>
</tr>
</tbody>
</table>

- U+0370–U+03FF: Greek and Coptic
- U+0400–U+04FF: Cyrillic
- U+0300–U+03FF: Greek Extended

**Range 3** Miscellaneous symbols. The block list is indicated in Table 4.

**Range 9** The intersection of the “General Punctuation” block (U+2000–U+206F) and Adobe-Japan1-7 character collection. This character range characters in Table 5.

**Range 4** 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.

```latex
\defcharrange{4}{% "5af0e"10ff, "12b0e"1dff, "2440e"24ff, "27c0e"28ff, "2a00e"2aff, "2c00e"2e7f, "4d00e"4dff, "a000e"a9ff, "a900e"abff, "e000e"f8ff, "f800e"fe0f, "fe20e"fe2f, "fe70e"f3ff, "1000e"1aff, "1b170e"1ff, "1f300e"1fff, \ldots (and characters in \texttt{\textbackslash u+2000e\textbackslash u+206f} which are not in range 9)\} % non-Japanese
```

**Range 5** Surrogates and Supplementary Private Use Areas.

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

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

**Notes on U+0000–U+00FF** You should treat characters in textttU+0000–U+00FF as \texttt{ALchar}, when you use traditional 8-bit fonts, such as the marvosym package.
Table 6. Unicode blocks in predefined character range 6.

<table>
<thead>
<tr>
<th>Codepoint Range</th>
<th>Character Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+2460–U+24FF</td>
<td>Enclosed Alphanumerics</td>
</tr>
<tr>
<td>U+3000–U+30FF</td>
<td>CJK Symbols and Punctuation</td>
</tr>
<tr>
<td>U+3040–U+30FF</td>
<td>Hiragana</td>
</tr>
<tr>
<td>U+30AB–U+30FF</td>
<td>Katakana</td>
</tr>
<tr>
<td>U+3190–U+319F</td>
<td>Kanbun</td>
</tr>
<tr>
<td>U+3200–U+32FF</td>
<td>Katakana Phonetic Extensions</td>
</tr>
<tr>
<td>U+3240–U+32FF</td>
<td>Enclosed CJK Letters and Months</td>
</tr>
<tr>
<td>U+3400–U+40FF</td>
<td>CJK Compatibility</td>
</tr>
<tr>
<td>U+4F00–U+4FFF</td>
<td>CJK Unified Ideographs</td>
</tr>
<tr>
<td>U+5E30–U+64FF</td>
<td>CJK Compatibility Ideographs</td>
</tr>
<tr>
<td>U+6F00–U+70EF</td>
<td>Vertical Forms</td>
</tr>
<tr>
<td>U+FE30–U+FE4F</td>
<td>CJK Compatibility Forms</td>
</tr>
<tr>
<td>U+FE50–U+FE6F</td>
<td>Small Form Variants</td>
</tr>
<tr>
<td>U+FF00–U+FFF</td>
<td>Halfwidth and Fullwidth Forms</td>
</tr>
<tr>
<td>U+1B00–U+1BFF</td>
<td>Kana Supplement</td>
</tr>
<tr>
<td>U+1F00–U+1FFFF</td>
<td>Enclosed Alphanumeric Supp.</td>
</tr>
<tr>
<td>U+20000–U+2FFF</td>
<td>(Supp. Ideographic Plane)</td>
</tr>
<tr>
<td>U+30000–U+3FFF</td>
<td>(Tert. Ideographic Plane)</td>
</tr>
</tbody>
</table>

Table 7. Unicode blocks in predefined character range 7.

<table>
<thead>
<tr>
<th>Codepoint Range</th>
<th>Character Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>U+1100–U+11FF</td>
<td>Hangul Jamo</td>
</tr>
<tr>
<td>U+2F00–U+2FF</td>
<td>Ideographic Description Characters</td>
</tr>
<tr>
<td>U+3100–U+31BF</td>
<td>Hangul Compatibility Jamo</td>
</tr>
<tr>
<td>U+3100–U+31EF</td>
<td>CJK Strokes</td>
</tr>
<tr>
<td>U+3100–U+31EF</td>
<td>CJK Strokes</td>
</tr>
<tr>
<td>U+3190–U+31EF</td>
<td>CJK Strokes</td>
</tr>
<tr>
<td>U+4960–U+49FF</td>
<td>Hangul Jamo Extended-A</td>
</tr>
<tr>
<td>U+4960–U+49FF</td>
<td>Hangul Jamo Extended-A</td>
</tr>
</tbody>
</table>

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

To avoid such situations, the default setting of Lua\TeX-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 \ltjchar and \ltjjachar respectively, as the following example.

\begin{verbatim}
\gfamily\large % default, ALchar, JAchar
\j, \ltjchar\j, \ltjjachar\j% \% default: ALchar
\j, \ltjchar\j, \ltjjachar\j% \% default: JAchar
\end{verbatim}

4.2 kanjiskip and xkanjiskip

JAglue 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.

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

Here \zw is an internal dimension which stores fullwidth of the current Japanese font. This \zw can be used as the unit zw in \pT\X.

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

\begin{verbatim}
kanjiskip: \ltjgetparameter[kanjiskip], kanjiskip: 0.0pt plus 0.4pt minus 0.5pt,
xkanjiskip: \ltjgetparameter[xkanjiskip] xkanjiskip: 2.40553pt plus 1pt 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 kanjiskip or xkanjiskip to \maxdimen (these "ideal width" cannot be retrieved by \ltjgetparameter).
4.3 Insertion setting of $xkanjiskip$

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

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

The second argument \texttt{preonly} means that the insertion of $xkanjiskip$ is allowed before this character, but not after. the other possible values are \texttt{postonly}, \texttt{allow}, and \texttt{inhibit}.

\texttt{jaxspmode} and \texttt{alxspmode} use a same table to store the parameters on the current version. Therefore, line 1 in the code above can be rewritten as follows:

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

One can use also numbers to specify these two parameters (see Subsection 9.1).

If you want to enable/disable all insertions of $kanjiskip$ and $xkanjiskip$, set \texttt{autospacing} and \texttt{autoxspacing} parameters to \texttt{true}/\texttt{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 \LaTeX, 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, LuaTeX-ja can independently set the shifting amount of the baseline of alphabetic fonts and that of Japanese fonts.


{| Alphabetic fonts | yalbaselineshift parameter | talbaselineshift parameter |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Japanese fonts</td>
<td>yjabaselineshift parameter</td>
<td>tjabaselineshift parameter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
\ltjsetparameter{yjabaselineshift=0pt, yalbaselineshift=0pt}abcあいう
\ltjsetparameter{yjabaselineshift=5pt, yalbaselineshift=2pt}abcあいう
\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):

\begin{verbatim}
\vrule width 150pt height4.417pt depth-4.217pt%
\kern-150pt
\large テキスト
\scriptsize
\ltjsetparameter{yjabaselineshift=-1.757pt, yalbaselineshift=-1.757pt}
\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):

\begin{verbatim}
\vrule width 150pt height4.417pt depth-4.217pt%
\kern-150pt
\large テキスト
\scriptsize
\ltjsetparameter{yjabaselineshift=-1.757pt, yalbaselineshift=-1.757pt}
\end{verbatim}

Note that setting positive $yalbaselineshift$ or $talbaselineshift$ parameters does not increase the depth of one-letter syllable $p$ of \texttt{ALchar}, if its left-protrusion (\texttt{\lpcode}) and right-protrusion (\texttt{\rpcode}) 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, LuaTeX-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 \lpcode and \rpcode are both non-zero.

This problem does not apply for \vjabaselineshift nor \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),

jaxspmode, alxspmode, prebreakpenalty, postbreakpenalty and 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 “ｱ”, by hwid feature. However, the penalty inserted after it is 10 which is the postbreakpenalty of “ｱ”, not 20.

\{addjfontfeatures{CharacterWidth=Half}\showpostpena{ア}
Part II

Reference

5 \texttt{\texttt{\textbackslash catcode} in Lua\TeX-ja}

5.1 Preliminaries: \texttt{\textbackslash catcode} in p\TeX{} and up\TeX{}

In p\TeX{} and up\TeX{}, the value of \texttt{\textbackslash catcode} determines whether a Japanese character can be used in a control word. For the detail, see Table 8.

\texttt{\textbackslash 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+FFE_F (Halfwidth and Fullwidth Forms) into three subblocks, and \texttt{\textbackslash 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 \texttt{\textbackslash 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 \texttt{JAchar} or \texttt{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 \texttt{\textbackslash catcode} to 11 (enabled) or 12 (disabled), as usual.
- Whether \texttt{\textbackslash jcharwidowpenalty} can be inserted before the character is controlled by the lowermost bit of the \texttt{\textbackslash kcatcode} parameter.
- Linebreak after a \texttt{JAchar} does not produce a space.

Default setting of \texttt{\textbackslash catcode} of Unicode characters with Lua\TeX{} is slightly inconvenient for p\TeX{} users to shifting to Lua\TeX-ja, because several fullwidth characters which can be used in a control word with p\TeX{}, such as ”1” (FULLWIDTH DIGIT ONE), cannot be used in a control word with Lua\TeX{}. Hence, Lua\TeX-ja changes the \texttt{\textbackslash catcode} of some characters—which line breaking class is ”ID” (Ideographic) in UAA \#14—, to allow these characters in the control word.

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 p\TeX{}, in up\TeX{}, and in Lua\TeX-ja. Table 9 shows the difference. Except for three characters \texttt{・}, \texttt{゛}, and \texttt{゜}, 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 \url{https://github.com/h-kitagawa/kct}.

6 Directions

Lua\TeX{} supports four \Omega-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 10. The second column (\texttt{yoko} direction) is just horizontal writing, and the third column (\texttt{tate} direction) is vertical writing. The fourth column (\texttt{dtou} direction) is actually a hidden feature of p\TeX{}. We implemented this for debugging purpose. The fifth column (\texttt{utod} direction) corresponds the ”\texttt{tate} (\texttt{math}) direction” of p\TeX{}.

Directions can be changed by \texttt{\textbackslash yoko}, \texttt{\textbackslash tate}, \texttt{\textbackslash dtou}, \texttt{\textbackslash 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 \texttt{utod}, when the direction outside the math formula is \texttt{tate} (vertical writing).
Table 8. `\kcatcode` in `up\TeX`

<table>
<thead>
<tr>
<th>kcatcode</th>
<th>meaning</th>
<th>control word</th>
<th>widow penalty</th>
<th>linebreak</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>non-cjk</td>
<td>(treated as usual <code>\TeX</code>)</td>
<td>Y</td>
<td>ignored</td>
</tr>
<tr>
<td>16</td>
<td>kanji</td>
<td>Y</td>
<td>Y</td>
<td>ignored</td>
</tr>
<tr>
<td>17</td>
<td>kana</td>
<td>Y</td>
<td>Y</td>
<td>ignored</td>
</tr>
<tr>
<td>18</td>
<td>other</td>
<td>N</td>
<td>N</td>
<td>ignored</td>
</tr>
<tr>
<td>19</td>
<td>hangul</td>
<td>Y</td>
<td>Y</td>
<td>space</td>
</tr>
</tbody>
</table>

Table 9. Difference of the set of non-kanji JIS X 0208 characters which can be used in a control word

<table>
<thead>
<tr>
<th>row</th>
<th>col.</th>
<th>p\TeX</th>
<th>up\TeX</th>
<th>Lua\TeX-ja</th>
</tr>
</thead>
<tbody>
<tr>
<td>(U+30FB)</td>
<td>1</td>
<td>6</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+309B)</td>
<td>1</td>
<td>11</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+309C)</td>
<td>1</td>
<td>12</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+FF4B)</td>
<td>1</td>
<td>14</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+FF3E)</td>
<td>1</td>
<td>16</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+FFE3)</td>
<td>1</td>
<td>17</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+FF3F)</td>
<td>1</td>
<td>18</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+30FB)</td>
<td>1</td>
<td>19</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+30FE)</td>
<td>1</td>
<td>20</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+309D)</td>
<td>1</td>
<td>21</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+309E)</td>
<td>1</td>
<td>22</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+3083)</td>
<td>1</td>
<td>23</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+EFDB)</td>
<td>1</td>
<td>24</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>(U+3085)</td>
<td>1</td>
<td>25</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+3086)</td>
<td>1</td>
<td>26</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+3087)</td>
<td>1</td>
<td>27</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(U+30FC)</td>
<td>1</td>
<td>28</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

6.1 Boxes in different direction

As in p\TeX, one can use boxes of different direction in one document. The below is an example.

```
\hbox{ここは横組}
\hbox{\tate}\hbox{縦組}\hbox{の中}\hbox{に}\hbox{横組の内容}\hbox{を}\hbox{挿入する}\hbox{ここは横組}\hbox{また横組に戻る}
```

Table 11 shows how a box is arranged when the direction inside the box and that outside the box differ.

\textbf{\vd and direction} In p\TeX, `\vd`, `\ht`, `\dp` means the dimensions of a box register \textit{with respect to the current direction}. This means that the value of `\vd` etc. might differ when the current direction is different, even if `\box0` stores the same box. However, this no longer applies in Lua\TeX-ja.
Table 10. Directions supported by LuaLaTeX-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

銀は、Ag

(Notation used in Ω)

TLT RTR, RTT LBL RTR

To access box dimensions with respect to current direction, one has to use the following commands instead of \wd etc.

\ltjgetwd(num), \ltjgetht(num), \ltjgetdp(num)

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

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

The following is an example.

\parindent0pt
1 \parindent0pt
2 \setbox0=\hbox{yo\koく\くみ}
3 \hboxsep=8mm\hbox{\copy0}
4 \hbox{\hspace{20mm}}
5 \yoko \the\ltjgetwd32767, \(
6 \the\ltjgetht32767, \the\ltjgetdp32767.\)
7 \yoko YOKO \the\ltjgetwd32767, \(
8 \the\ltjgetht32767, \the\ltjgetdp32767.\)
9 \tate TATE \the\ltjgetwd32767, \(
10 \the\ltjgetht32767, \the\ltjgetdp32767.\)
11 \dtou DTOU \the\ltjgetwd32767, \(
12 \the\ltjgetht32767, \the\ltjgetdp32767.\)

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

These commands set the dimension of \box(num). One does not need to group the argument (num); 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 direction parameter returns the current direction, and the boxdir parameter (with the argument (num)) returns the direction of a box register \box(num). The returned value of these parameters are a 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>11</td>
<td>11</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 11. 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>$W_t = h_t + d_t$</td>
<td>$W_D = h_D + d_D$</td>
</tr>
<tr>
<td>$H_y = w_y$</td>
<td>$H_t = w_t/2$</td>
<td>$H_D = h_D$</td>
</tr>
<tr>
<td>$D_y = 0$ pt</td>
<td>$D_t = w_t/2$</td>
<td>$D_D = h_D$</td>
</tr>
</tbody>
</table>

Redefined primitives by LuaTeX-ja

The following primitives are redefined by LuaTeX-ja (using \protected\def), for supporting Japanese typesetting and multiple directions:

```
/ \unhbox\langle num\rangle, \unvbox\langle num\rangle, \unhcopy\langle num\rangle, \unvcopy\langle num\rangle
\vadjust{\langle material\rangle}
\insert\langle number\rangle{\langle material\rangle}
\lastbox
\raise\langle dimen\rangle{\langle box\rangle}, \lower\langle dimen\rangle{\langle box\rangle}, \moveleft\langle dimen\rangle{\langle box\rangle}, \moveright\langle dimen\rangle{\langle box\rangle},
\split\langle number\rangle to\langle dimen\rangle, \vcenter{\langle material\rangle}
```
Figure 1. Redefining \foreach\vadjust primitive by \LuTeX-ja

\makeatletter
\def\ltj@stop@overwrite@primitive{\insert\vadjust\slash\unhbox\vcenter\fontseries}
\makeatother
%% Keep the meaning of \insert, \vadjust, \slash, \unhbox and \vcenter.
%% \fontseries will still be redefined by \LuTeX-ja, because it is not primitive.
\usepackage{luatexja}
...
\usepackage{breqn}
...
\makeatletter
\ltj@overwrite@primitive\expandafter{\insert\vadjust\slash\unhbox\vcenter}
\makeatother
%% Redefine \insert, \vadjust, \slash, \unhbox and \vcenter.

Figure 2. \ltj@stop@overwrite@primitive and \ltj@overwrite@primitive

On each primitive \langle primitive\rangle in the list above, its meaning just before loading \LuTeX-ja is backed up into \ltj@@orig@\langle primitive\rangle, and the meaning after redefinition by \LuTeX-ja is stored in \ltj@@\langle primitive\rangle. For example, Figure 1 shows the situation of \vadjust primitive.

7.1 Suppressing redefinitions

Sometimes redefining primitives by \LuTeX-ja causes a problem. For example, the \breqn package (v0.98k) assumes that \vadjust and \insert have their primitive meanings. So, this package cannot by loaded after \LuTeX-ja by default.

\LuTeX-ja version 20210517.0 has features for that problem. Namely:

- Primitives which is listed in \ltj@stop@overwrite@primitive are retain their meanings at just before loading \LuTeX-ja.
- After loading \LuTeX-ja, one can specify primitives to \ltj@overwrite@primitive, to redefine them by \LuTeX-ja.

See Figure 2 for an example.

8 Font Metric and Japanese Font

8.1 \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. \LuTeX-ja automatically loads \luaotfload package, so TrueType/OpenType fonts with features can be used for Japanese fonts:

\jfont\tradmc={IPAexMincho:script=latn;\%
+trad;-kern;jfm=ujis} at 14pt
\tradmc 当／體／醫／區

It is required to specify a (horizontal) JFM in at each calling of \jfont. A JFM is a Lua script which contains measurements of characters and glues/kerns that are automatically inserted for Japanese typesetting. The structure of JFM will be described in the next subsection.
Figure 3. Example of `jfmvar` key

Table 12. Differences between horizontal JFMs shipped with Lua\TeX-ja

![Table 12](image)

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

■ Specifying JFM

The general scheme for specifying a JFM is the following:

\begin{verbatim}
\jfont⟨\jfont_cs⟩=...;jfm=⟨JFM name⟩]/{⟨JFM features⟩};...;[jfmvar=⟨identifier⟩];...
\end{verbatim}

⟨JFM name⟩ The name of a (horizontal) JFM. Lua\TeX-ja searches and loads jfm-(⟨JFM name⟩).lua.

⟨JFM features⟩ An optional comma-separated list of JFM options. Enclosing braces (⟨⟩) are optional, but this does not escape any characters. The contents of this list can be accessed by a table \texttt{luatexja.jfont.jfm\_feature} from a JFM, at its loading. See Figure 4 for an example.

Note that any JFM files which is shipped with Lua\TeX-ja does not use this feature.

⟨identifier⟩ An optional string.

Lua\TeX-ja “does not distinguish” two Japanese fonts which uses same JFM and are the same size. Here “uses same JFM” means that all of ⟨JFM name⟩, ⟨JFM features⟩ and ⟨identifier⟩ of two fonts agree. For example, The first ⟨⟩ and ⟨⟩ in Figure 3 are typeset in different real fonts. However, because they use the same JFMs and their size are same, Lua\TeX-ja inserts penalties, glues and kerns as if these two character are typeset in a same font. Namely, the glue between these characters is halfwidth, as in ⟨⟩. However, this does not applies with ⟨F⟩ and ⟨H⟩ in Figure 3, because their ⟨identifier⟩ are different.

■ Horizontal JFMs

The following horizontal JFMs are shipped with Lua\TeX-ja.

\texttt{jfm-ujis.lua} A standard horizontal JFM of Lua\TeX-ja. This file is based on \texttt{upnmlminr-h.tfm}, a metric for UTF/OTF package that is used in up\TeX. When you are going to use the \texttt{luatexja-otf} package, you should use this JFM.
### Using kerning information in a font

Some fonts have information for inter-glyph spacing. LuaTeX-ja 20140324.0 or later treats kerning spaces like an italic correction; any glue and/or kern from the JFM and a kerning space from the font can coexist. See Figure 5 for detail.

At version 20220411.0, defaults Japanese fonts which are defined at the loading of LuaTeX-ja, \texttt{ltjclasses}, and \texttt{ltjsclasses} do not insert font-derived kerning spaces by default. This is because standard JFMs do not expect font-derived kerning spaces between Japanese characters.

Also note that in \texttt{\setmainjfont} etc. which are provided by luatexja-fontspec package, kerning option is set \texttt{off} (\texttt{Kerning=Off}) by default. This means the following two lines have the same meaning:

\begin{verbatim}
  \setmainjfont{HaranoAjiMincho-Regular}
  \setmainjfont[Kerning=Off]{HaranoAjiMincho-Regular}
\end{verbatim}

### extend and slant

The following setting can be specified as OpenType font features:

- \texttt{extend} (\texttt{extend}) expand the font horizontally by \texttt{(extend)}.
- \texttt{slant} (\texttt{slant}) slant the font.

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

\begin{verbatim}
  \jfont{E=HaranoAjiMincho-Regular:extend=1.5;ujis;-kern}
  \jfont{S=HaranoAjiMincho-Regular:slant=1;ujis;-kern}
\end{verbatim}

![Figure 4. Example of JFM features](image-url)
\texttt{\newcommand\test{\vrule \text{ダイナミックダイクマ}\vrule}}
\texttt{\jfont\KMFW = HaranoAjiMincho-Regular:jfm=prop;-kern at 17.28pt}
\texttt{\jfont\KMFK = HaranoAjiMincho-Regular:jfm=prop at 17.28pt % kern is activated}
\texttt{\jfont\KMPW = HaranoAjiMincho-Regular:jfm=prop;script=dl;+palt;-kern at 17.28pt}
\texttt{\jfont\KMPK = HaranoAjiMincho-Regular:jfm=prop;script=dl;+palt;+kern at 17.28pt}
\begin{multicols}{2}
\begin{align}
\text{\begin{multicols}{2}
\leavevmode
\ltjsetparameter{kanjiskip=0pt plus 3\zw}
\vrule\hbox to 15\zw{あ「い」う,えお}\vrule
\jfont\G=HaranoAjiMincho-Regular%:jfm=ujisv;-ltjksp at \zw
\G\leavevmode%\vrule\hbox to 15\zw{あ「い」う,えお}\vrule
\end{multicols}}
\end{align}
\end{multicols}

\begin{multicols}{2}
\begin{align}
\text{Figure 5. Kerning information and \texttt{kanjiskip}}
\end{align}
\end{multicols}

\begin{multicols}{2}
\begin{align}
\text{Figure 6. \texttt{ltjksp 'feature'}}
\end{align}
\end{multicols}

\begin{itemize}
\item ltjksp “feature” \texttt{kanjiskip_natural, kanjiskip_stretch, kanjiskip_shrink} keys (Page ??) makes the Lua\TeX{}-ja inserts not only a glue which is specified by a JFM, and also the natural width/stretch part/shrink part of \texttt{kanjiskip}. This functionality is disabled by -ltjksp specification, as shown in Figure 6.
\end{itemize}

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

\begin{itemize}
\item OpenType Feature vrt2\footnote{If the font does not define the vrt2 feature, vert is used instead.} is automatically activated, unless vert and/or vrt2 features are explicitly activated or deactivated (as the second line in the example below).

\begin{align}
\texttt{\tfont}\S=\text{HaranoAjiMincho-Regular:jfm=ujisv} % \texttt{vrt2} is automatically activated
\texttt{\tfont}\T=\text{HaranoAjiMincho-Regular:jfm=ujisv;\texttt{-vert}} % \texttt{vert} and \texttt{vrt2} are not activated
\texttt{\tfont}\U=\text{file:ipaexm.ttf:jfm=ujisv}
\end{align}

% \texttt{vert} is automatically activated, since this font does not have \texttt{vrt2}

\item 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, Lua\TeX{}-ja performs all replacements which is defined in vert feature for some scripts for some languages.

\end{itemize}
• 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.

• \tfont uses a vertical JFM instead of a horizontal JFM. LuaTeX-ja ships following vertical JFMs:

  jfm-ujisv.lua A standard vertical JFM in LuaTeX-ja. This JFM is based on upnmlminr-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.

• If vert and/or vrt2 features are activated, one can specify jpotf to additional substitutions. By default, it substitutes ideographic comma/period for fullwidth comma/period, and double prime quotation marks for double quotation marks (See Figure 7). One can customize substitutions by lua function luatexja.jfont.register_vert_replace (see Japanese version of this manual).

8.3 Default Japanese fonts and JFMs

If following commands are defined at loading LuaTeX-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@stdtatejf The default JFM for vertical direction.

For example,
\def\ltj@stdmcfont{IPAMincho} \def\ltj@stdgtfont{IPAGothic}

makes that IPA Mincho and IPA Gothic will be used as default Japanese fonts, instead of Harano Aji fonts.

This feature is intended for classes which use special JFMs\footnote{This is because commands has @ in their names.}. It is recommended to use \luatexja-preset or \luatexja-fontspec package to select standard fonts in ordinary \TeX sources.

For compatibility with earlier versions, LuaTeX-ja reads luatexja.cfg automatically if it is found by LuaTeX. One should not overuse this luatexja.cfg; it will overwrite the definition of \ltj@stdmcfont and others.
8.4 Prefix psft

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

OpenType font features, such as "+jp9\textgamma", 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 psft prefix, because they are only simple linear transformations.

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

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

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

At present, LuaTeX-ja supports only 5 values written in the sample code above. Specifying other values, e.g.,

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

produces the following error:

! Package luatexja Error: bad cid key 'Adobe-Japan2'.

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

\begin{verbatim}
\end{verbatim}

\begin{verbatim}
\end{verbatim}

I couldn't find any non-embedded font information for the CID 'Adobe-Japan2'. For now, I'll use 'Adobe-Japan1-7'.

Please contact the LuaTeX-ja project team.

?

8.5 Structure of a JFM file

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

\begin{verbatim}
luatexja.jfont.define_jfm { ... }
\end{verbatim}

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.

version=(version) (optional, default value is 1)

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

dir=(direction) (required)

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

zw=(length) (required)

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

**kanjiskip**={⟨natural⟩, ⟨stretch⟩, ⟨shrink⟩} (optional)

This field specifies the "ideal" amount of **kanjiskip**. As noted in Subsection 4.2, if the parameter **kanjiskip** is \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.

**xkanjiskip**={⟨natural⟩, ⟨stretch⟩, ⟨shrink⟩} (optional)

Like the **kanjiskip** field, this field specifies the "ideal" amount of **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:

**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.)

**width**=⟨length⟩, **height**=⟨length⟩, **depth**=⟨length⟩, **italic**=⟨length⟩ (required)

Specify the width of characters in character class \( i \), the height, the depth and the amount of italic correction. All characters in character class \( i \) are regarded that its width, height, and depth are as values of these fields. The default values are shown in Table 13.

**left**=⟨length⟩, **down**=⟨length⟩, **align**=⟨align⟩

These fields are for adjusting the position of the "real" glyph. Legal values of **align** field are 'left', 'middle', and 'right'. If one of these 3 fields are omitted, left and down are treated as 0, and align field is treated as 'left'. The effects of these 3 fields are indicated in Figure 8 and Figure 9.

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'.

**kern**={[\( j \)]={⟨kern⟩}, [\( j' \)]={⟨kern⟩}, \{ratio=⟨ratio⟩\}], ...}

**glue**={[\( j \)]={⟨width⟩, ⟨stretch⟩, ⟨shrink⟩, \{ratio=⟨ratio⟩, ...\}], ...}

Specifies the amount of kern or glue which will be inserted between characters in character class \( i \) and those in character class \( j \).

 ⟨ratio⟩ specifies how much the glue is originated in the “right” character. It is a real number between 0 and 1, and treated as 0.5 if omitted. For example, The width of a glue between an ideographic...
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.

**Figure 8. The position of the real glyph (horizontal Japanese fonts)**

**Figure 9. The position of the real glyph (vertical Japanese fonts)**

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 \((\text{ratio})\) to \(0.25/(0.5 + 0.25) = 1/3\).

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

- **priority=⟨priority⟩** An integer in \([-4, 3]\) (treated as 0 if omitted), or a pair of these integers \{(⟨stretch⟩), (⟨shrink⟩)} (version 2 or later). This is used only in line adjustment with priority by luatexja-adjust (see Subsection 13.3). Higher value means the glue is easy to stretch, and is also easy to shrink.

- **kanjiskip_natural=⟨num⟩, kanjiskip_stretch=⟨num⟩, kanjiskip_shrink=⟨num⟩**

  These keys specifies the amount of the natural width of \(\text{kanjiskip}\) (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 \(\text{jfm-ujis.1ua}\), 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 \(\text{kanjiskip}\) times the value of \(\text{kanjiskip_stretch}\) key (1 in this case).
- Between an ideographic closing brackets (including the ideographic comma “，”) and an ordinal letter (including an \textbf{ALChar} “ł”), we have the same glue. Again, this glue also can be stretched as much as the stretch part of \(\text{kanjiskip}\) times the value of \(\text{kanjiskip_stretch}\) 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 \(\text{kanjiskip}\).

Hence we have the following result:
\leavevmode\let\V=\vrule
\ltjsetparameter{kanjiskip=0pt plus 5\zw}
\ltjsetparameter{xkanjiskip=0pt plus 0.5\zw}
\V\hbox spread 7\zw{あ「い」う,えお」\V}
\vrule\hbox{あ「い」う,えお」\V\par}
\ltjsetparameter{kanjiskip=0pt minus \zw}
\V\hbox spread -2.5\zw{あ「い」う,えお」\V\par}

end\_stretch=\{kern\}, end\_shrink=\{kern\} (optional, version 1 only)
end\_adjust=\{(kern), \{kern\}, ...\} (optional, version 2 or later)
round\_threshold=\{float\} (optional, version 3 or later, only available in character class 0)

\begin{itemize}
  \item \textbf{Character to character classes} We explain how the character class of a character is determined, using \texttt{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 = { '。', 'ヒ' },
    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=IPAexMincho:jfm=test;+hwid
  \setbox0\hbox{\a び漢}
  \the\wd0
  \end{verbatim}

  Now we look why the above source outputs 15 pt.

  1. The character “び” is converted to its half width form “び” by \texttt{hwid} feature.
  2. According to the JFM, the character class of “び” 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 $\hbox$ equals to 15 pt.

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

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

  \begin{verbatim}
  \jfont\a=HaranoAjiMincho-Regular:jfm=test;+vert
  \a 漢。\inhibitglue 漢
  \end{verbatim}

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

  1. As the case of “び”, the ideographic full stop “。” is converted to its vertical form “○” (U+FE12) by \texttt{vert} feature.
  2. The character class of “○”, according to the JFM is zero.
  3. However, \TeX\-ja remembers that this “○” is obtained from “。” by font features. The character class of “。” is \textit{non-zero value}, namely, 2000.
  4. Hence the ideographic full stop “。” in above belongs the character class 2000.
\end{itemize}
Table 14. Commands for Japanese math fonts

<table>
<thead>
<tr>
<th>Japanese fonts</th>
<th>alphabetic fonts</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\jfam \in [0, 256)$</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>

■ Imaginary characters As described before, one can specify several “imaginary characters” in chars field. The most of these characters are regarded as the characters of class 0 in \p\TeX. As a result, Lua\TeX-ja can control typesetting finer than \p\TeX. The following is the list of imaginary characters:

- **'boxbdd'**
  The beginning/ending of a hbox, and the beginning of a noindented (i.e., began by \noindent) paragraph. If a hbox $b$ begins (resp. ends) a glue or kern between this "charater" and a J\textsc{Achar}, J\textsc{Aglue} won’t be inserted before(resp. after) the hbox $b$. \textsc{kanjiskip} and \textsc{xkanjiskip} around a hbox.

- **'parbdd'**
  The beginning of an (indented) paragraph.

- **'jcharbdd'**
  A boundary between J\textsc{Achar} and anything else.

- **'alchar', 'nox_alchar'**
  (version 3 or later) A boundary between J\textsc{Achar} and A\textsc{Lchar}.

- **'glue'**
  (version 3 or later) A boundary between J\textsc{Achar}, and, a glue or kern.

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

■ Porting JFM from \p\TeX See Japanese version of this manual.

8.6 Math font family

\TeX handles fonts in math formulas by 16 font families\(^{10}\), and each family has three fonts: \textfont, \scriptfont and \scriptscriptfont.

Lua\TeX-ja’s handling of Japanese fonts in math formulas is similar; Table 14 shows counterparts to \TeX’s primitives for math font families. There is no relation between the value of $\fam$ and that of $\jfam$; with appropriate settings, one can set both $\fam$ and $\jfam$ to the same value. Here (jfont_cs) in the argument of \jatextfont etc. is a control sequence which is defined by \jfont, i.e., a horizontal Japanese font.

8.7 Callbacks

Lua\TeX-ja also has several callbacks. These callbacks can be accessed via luatexbase.add_to_callback function and so on, as other callbacks.

\texttt{luatexja.load\_jfm} callback

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

```lua
function (<table> jfm_info, <string> jfm_name)
  return <table> new_jfm_info
end
```

The argument \texttt{\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 ltj\texttt{article} class, with forcefully assigning character class 0 to 'parbdd' in the JFM \texttt{\jfm\_min.lua}.

\(^{10}\)Omega, Aleph, Lua\TeX and \v{u} (u)p\TeX can handles 256 families, but an external package is needed to support this in \plain\TeX and \La\TeX.
luatexja.define_font 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 luatexja.define_font callback is called just when new Japanese font is loaded.

```lua
function (<table> jfont_info, <number> font_number)
  return <table> new_jfont_info
end
```

jfont_info has the following fields, which may not overwritten by a user:

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

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

9 Parameters

9.1 \texttt{\textbackslash ltjsetparameter}

As described before, \texttt{\textbackslash ltjsetparameter} and \texttt{\textbackslash 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 p\TeX{} (e.g., `\prebreakpenalty` =10000) is the position of `hpack_filter` callback in the source of LuaTEX, see Section 14.

\texttt{\textbackslash ltjsetparameter} and \texttt{\textbackslash 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; \texttt{\textbackslash ltjsetparameter} does a local assignment and \texttt{\textbackslash ltjglobalsetparameter} does a global one by default. They also obey the value of \texttt{\globaldefs}, like other assignments.

The following is the list of parameters which can be specified by the \texttt{\textbackslash ltjsetparameter} command. [\texttt{\cs}] indicates the counterpart in p\TeX{}, 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.

\texttt{jcharwidowpenalty}=(penalty)∗ [\texttt{jcharwidowpenalty}]  
Penalty value for suppressing orphans. This penalty is inserted just after the last \texttt{J\textchar9} which is not regarded as a (Japanese) punctuation mark.

\texttt{kcatcode}={⟨char\_code⟩,⟨natural\_number⟩}∗  
An additional attributes which each character whose character code is ⟨char\_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 \texttt{jcharwidowpenalty} above).

\texttt{prebreakpenalty}={⟨char\_code⟩,⟨penalty⟩}∗ [\texttt{prebreakpenalty}]  
Set a penalty which is inserted automatically before the character ⟨char\_code⟩, to prevent a line starts from this character. For example, a line cannot started with one of closing brackets “〗”, so LuaTEX-ja sets

\texttt{\textbackslash ltjsetparameter{prebreakpenalty={`}},10000}}

by default.

p\TeX{} has following restrictions on \texttt{prebreakpenalty} and \texttt{postbreakpenalty}, but they don’t exist in LuaTEX-ja:

- Both \texttt{prebreakpenalty} and \texttt{postbreakpenalty} cannot be set for the same character.
- We can set \texttt{prebreakpenalty} and \texttt{postbreakpenalty} up to 256 characters.
postbreakpenalty =\{(char\_code),\{penalty\}\}* \[postbreakpenalty\]

Set a penalty which is inserted automatically after the character \(\text{char\_code}\), to prevent a line ends with this character.

\texttt{jatextfont} =\{(fam),\{jfont\_cs\}\} \[jatextfont\]
\texttt{jascriptfont} =\{(fam),\{jfont\_cs\}\} \[jascriptfont\]
\texttt{jascriptscriptfont} =\{(fam),\{jfont\_cs\}\} \[jascriptscriptfont\]
\texttt{yjabaselineshift} =\{dimen\} \[yjabaselineshift\]
\texttt{yalbaselineshift} =\{dimen\} \[yalbaselineshift\]
\texttt{tjabaselineshift} =\{dimen\} \[tjabaselineshift\]
\texttt{talbaselineshift} =\{dimen\} \[talbaselineshift\]
\texttt{jaxspmode} =\{(char\_code),\{mode\}\}* \[jaxspmode\]

Set whether inserting \texttt{xkanjiskip} is allowed before/after a \texttt{JChar} whose character code is \(\text{char\_code}\). The followings are allowed for \(\text{mode}\):

\begin{itemize}
\item \texttt{0, inhibit} Insertion of \texttt{xkanjiskip} is inhibited before the character, nor after the character.
\item \texttt{1, preonly} Insertion of \texttt{xkanjiskip} is allowed before the character, but not after.
\item \texttt{2, postonly} Insertion of \texttt{xkanjiskip} is allowed after the character, but not before.
\item \texttt{3, allow} Insertion of \texttt{xkanjiskip} is allowed both before the character and after the character. This is the default value.
\end{itemize}

This parameter is similar to the \texttt{x\_inhibitxp\_code} primitive of \TeX, but not compatible with \texttt{x\_inhibitxp\_code}.

\texttt{alxspmode} =\{(char\_code),\{mode\}\}* \[alxspmode\]

Set whether inserting \texttt{xkanjiskip} is allowed before/after a \texttt{ALChar} whose character code is \(\text{char\_code}\). The followings are allowed for \(\text{mode}\):

\begin{itemize}
\item \texttt{0, inhibit} Insertion of \texttt{xkanjiskip} is inhibited before the character, nor after the character.
\item \texttt{1, preonly} Insertion of \texttt{xkanjiskip} is allowed before the character, but not after.
\item \texttt{2, postonly} Insertion of \texttt{xkanjiskip} is allowed after the character, but not before.
\item \texttt{3, allow} Insertion of \texttt{xkanjiskip} is allowed both before the character and after the character. This is the default value.
\end{itemize}

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

\texttt{autospaning} =\{bool\} \[autospaning\]
\texttt{autoxspaning} =\{bool\} \[autoxspaning\]
\texttt{kanjiskip} =\{skip\}* \[kanjiskip\]

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

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 8.5).

\texttt{xkanjiskip} =\{skip\}* \[xkanjiskip\]

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

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 8.5).
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)
kansujichar = {(digit), char_code} [\kansujichar]
direction = (dir) (always local)
Assigning to this parameter has the same effect as \yoko (if \langle dir \rangle = 4), \tate (if \langle dir \rangle = 3), \dtou (if \langle dir \rangle = 1) or \utod (if \langle dir \rangle = 11). If the argument \langle dir \rangle is not one of 4, 3, 1 nor 11, the behavior of this assignment is undefined.

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

\ltjgetparameter{differentjfm}, \ltjgetparameter{autospacing}, \ltjgetparameter{kanjiskip}, \ltjgetparameter{prebreakpenalty}{``}) paverage, 1, 0.0pt plus 0.4pt minus 0.5pt, 10000.

The return value of \ltjgetparameter is always a string, which is outputted by 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

  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.

• If first argument is one of the following, an additional argument—a character code, for example—is needed.

  \ltjgetparameter{kjacharrange}{\langle range \rangle} returns 0 if "characters which belong to the character range \langle range \rangle are JAChar", 1 if "... are ALChar". Although there is no character range ^-1, specifying ^-1 to \langle range \rangle does not cause an error (returns 1).

• For an integer (digit) between 0 and 9, \ltjgetparameter{kansujichar}{\langle digit \rangle} returns the character code of the result of \kansuji\langle digit\rangle.

• \ltjgetparameter{adjustdir} returns a integer which represents the direction of the surrounding vertical list. As direction, the return value 1 means \dtou direction, 3 means \tate direction (vertical typesetting), and 4 means \yoko direction (horizontal typesetting).

• For an integer (register) between 0 and 65535, \ltjgetparameter{boxdir}{\langle register \rangle} returns the direction of \box\langle register \rangle. If this box register is void, the returned value is zero.

• The following parameter names cannot be specified in \ltjgetparameter.

  jatextfont, jascriptfont, jascriptscriptfont, jacharrange

35
\texttt{\textbackslash getparameter\{chartorange\}\{\texttt{char\_code}\}} returns the range number which \texttt{char\_code} belongs to (although there is no parameter named “chartorange”). If \texttt{char\_code} is between 0 and 127, this \texttt{char\_code} does not belong to any character range. In this case, \texttt{\textbackslash getparameter\{chartorange\}\{\texttt{char\_code}\}} returns \texttt{-1}.

Hence, one can know whether \texttt{char\_code} is JA\texttt{char} or not by the following:

\begin{verbatim}
\textbackslash getparameter\{jacharrange\}\{\textbackslash getparameter\{chartorange\}\{\texttt{char\_code}\}\}
\end{verbatim}

% 0 if JA\texttt{char}, 1 if AL\texttt{char}

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

\begin{verbatim}
\ifdim\textbackslash getparameter\{kanjiskip\}>\z@ ... \fi
\ifdim\textbackslash getparameter\{xkanjiskip\}>\z@
\fi
\end{verbatim}

The correct way is using a temporary register.

\begin{verbatim}
\let\tempskipa=\textbackslash getparameter\{kanjiskip\}
\ifdim\tempskipa>\z@
\fi\tempskipa=\textbackslash getparameter\{xkanjiskip\}
\ifdim\tempskipa>\z@
\fi
\end{verbatim}

9.3 Alternative Commands to \texttt{\textbackslash setparameter}

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

\begin{itemize}
  \item \textbf{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{\textbackslash setparameter}:

\begin{verbatim}
\textbackslash setparameter\{kanjiskip=0\zw plus .1\zw minus .01\zw\}
\let\tempskipa=\textbackslash getparameter\{xkanjiskip\}
\ifdim\tempskipa>\z@
\fi\tempskipa=\textbackslash getparameter\{xkanjiskip\}
\if@slide
\textbackslash setxkanjiskip.1em
\else
\textbackslash setxkanjiskip.25em plus .15em minus .06em
\fi
\fi
\end{verbatim}

Hence, Lua\TeX-ja defines more primitive commands, namely \texttt{\textbackslash @setpar\global}, \texttt{\textbackslash setkanjiskip}, and \texttt{\textbackslash setxkanjiskip}. Here

\begin{verbatim}
\textbackslash @setpar\global\textbackslash setkanjiskip 10pt
\end{verbatim}

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

\begin{verbatim}
\textbackslash setkanjiskip{\z@ plus .1\zw minus .01\zw}
\let\tempskipa=\textbackslash getparameter\{xkanjiskip\}
\ifdim\tempskipa>\z@
\fi\tempskipa=\textbackslash getparameter\{xkanjiskip\}.1em
\else
\tempskipa=\textbackslash getparameter\{xkanjiskip\}.25em plus .15em minus .06em
\fi
\fi
\end{verbatim}

Note that using \texttt{\textbackslash setkanjiskip} or \texttt{\textbackslash setxkanjiskip} alone, that is, without executing \texttt{\textbackslash @setpar\global} in advance, is \textit{not} supported.
10 Other Commands for plain \TeX and \LaTeXe

10.1 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 \\kansuji converts an integer into its Chinese numerals.

\kuten, \jis, \euc, \\euj, \ucs, \kansuji

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

\begin{verbatim}
\newcount\hoge
\hoge=2423
\text{\kutsu, \jis, \euc, \euj, \ucs, \kansuji}
\end{verbatim}

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

\begin{verbatim}
\jsetparameter{kansujichar={1,`壹}}
\jsetparameter{kansujichar={7,`漆}}
\jsetparameter{kansujichar={0,`零}}
\kansuji
\end{verbatim}

10.2 \textbackslash inhibitglue, \textbackslash disinhibitglue

\textbackslash inhibitglue suppresses the insertion of a glue/kern specified in JFM at the place. The following is an example, using a special JFM that there will be a glue between the beginning of a box and \texttt{あ}, and also between \texttt{あ} and \texttt{ウ}.

\begin{verbatim}
\jfont{g=HaranoAjiMincho-Regular:jfm=test} \g
\fbox{\hbox{あウあ}} \inhibitglue
\inhibitglue\par
\inhibitglue
\inhibitglue\par
\par
\inhibitglue\par
\par\hrule
\end{verbatim}

With the help of this example, we remark the specification of \textbackslash inhibitglue:

- The call of \textbackslash inhibitglue in the (internal) vertical mode is simply ignored.

- \textbackslash inhibitglue does not suppress \texttt{kanjiskip} or \texttt{\xkanjiskip}.

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

- The call of \textbackslash inhibitglue in math mode is just ignored.

\textbackslash disinhibitglue suppresses the effect of \textbackslash inhibitglue. In other words, \textbackslash disinhibitglue allows the insertion of a glue/kern specified by JFM. If \textbackslash inhibitglue and \textbackslash disinhibitglue both specified at the same time, the latest one is effective. This command is added in the version 20201224.0.

Note that \textbackslash disinhibitglue also cancels ligatures and kernings.

10.3 \textbackslash ltjfakeboxbdd, \textbackslash ltjfakeparbegin

Sometimes \texttt{\parbdd} and \texttt{\bboxdd} specifications look like “fail”, especially in paragraphs inside list environments. This is because \texttt{\everypar} inserts some nodes such as boxes and kerns, so the “first letter” in a paragraph is in fact not the first letter.
\texttt{\textbackslash ltjfakeboxbdd} and \texttt{\textbackslash ltjfakeparbegin} primitives resolve this situation.

- \texttt{\textbackslash ltjfakeparbegin} creates a node which indicates “beginning of an indented paragraph” to the insertion process of JAglue.
- \texttt{\textbackslash 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:

10.4 \texttt{\textbackslash insertxkanjiskip}, \texttt{\textbackslash insertkanjiskip}

There are some situations which one wants to insert \texttt{xkanjiskip} manually. A simple approach is to use \texttt{\textbackslash hskip\textbackslash ltjgetparameter[xkanjiskip]}, but this approach has several weak points. To cope with these weak points, LuaTeX-ja defines a command \texttt{\textbackslash insertxkanjiskip} which inserts \texttt{xkanjiskip} glue manually, from the version 20201224.0.

- “\texttt{\textbackslash insertxkanjiskip}” (without any keyword) uses the value of \texttt{xkanjiskip} at the place.
- “\texttt{\textbackslash insertxkanjiskip late}” (with “late” keyword) uses the value of \texttt{xkanjiskip} at the end of a paragraph/hbox.

See the example below.

There is a similar command \texttt{\textbackslash insertkanjiskip} (\texttt{kanjiskip} instead of \texttt{xkanjiskip}) is also defined. Note that any shorthand form of \texttt{\textbackslash insert[x]kanjiskip} are not defined by LuaTeX-ja.

10.5 \texttt{\textbackslash ltjdeclarealtfont}

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

\texttt{\textbackslash ltjdeclarealtfont\{base\_font\_cs\}\{alt\_font\_cs\}\{(range)\}}

where \texttt{\{base\_font\_cs\}} and \texttt{\{alt\_font\_cs\}} are defined by \texttt{\jfont}. Its meaning is

If the current Japanese font is \texttt{\{base\_font\_cs\}}, characters which belong to \texttt{\{range\}} is typeset by another Japanese font \texttt{\{alt\_font\_cs\}}, instead of \texttt{\{base\_font\_cs\}}.
Here, \((range)\) 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 \(\text{base_font.cs}\). Note that characters which do not exist in \(\text{alt_font.cs}\) are ignored.

For example, if \(\text{\textbackslash hoge}\) uses \texttt{jfm-ujis.lua}, the standard JFM of Lua\TeX-ja, then
\[
\text{\textbackslash ltjdeclarealtfont\textbackslash hoge\textbackslash piyo\{"3000-"30FF, \{-1\}-\{-1\}}
\]
does

If the current Japanese font is \(\text{\textbackslash hoge}\), U+3000–U+30FF and characters in class 1 (ideographic opening brackets) are typeset by \(\text{\textbackslash piyo}\).

Note that specifying negative numbers needs specification like \(\{-1\}-\{-1\}\), because simple \("-1\) is treated as the range between 0 and 1.

\begin{enumerate}
\item \texttt{\textbackslash selectfont}\ loads (and chooses) only the Japanese font for the current direction, and does not load the Japanese font for other direction (Lua\TeX-ja only detects its size and JFM, to calculate the amount of shifting the baseline).
\item Direction changing commands (\texttt{\textbackslash yoko}, \texttt{\textbackslash tate}, \texttt{\textbackslash dtou}, \texttt{\textbackslash utod}) are patched to include the following process:
   \begin{itemize}
   \item If the Japanese font for new direction is not loaded, Lua\TeX-ja loads it automatically.
   \end{itemize}
Original commands are saved as \texttt{\textbackslash ltj@@orig\textbackslash yoko} etc.
\item Specifying Japanese font command which is defined by \texttt{\jfont}, \texttt{\tfont}, or \texttt{\DeclareFixedFont} directly actually loads (and selects) the Japanese font. For example, \texttt{\jchars} in \texttt{\box0} will be typeset in \texttt{\HOGF}, in the following code:
\begin{verbatim}
% in horizontal direction (\texttt{\textbackslash yoko})
\DeclareFixedFont{\HOGF}{JT3}{gt}{m}{n}{12} % JT3: for vertical direction
\HOGF
\setbox0=\hbox{\tate \textbackslash あいう}
\end{verbatim}
\end{enumerate}

\section{Commands for \LaTeX\ 2E}

\subsection{Loading Japanese fonts in \LaTeX\ 2E}

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

\begin{itemize}
\item \texttt{\selectfont} loads (and chooses) only the Japanese font for the current direction, and does not load the Japanese font for other direction (Lua\TeX-ja only detects its size and JFM, to calculate the amount of shifting the baseline).
\item Direction changing commands (\texttt{\yoko}, \texttt{\tate}, \texttt{\dtou}, \texttt{\utod}) are patched to include the following process:
   \begin{itemize}
   \item If the Japanese font for new direction is not loaded, Lua\TeX-ja loads it automatically.
   \end{itemize}
Original commands are saved as \texttt{\ltj@@orig@yoko} etc.
\item Specifying Japanese font command which is defined by \texttt{\jfont}, \texttt{\tfont}, or \texttt{\DeclareFixedFont} directly actually loads (and selects) the Japanese font. For example, \texttt{\jchars} in \texttt{\box0} will be typeset in \texttt{\HOGF}, in the following code:
\begin{verbatim}
% in horizontal direction (\texttt{\textbackslash yoko})
\DeclareFixedFont{\HOGF}{JT3}{gt}{m}{n}{12} % JT3: for vertical direction
\HOGF
\setbox0=\hbox{\tate \textbackslash あいう}
\end{verbatim}
\end{itemize}

\subsection{Patch for NFSS2}

Japanese patch for NFSS2 in Lua\TeX-ja is based on \texttt{plfonts.dtx} which plays the same role in \texttt{pl\LaTeX\ 2E}. We will describe commands which are not described in Subsection 3.1.

\textbf{additional dimensions}

Like \texttt{pl\LaTeX\ 2E}, Lua\TeX-ja defines the following dimensions for information of current Japanese font:
\begin{verbatim}
\cht (height), \cdp (depth), \cHT (sum of former two),
\cwd (width), \cvs (lineskip), \chs (equals to \cwd)
\end{verbatim}

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 \zw and \zh respectively. On the one hand the former dimensions are determined from a character whose character class is zero, but on the other hand \zw and \zh are specified by JFM.

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

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 commands define a new encoding scheme for Japanese font families.

\DeclareKanjiEncodingDefaults{⟨text-settings⟩}{⟨math-settings⟩}
\DeclareKanjiSubstitution{⟨encoding⟩}{⟨family⟩}{⟨series⟩}{⟨shape⟩}
\DeclareErrorKanjiFont{⟨encoding⟩}{⟨family⟩}{⟨series⟩}{⟨shape⟩}{⟨size⟩}

The above 3 commands are just the counterparts for \DeclareFontEncodingDefaults and others.

\reDeclareMathAlphabet{⟨unified-cmd⟩}{⟨al-cmd⟩}{⟨ja-cmd⟩}
\DeclareRelationFont{⟨ja-encoding⟩}{⟨ja-family⟩}{⟨ja-series⟩}{⟨ja-shape⟩}{⟨al-encoding⟩}{⟨al-family⟩}{⟨al-series⟩}{⟨al-shape⟩}

This command sets the “accompanied” alphabetic font (given by the latter 4 arguments) with respect to a Japanese font given by the former 4 arguments.

\SetRelationFont

This command is almost same as \DeclareRelationFont, except that this command does a local assignment, where \DeclareRelationFont does a global assignment.

\userelfont

(Only) at the next call of \selectfont, change current alphabetic font encoding/family/… to the ‘accompanied’ alphabetic font family with respect to current Japanese font family, which was set by \DeclareRelationFont or \SetRelationFont.

The following is an example of \SetRelationFont and \userelfont:

```latex
\makeatletter
\SetRelationFont{JY3}{\k@family}{m}{n}{TU}{lmss}{m}{n}
\% \k@family: current Japanese font family
\userelfont\selectfontあいう abc
```

\adjustbaseline

In p\LaTeX 2e, \adjustbaseline sets \baselineshift to match the vertical center of “M” and that of “漢” in vertical typesetting:

\[ \text{\baselineshift} = \frac{(h_M + d_M) - (h_{\text{漢}} + d_{\text{漢}})}{2} + d_{\text{漢}} - d_M, \]

where \( h_a \) and \( d_a \) denote the height of “a” and the depth, respectively. In Lua\LaTeX-ja, this \adjustbaseline does similar task, namely setting the \texttt{talbaselineshift} parameter (a Japanese character whose character class is zero is used, instead of “漢”).

\fontfamily{⟨family⟩}

As in \LaTeX 2e, this command changes current font family (alphabetic, Japanese, or both) to ⟨family⟩. See Subsection 11.3 for detail.

\fontshape{⟨shape⟩}, \fontshapeforce{⟨shape⟩}

As in \LaTeX 2e, this command changes current alphabetic font shape according to shape change rules. Traditionally, \fontshape changes also current Japanese font shape always. However, this leads a lot of \LaTeX font warning like
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 10. An example of \DeclareAlternateKanjiFont

Font shape `JY3/mc/m/it' undefined
using `JY3/mc/m/n' instead on ....

when \itsshape is called, because almost all Japanese fonts only have shape "n", and \itsshape calls \fontshape.

LuaTeX-ja 20200323.0 change the behavior. Namely, \fontshape{⟨shape⟩} and \fontshapeforce{⟨shape⟩} change current Japanese font shape, only if the required shape (according to shape changing rules) or ⟨shape⟩ is available in current Japanese font family/series. When this is not the case, an info such as

Kanji font shape JY3/mc/m/it' undefined
No change on ...

is issued instead of a warning.

\kanjishape{⟨shape⟩}, \kanjishapeforce{⟨shape⟩}
\kanjishape{⟨shape⟩} changes current Japanese font shape according to shape change rules, and
\kanjishapeforce{⟨shape⟩} changes current Japanese font shape to ⟨shape⟩, regardless of the rules. Hence \kanjishape{it} produces a warning

Font shape `JY3/mc/m/it' undefined
using `JY3/mc/m/n' instead on ....

which is not produced by \fontshape{it}.

\DeclareAlternateKanjiFont
{⟨base-encoding⟩}{⟨base-family⟩}{⟨base-series⟩}{⟨base-shape⟩}
{⟨alt-encoding⟩}{⟨alt-family⟩}{⟨alt-series⟩}{⟨alt-shape⟩}{⟨range⟩}
As \ltjdeclarealtfont (Subsection 10.5), characters in ⟨range⟩ of the Japanese font (we say the base font) which specified by first 4 arguments are typeset by the Japanese font which specified by fifth to eighth arguments (we say the alternate font). An example is shown in Figure 10.

• 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 NSSF2 commands, such as \DeclareSymbolFont and \SetSymbolFont, to specify Japanese fonts as math fonts.

Specifying disablesfam option in \usepackage prevents applying these patches. Hence one cannot write Japanese Characters in math mode directly if disablefam option is specified. The code below does not work either:
\DeclareSymbolFont{mincho}{JY3}{mc}{m}{n}
\DeclareSymbolFontAlphabet{\mathmc}{mincho}
11.3 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.

Remarks

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 \LaTeX 2e, but we re-implemented in Lua. We use an auxiliary list $N_J$.

1. If the family \fontfamily{family} has been defined already by \DeclareKanjiFamily, \fontfamily{family} is recognized as a Japanese font family. Note that \fontfamily{family} need not be defined under current Japanese font encoding.
2. If the family \fontfamily{family} has been listed in a list $N_J$, this means that \fontfamily{family} is not a Japanese font family.
3. If the luatexja-fontspec package is loaded, we stop here, and \fontfamily{family} is not recognized as a Japanese font family.
   
   If the luatexja-fontspec package is not loaded, now Lua\TeX-ja looks whether there exists a Japanese font encoding \en (such that a font definition named \en{\fontfamily{family}}.fd (the file name is all lowercase) exists. If so, \fontfamily{family} is recognized as a Japanese font family (the font definition file won’t be loaded here). If not, \fontfamily{family} is not a Japanese font family, and \fontfamily{family} is appended to the list $N_J$.

Recognition as alphabetic font family  Next, whether alphabetic font family will be changed is determined in following order. We use auxiliary lists $F_A$ and $N_A$.

1. If the family \fontfamily{family} has been listed in a list $F_A$, \fontfamily{family} is recognized as an alphabetic font family.
2. If the family \fontfamily{family} has been listed in a list $N_A$, this means that \fontfamily{family} is not an alphabetic font family.
3. If there exists an alphabetic font encoding such that the family \fontfamily{family} has been defined under it, \fontfamily{family} is recognized as an alphabetic font family, and to memorize this, \fontfamily{family} is appended to the list $F_A$.
4. Now Lua\TeX-ja looks whether there exists an alphabetic font encoding \en such that a font definition named \en{\fontfamily{family}}.fd (the file name is all lowercase) exists. If so, current alphabetic font family will be changed to \fontfamily{family} (the font definition file won’t be loaded here). If not, current alphabetic font family won’t be changed, and \fontfamily{family} is appended to the list $N_A$.

Also, each call of \DeclareFontFamily after loading of Lua\TeX-ja makes the second argument (family) is appended to the list $F_A$.

The above order is very similar to \fontfamily in \LaTeX 2e, but more complicated (clause 3.). This is because \LaTeX 2e is a format however Lua\TeX-ja is not, hence Lua\TeX-ja does not know calls of \DeclareFontFamily before itself is loaded.

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

11.4 Notes on \DeclareTextSymbol

From \LaTeX 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 produces 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 \textquotedblleft is chardef tokens. However, this no longer holds in Lua\TeX-ja; for example, the meaning of \textquotedblleft is \tjalchar8222\textquotedblleft.
Table 15. 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>dtou</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>

11.5 \texttt{\textbackslash strutbox}

As p\TeX\ (2017/04/08 or later), \texttt{\textbackslash strutbox} is a macro which is expanded to one of \texttt{\textbackslash ystrutbox}, \texttt{\textbackslash tstrutbox}, and \texttt{\textbackslash dstrutbox} (all of them are shown in Table 15), according to the current direction. Similarly, \texttt{\textbackslash strut} now uses one of these boxes.

12 expl3 interface

This section describes expl3 interfaces provided by Lua\TeX-ja. All of them belong to the \texttt{platex} module, since they are provided for compatibility with Japanese p\TeX. Note that commands which are marked with dagger (“†”) are additions by Lua\TeX-ja.

\texttt{\textbackslash platex\ direction}\ yoko: \texttt{\textbackslash platex\ direction\ tate:} \texttt{\textbackslash platex\ direction\ dtou:}

Synonyms for \texttt{\textbackslash yoko}, \texttt{\textbackslash tate} and \texttt{\textbackslash dtou}, respectively.

\texttt{\textbackslash platex\ if\ direction\ yoko\_p:}
\texttt{\textbackslash if\ direction\ yoko:TF} {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the current direction is \texttt{yoko} (horizontal writing).

\texttt{\textbackslash if\ direction\ tate\_nomath\_p:†}
\texttt{\textbackslash if\ direction\ tate\_nomath:TF}† {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the current direction is \texttt{tate} (vertical writing).

\texttt{\textbackslash if\ direction\ tate\_math\_p:†}
\texttt{\textbackslash if\ direction\ tate\_math:TF}† {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the current direction is \texttt{utod}.

\texttt{\textbackslash if\ direction\ tate\_p:}
\texttt{\textbackslash if\ direction\ tate:TF} {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the current direction is \texttt{tate} or \texttt{utod}.

\texttt{\textbackslash if\ direction\ dtou\_p:}
\texttt{\textbackslash if\ direction\ dtou:TF} {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the current direction is \texttt{dtou}.

\texttt{\textbackslash if\ box\_yoko\_p:N} \langle\texttt{box}\rangle
\texttt{\textbackslash if\ box\_yoko:N\textbackslash TF} \langle\texttt{box}\rangle {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the direction of \langle\texttt{box}\rangle is \texttt{yoko}.

\texttt{\textbackslash if\ box\_tate\_nomath\_p:N}† \langle\texttt{box}\rangle
\texttt{\textbackslash if\ box\_tate\_nomath:N\textbackslash TF}† \langle\texttt{box}\rangle {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the direction of \langle\texttt{box}\rangle is \texttt{tate}.

\texttt{\textbackslash if\ box\_tate\_math\_p:N}† \langle\texttt{box}\rangle
\texttt{\textbackslash if\ box\_tate\_math:N\textbackslash TF}† \langle\texttt{box}\rangle {\{\texttt{true code}\}}{\{\texttt{false code}\}}
Tests if the direction of \langle\texttt{box}\rangle is \texttt{utod}.

43
YokoFeatures={Color=FF1900}, TateFeatures={Color=003FFF},
TateFont=HaranoAjiGothic-Regular
{HaranoAjiMincho-Regular}
\hbox{\yoko 横組のテスト}\hbox{\tate 縦組のテスト}
\addjfontfeatures{Color=00AF00}
\hbox{\yoko 横組}\hbox{\tate 縦組}

Figure 11. An example of TateFeatures etc.

\platex_if_box_dtou_p:N ⟨box⟩
\platex_if_box_dtou:N{TF} ⟨box⟩ {⟨true code⟩}{⟨false code⟩}
Tests if the direction of ⟨box⟩ is dtou.

13 Addon packages

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

13.1 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:

CID=⟨name⟩, JFM=⟨name⟩, JFM-var=⟨name⟩
These 3 keys correspond to cid, jfm and jfmvar keys for \jfont and \tfont respectively. See Subsections 8.1 and 8.4 for details of cid, jfm and jfmvar 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 8.4.

Kanjiskip=⟨bool⟩

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 11.

YokoFeatures={⟨features⟩}
The YokoFeatures key specifies font features which are only turned on in horizontal writing. A demonstration is shown in Figure 11.

AltFont
As \ltjdeclarealtfont (Subsection 10.5) and \DeclareAlternateKanjiFont (Subsection 11.2), 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:
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 12. An example of AltFont

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 12.

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=FF1900} } }

does not work. Instead, one have to write

UprightFeatures = {
  AltFont = { { Font=HogeraMin-Light, Range="3000-"30FF, } },
},
BoldFeatures = {
  AltFont = { { Font=HogeraMin-Bold, Range="3000-"30FF, Color=FF1900 } },
}

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.

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 11 has
no effect, because a color specification is already done in YokoFeatures and TateFeatures keys.

13.2 luatexja-otf

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

\CID{⟨number⟩}

Typeset a glyph whose CID number is ⟨number⟩. If the Japanese font is neither Adobe-Japan1, Adobe-
GB1, Adobe-CNS1, Adobe-Korea1, nor Adobe-KR CID-keyed font, LuaTeX-ja treats that ⟨number⟩ is
a CID number of Adobe-Japan1 character collection, and tries to typeset a “most suitable glyph”.

Note that if the Japanese font is loaded using the HarfBuzz library, this \CID command does not work.
以上の原理は、「包除原理」とよく呼ばれるが

Figure 13. Line adjustment

\UTF{⟨hex_number⟩}

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

This package automatically loads lua\texttt{texja-ajmacros.sty}, which is slightly modified version of ajmacros.sty\textsuperscript{11}. Hence one can use macros which are defined in ajmacros.sty, such as \aj半角.

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

- Always treated as JA\texttt{Char}s.
- In vertical direction, vert/vrt2 feature are automatically applied to characters by \UTF{}, regardless these feature are not activated in current Japanese font.
- Processes for supporting other OpenType features (for example, glyph replacement and kerning) by the luaotfload package is not performed to these characters.

\textbf{Additional syntax of JFM} The package lua\texttt{texja-of} 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 jfm-ujis.lua to typeset halfwidth Hiragana glyphs (CID 516–598) in halfwidth.

13.3 \texttt{luatexja-adjust}

(see Japanese version of this manual)

13.4 \texttt{luatexja-ruby}

This addon package provides functionality of "ruby" (furigana) annotations using callbacks of Lua\TeX{}ja. There is no detailed manual of luatexja-ruby.sty in English. (Japanese manual is another PDF file, luatexja-ruby.pdf.)

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

\begin{quote}

東西線\ruby{}妙典}{みようでん}駅は……

東西線妙典駅は……

東西線の\ruby{}妙典}{みようでん}駅は……

東西線の妙典駅は……

東西線の\ruby{}妙典}{みようでん}という駅……

東西線の妙典という駅……

東西線\ruby{}葛西}{かさい}駅は……

東西線葛西駅は……

\end{quote}

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

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

\textsuperscript{11}Useful macros by iNOUE Koich\textsubscript{i}, for the japanese-otf package.
東西線の妙典でん駅は……

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.

13.5 lltjext.sty

pLaTeX supplies additional macros for vertical writing in the plex package. The lltjext package which we want to describe here is the LuaLaTeX-ja counterpart of the plex package.

\begin{tabular}<dir>[pos]{table spec} ... \end{tabular}
\begin{array}<dir>[pos]{table spec} ... \end{array}
\begin{minipage}<dir>[pos]{width} ... \end{minipage}
\parbox<dir>\[⟨pos⟩]{⟨width⟩}{⟨contents⟩}

This command typeset \langle contents⟩ in LR-mode, in \langle dir⟩ direction. If \langle width⟩ is positive, the width of the box becomes this \langle width⟩. In this case, \langle contents⟩ will be aligned to left (when \langle pos⟩ is 1), center (c), or right (r).
13.6 \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-pron]{luatexja-preset}
\%--------
\usepackage[no-math,match,kozuka-pron]{luatexja-preset}
\end{verbatim}

13.6.1 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 \texttt{fontspec} package is automatically loaded by this package.

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

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

\texttt{nfssonly}

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

When this option is specified, \texttt{fontspec} and \texttt{luatexja-fontspec} are \textit{not} loaded by default. Nevertheless, the \texttt{fontspec} can coexist with the option, as the following:

\begin{verbatim}
\usepackage[...]{fontspec}
\usepackage[hiragino-pron,nfssonly]{luatexja-preset}
\end{verbatim}

In this case, one can use \texttt{setmainfont} etc. to select \textit{alphabetic} fonts.

\texttt{match}

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

\texttt{nodeluxe} (enabled by default)

The negation of \texttt{delyxe} option. Use one-weighted \textit{mincho} and \textit{gothic} font families. This means that \texttt{\mcfamily\bfseries}, \texttt{\gtfamily\bfseries} and \texttt{\gtfamily\mdseries} use the same font.

\footnote{if \texttt{nfssonly} option is \textit{not} specified; in this case these options are simply ignored.}
**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 \gtfamily\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, \gtfamily\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 medium series of gothic and bold series of mincho. If node\texttt{lux}e 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.

**jf\texttt{m}_yoko=\texttt{jf\texttt{m}}**

Use the JFM \texttt{jf\texttt{m}-(jf\texttt{m})\texttt{.lua}} for horizontal direction, instead of \texttt{jf\texttt{m}-ujis\texttt{.lua}} (default JFM).

**jf\texttt{m}_tate=\texttt{jf\texttt{m}}**

Use the JFM \texttt{jf\texttt{m}-(jf\texttt{m})\texttt{.lua}} for vertical direction, instead of \texttt{jf\texttt{m}-ujisv\texttt{.lua}} (default JFM).

**jis**

Same as jf\texttt{m}_yoko=\texttt{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.

### 13.6.2 Presets which support multi weights

Besides bizud, haranoaji, morisawa-pro, and morisawa-pr6n presets, fonts are specified by font name, not by file name. In following tables, starred fonts (e.g. KozGo...-Regular) are used for medium series of gothic, if and only if \texttt{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></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>mincho</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>gothic</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-Heavy</td>
<td>KozGoProVI-Heavy</td>
<td>KozGoPr6N-Heavy</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>KozGoPro-Heavy</td>
<td>KozGoProVI-Heavy</td>
<td>KozGoPr6N-Heavy</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>KozGoPro-Heavy</td>
<td>KozGoProVI-Heavy</td>
<td>KozGoPr6N-Heavy</td>
</tr>
</tbody>
</table>

\texttt{Provided by \mgfamily and \textmg}, because "rounded gothic" is called maru gothic (丸ゴシック) in Japanese.
Hiragino fonts (except Hiragino Mincho W2) are bundled with Mac OS X 10.5 or later. 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>mincho</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>gothic</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>rounded gothic</td>
<td></td>
<td>Hiragino Maru Gothic Pro W4</td>
<td>Hiragino Maru Gothic ProN W4</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>medium</td>
<td>BIZ-UDMinchoM.ttc</td>
</tr>
<tr>
<td>gothic</td>
<td>bold</td>
<td>BIZ-UDGothicB.ttc</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>BIZ-UDGothicB.ttc</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>BIZ-UDGothicB.ttc</td>
</tr>
</tbody>
</table>
Morisawa Pro (Adobe-Japan1-4) fonts.
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-RyuminPro-Bold.otf</td>
<td>A-OTF-RyuminPr6N-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-GothicBBBPro-Bold.otf</td>
<td>A-OTF-GothicBBBPr6N-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></td>
<td>A-OTF-Jun101Pro-Light.otf</td>
<td>A-OTF-ShinMGoPr6N-Light.otf</td>
</tr>
</tbody>
</table>

Yu fonts

Yu fonts bundled with Windows 8.1.
Yu fonts bundled with Windows 10.
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></td>
<td>YuGothic-Bold</td>
<td>YuGothic-Bold</td>
<td>YuGothic Bold</td>
</tr>
</tbody>
</table>

Moga-Mobo
MogaMincho, MogaGothic, and MoboGothic.

MogaEx-Mobo
MogaExMincho, MogaExGothic, and MoboExGothic.

These fonts can be downloaded from http://yozvox.web.fc2.com/.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>default, jis2004 option</th>
<th>jis2004 option</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>medium</td>
<td>MogaMincho</td>
<td>MogaMincho Bold</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>MogaMincho Bold</td>
<td>MoboGothicBold</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>Moga90Gothic</td>
<td>Moga90GothicBold</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Moga90Gothic Bold</td>
<td>MoboGothicBold</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>Moga90Gothic Bold</td>
<td>MoboGothicBold</td>
</tr>
</tbody>
</table>

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

Ume
Ume Mincho and Ume Gothic.

These fonts can be downloaded from https://ja.osdn.net/projects/ume-font/wiki/FrontPage.
sourcehan  Source Han Serif and Source Han Sans fonts (Language-specific OTF or OTC)
sourcehan-jp  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 Sans Regular*</td>
<td>Source Sans JP Regular*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Source Sans Medium</td>
<td>Source Sans JP Medium</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>Source Sans Bold</td>
<td>Source Sans JP Bold</td>
</tr>
<tr>
<td></td>
<td>extra</td>
<td>Source Sans Heavy</td>
<td>Source Sans JP Heavy</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>Source Han Sans Medium</td>
<td>Source Han Sans JP Medium</td>
</tr>
</tbody>
</table>

noto-otc  Noto Serif CJK and Noto Sans CJK fonts (OTC)
noto-otf, noto  Noto Serif CJK and Noto Sans CJK fonts (Language-specific OTF)
noto-jp  Noto Serif CJK and Noto Sans CJK fonts (Region-specific subset OTF)

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>noto-otc</th>
<th>noto-otf, noto</th>
<th>noto-jp</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>light</td>
<td>Noto Serif CJK Light</td>
<td>Noto Serif CJK JP Light</td>
<td>Noto Serif JP Light</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>Noto Serif CJK Regular</td>
<td>Noto Serif CJK JP Regular</td>
<td>Noto Serif JP Regular</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>Noto Sans CJK Regular*</td>
<td>Noto Sans CJK JP Regular*</td>
<td>Noto Sans JP Regular*</td>
</tr>
</tbody>
</table>

haranoaji  Harano Aji Fonts.
These fonts can be downloaded from
https://github.com/trueroad/HaranoAjiFonts. There is not rounded gothic family in Harano Aji Fonts.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>haranoaji</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>light</td>
<td>HaranoAjiMincho-Light.otf</td>
</tr>
<tr>
<td></td>
<td>medium</td>
<td>HaranoAjiMincho-Regular.otf</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>HaranoAjiMincho-Bold.otf</td>
</tr>
<tr>
<td>gothic</td>
<td>medium</td>
<td>HaranoAjiGothic-Regular.otf*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>HaranoAjiGothic-Medium.otf</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>HaranoAjiGothic-Bold.otf</td>
</tr>
<tr>
<td></td>
<td>extra</td>
<td>HaranoAjiGothic-Heavy.otf</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>HaranoAjiGothic-Medium.otf</td>
</tr>
</tbody>
</table>

### 13.6.3 Presets which do not support multi weights

Next, we describe settings for using only single weight.

<table>
<thead>
<tr>
<th>family</th>
<th>series</th>
<th>embed</th>
<th>ipa</th>
<th>ipaex</th>
<th>ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>mincho</td>
<td>Ryumin-Light (non-embedded)</td>
<td>no</td>
<td>IPA Mincho</td>
<td>IPAex Mincho</td>
<td>MS Mincho</td>
</tr>
<tr>
<td>gothic</td>
<td>GothicBBB-Medium (non-embedded)</td>
<td>no</td>
<td>IPA Gothic</td>
<td>IPAex Gothic</td>
<td>MS Gothic</td>
</tr>
</tbody>
</table>
13.6.4 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>HG Mincho E</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></td>
<td>HG Gothic M</td>
<td>HG Gothic M</td>
<td>HG Gothic M</td>
</tr>
<tr>
<td></td>
<td>bold</td>
<td>HG Gothic E</td>
<td>HG Gothic E</td>
<td>HG Gothic E</td>
</tr>
<tr>
<td></td>
<td>extra bold</td>
<td>HG Soei Kaku Gothic UB</td>
<td>HG Soei Kaku Gothic UB</td>
<td>HG Soei Kaku Gothic UB</td>
</tr>
<tr>
<td>rounded gothic</td>
<td></td>
<td>HG Maru Gothic M PRO</td>
<td>HG Maru Gothic M PRO</td>
<td>HG Maru Gothic M PRO</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.).

jis98, 98jis by file name (hgrme.ttc, hgrge.ttc, hgrsgu.ttc, hgrsmp.ttf).


13.6.5 Define/Use Custom Presets

From version 20170904.0, one can define new presets using \ltjnewpreset, and use them by \ltjapplypreset. These two commands can only be used in the preamble.

\ltjnewpreset{(name)}{(specification)}

Define new preset (name). This <name> cannot be same as other presets, options described in Sub-subsection 13.6.1, nor following 13 strings:

mc mc-l mc-m mc-b mc-bx gt gt-u gt-d gt-m gt-b gt-bx gt-eb mg-m

(specification) is a comma-separated list which consists of other presets and/or the following keys:

mc-l=⟨font⟩ mincho light
mc-m=⟨font⟩ mincho medium
mc-b=⟨font⟩ mincho bold
mc-bx=⟨font⟩ synonym for mc-b=⟨font⟩

gt-u=⟨font⟩ gothic, when deluxe option is not specified.
gt-d=⟨font⟩ gothic medium, when deluxe option is specified.
gt-m=⟨font⟩ gothic medium. This key is equivalent to "gt-u=⟨font⟩, gt-d=⟨font⟩".
gt-b=⟨font⟩ gothic bold

Note that this key also specifies mincho bold if bold option is specified.
gt-bx=⟨font⟩ synonym for gt-b=⟨font⟩
gt-eb=⟨font⟩ gothic extra bold
mg-m=⟨font⟩ rounded gothic
mc=⟨font⟩ Equivalent to

mc-l=⟨font⟩, mc-m=⟨font⟩, mc-b=⟨font⟩

gt=⟨font⟩ Equivalent to

gt-u=⟨font⟩, gt-d=⟨font⟩, gt-b=⟨font⟩, gt-eb=⟨font⟩

\ltjnewpreset*{(name)}{(specification)}

Almost same as \ltjnewpreset. However, if <name> matches a preset which already defined, this command simply overwrite it.
\ltxjaplypreset{⟨name⟩}
Set Japanese font families using preset ⟨name⟩.

Note that \ltxjnewpreset does not "expand" the definition to define a preset. This means that one can write as the following:

\ltxjnewpreset{hoge}{piyo, ac-b=HiraMinProN-W6}
\ltxjnewpreset{piyo}{mg-m=HiraMaruProN-W4}
\ltxjaplypreset{hoge}

■ Restrictions Presets which are defined by \ltxjnewpreset have following restrictions:

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

14 Storing Parameters

14.1 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.

\ltj@dimen@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.

\ltj@dimen@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.

\ltj@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.

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

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

\ltj@yablshift (attribute) The amount of shifting the baseline of alphabetic fonts in scaled point (2−16 pt). “unset” means zero.

\ltj@ykb1shift (attribute) The amount of shifting the baseline of Japanese fonts in scaled point (2−16 pt).

\ltj@tablshift (attribute)

\ltj@tkb1shift (attribute)

\ltj@autospc (attribute) Whether the auto insertion of kanjiskip is allowed at the node. 0 means “not allowed”, and the other value (including “unset”) means “allowed”.

\ltj@autoxspc (attribute) Whether the auto insertion of xkanjiskip is allowed at the node. 0 means “not allowed”, and the other value (including “unset”) means “allowed”.

\ltj@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 JAglue, unlike explicit \kern.

packed (2)

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

from_jfm–(from_jfm + 63) (4–67) Glues/kerns from JFM.

kanji.skip (68), kanji.skip.jfm (69) Glues from kanjiskip.
**xkanji skip** (70), **xkanji skip_jfm** (71) Glues from `xkanji.skip`.

**processed** (73) Nodes which is already processed by ....

**ic_processed** (74) Glues from an italic correction, but already processed in the insertion process of JAgles.

**boxbdd** (75) Glues/kerns that inserted just the beginning or the ending of an hbox or a paragraph.

**special_jaglue** (76) Glues from `\insert[x]kanjiskip`.

`\ltj@keat 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) **dir_node_auto** (128)

**dir_node_manual** (256)

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

**inhibitglue** Nodes for indicating that `\inhibitglue` is specified. The value field of these nodes doesn’t matter.

**stack_marker** Nodes for LuaTeX-ja’s stack system (see the next subsection). The value field of these nodes is current group level.

**char_by_cid** Nodes for JAchar which processes by luatoftload won’t be applied, and the character code is stored in the value field. Each node of this type are converted to a `glyph_node` after processes by luatoftload. Nodes of this type is used by `luatexbase.newuserwhatsitid`.

**replace_vs** Similar to `char_by_cid` whatsis above. These nodes are for ALchar which the callback process of luatoftload won’t be applied.

**begin_par** Nodes for indicating beginning of a paragraph. A paragraph which is started by `\item` in list-like environments has a horizontal box for its label before the actual contents. So ...

**direction**

These whatsis will be removed during the process of inserting JAgles.

### 14.2 Stack system of LuaTeX-ja

**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 `kanjiskip` is stored by a skip, and consider the following source:

1. `\ltjsetparameter{kanjiskip=0pt}ふがふが。%`
2. `\setbox8=\hbox[\ltjsetparameter{kanjiskip=5pt}ほげほげ]ふがふが。ほげほげびょびよ`
3. `\box8。びょびよ\par`

As described in Subsection 9.1, the only effective value of `kanjiskip` in an hbox is the latest value, so the value of `kanjiskip` which applied in the entire hbox should be 5 pt. However, by the implementation method of LuaTeX, this "5 pt" cannot be known from any callbacks. In the `tex/packaging.w` which is a file in the source of LuaTeX, there are the following codes:

```c
void package(int c)
{
  scaled h; /* height of box */
  halfword p; /* first node in a box */
  scaled d; /* max depth */
```
int grp;
grp = cur_group;
d = box_max_depth;

unsave();
save_ptr -= 4;

if (cur_list.mode_field == -hmode) {
    cur_box = filtered_hpack(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_hpack(), 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{\textbackslash ltj\@stack} for the stack level, and \texttt{\textbackslash ltj\@group@level} for the \TeX{}'s group level when the last assignment was done. Parameters are stored in one big table named charprop_stack_table, where charprop_stack_table\[i\] stores data of stack level \(i\). If a new stack level is created by \texttt{\textbackslash ltj\@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 44 (user defined), stack marker and the current group level respectively is appended to the current list (we refer this node by stack_flag). This enables us to know whether assignment is done just inside a hbox. Suppose that the stack level is \(s\) and the \TeX{}’s group level is \(t\) just after the hbox group, then:

\begin{itemize}
\item If there is no 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 \(s\).
\item If there is a stack_flag node whose value is \(t + 1\), 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 \(s + 1\).
\item If there are stack_flag nodes but all of their values are more than \(t + 1\), 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 \(s\).
\end{itemize}

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

14.3 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:

\begin{verbatim}
luatexja.stack.set_stack_table(index, <any> data)
\end{verbatim}

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} == \texttt{’global’}).

Stored data can be obtained as the return value of

\begin{verbatim}
luatexja.stack.get_stack_table(index, <any> default, <number> level)
\end{verbatim}

where \texttt{level} is the stack level, which is usually the value of \texttt{\textbackslash ltj\@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}.

\[14\texttt{[Dev-luatex]} \texttt{tex.currentgrouplevel}, a \texttt{post at 2008/8/19 by Jonathan Sauer.}]
14.4 Extending Parameters

Keys for \ltjsetparameter and \ltjglobalsetparameter can be extended, as in luatexja-adjust.

Setting parameters  Figure 14 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\setkeys[ltj]{japaram}{...}{...}
\ltjsetparameter and \ltjglobalsetparameter automatically sets luatexja.isglobal. Its meaning is the following.

\begin{equation}
\text{luatexja.isglobal} = \begin{cases} 
'global' & \text{(global assignment)}, \\
'\phantom{\text{global}}' & \text{(local assignment)}. 
\end{cases}
\end{equation}

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.
\begin{verbatim}
function luatexja.unary_pars.hoge (t)
  return 42
end
\end{verbatim}

Here the argument of luatexja.unary_pars.hoge is the stack level of Lua\TeX-ja’s stack system (see Subsection 14.2).

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,
\begin{verbatim}
function luatexja.binary_pars.fuga (c, t)
  return tostring(c) .. ',', ' .. tostring(42)
end
\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
\begin{verbatim}
\ltj@@decl@array@param{fuga}
\end{verbatim}
to indicate that "the parameter fuga needs an additional argument".

Figure 14. Definition of parameter setting commands
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 15. State transitions of \pTeX{}'s input processor

15  Linebreak after a Japanese Character

15.1  Reference: behavior in \pTeX{}

In \pTeX{}, 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{} due to the specification of callbacks in \LuaTeX{}. To clarify the difference between \pTeX{} and \LuaTeX{}, We briefly describe the handling of a line break in \pTeX{}, in this subsection.

\pTeX{}'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 15. Note that \pTeX{} doesn’t leave state $K$ after “beginning/ending of a group” characters.

15.2  Behavior in \LuaTeX{}-ja

States in the input processor of \LuaTeX{} 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 \LuaTeX{}-ja are as follows:
Table 16. Normalization of Kana Character Sequences with Combining (Semi)-voiced Sound Mark

| う+゛ → ゔ | か+゛ → が | き+゛ → ぎ | く+゛ → ぐ | け+゛ → げ | こ+゛ → ご |
| さ+゛ → ざ | し+゛ → さ | す+゛ → す | せ+゛ → せ | そ+゛ → そ | た+゛ → た |
| ち+゛ → ぢ | つ+゛ → つ | て+゛ → て | と+゛ → と | ほ+゛ → ほ | へ+゛ → へ |
| は+゜ → ぱ | ひ+゜ → ぴ | ふ+゜ → ぷ | け+゜ → け | こ+゜ → こ | ね+゜ → ね |

A character whose character code is \texttt{\textbackslash ltjlineendcomment}\textsuperscript{15} is appended to an input line, before \TeX{} actually process it, if and only if the following three conditions are satisfied:

1. The category code of \texttt{\textbackslash endlinechar}\textsuperscript{16} is 5 (end-of-line).
2. The category code of \texttt{\textbackslash ltjlineendcomment} itself is 14 (comment).
3. The input line matches the following "regular expression":

\[(\text{any char})^* (\text{JAchar}) (\{\text{catcode = 1}\} \cup \{\text{catcode = 2}\})^*\]

\textbf{Remark} The following example shows the major difference from the behavior of p\TeX{}.

\begin{verbatim}
\fontspec[Ligatures=TeX]{Linux Libertine O}
\setparameter{autoxspacing=false}
\setparameter{jacharrange={-6}}
\hfillあ\hfill
\setparameter{jacharrange={+6}}
\hfillい\hfill
\setparameter{autoxspacing=false}
\setparameter{jacharrange={+6}}
\hfillう
\end{verbatim}

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 \texttt{JChar}.

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 \texttt{JChar}. Since line 3 ends with an \texttt{JChar}, the comment character (whose character code is \texttt{\textbackslash 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 \texttt{ALChar}. Since line 4 ends with an \texttt{ALChar}, the linebreak immediately after this line emits a space.

### 15.3 Composition of Kana from Combining Character Sequences

In (u)p\TeX{}, Hiragana and Katakana in the NFD form\textsuperscript{17} are normalized to precomposed characters before (u)p\TeX{} looks into the input line. Character sequences which are involved in this normalization are shown in Table 16.

The above normalization process is also performed in Lua\TeX{}-ja version 20220103.0 or later, because fonts might not have these transformation as an OpenType feature.

\textsuperscript{15}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.

\textsuperscript{16}Usually, it is \texttt{return} (whose character code is 13).

\textsuperscript{17}namely, character sequences which contains "COMBINING KATAKANA-HIRAGANA VOICED SOUND MARK" (U+3099) and "COMBINING KATAKANA-HIRAGANA SEMI-VOICED SOUND MARK" (U+309A).
16 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 \LaTeX{} 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 \texttt{\ltjlineendcomment}) before each letter above \texttt{U+0080}. This method can omit 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}}.

16.1 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.

\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.

\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 \texttt{vscmd} key in this document.

\begin{verbatim}
def\IVSA#1#2#3#4#5{%  
\hbox to1em{\hss\textcolor{blue}{\raisebox{3.5pt}{\normalfont\ttfamily\fboxsep=0.5pt\fbox{\hbox to0.75em{\hss\tiny \oalign{0#1#2\crcr#3#4#5\crcr}\hss}}}}\hss}
}\end{verbatim}

\lstset{vscmd=\IVSB}

The default output command of variation selectors is stored in \texttt{\ltjlistingsvsstdcmd}.

■ The doubleletterspace key  \texttt{lltjp-listing} adds the \texttt{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{lstlisting}[doubleletterspace]
H : H H H H :
\end{lstlisting}

The \texttt{doubleletterspace} key is to adjust the column format. Sometimes characters are not vertically aligned. The following example is typeset with \texttt{basewidth=2em}, and you'll see the leftmost "H" is not vertically aligned.

\begin{lstlisting}
H :
H H H H :
\end{lstlisting}

\texttt{lltjp-listing} adds the \texttt{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{lstlisting}[doubleletterspace]
H :
H H H H :
\end{lstlisting}
16.2 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 \ltjt 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:

<table>
<thead>
<tr>
<th></th>
<th>Letter</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lst@ifletter</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>\lst@ifkanji</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Kanji</th>
<th>Open</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lst@ifletter</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>\lst@ifkanji</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>char in an identifier</th>
<th>other alphabet</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lst@ifkanji</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>\lst@ifkanji</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

Meaning

<table>
<thead>
<tr>
<th></th>
<th>Kanji</th>
<th>Open</th>
<th>Close</th>
</tr>
</thead>
<tbody>
<tr>
<td>\lst@ifletter</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>\lst@ifkanji</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

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></th>
<th>Next</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Letter</td>
</tr>
<tr>
<td></td>
<td>collects</td>
</tr>
<tr>
<td>Prev</td>
<td>collect</td>
</tr>
<tr>
<td>Kanji</td>
<td>outputs</td>
</tr>
<tr>
<td>Open</td>
<td></td>
</tr>
<tr>
<td>Close</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 \U0080 except Variation Selectors are classified into above 5 classes by the following rules:

- **ALchars** above or equal to \U0080 are classified as Letter.
- **J Achars** 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.
Table 17. 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.gz,luc}</td>
<td>UniJIS2004-UTF32-{H,V} Adobe-Japan1-UCS2</td>
</tr>
<tr>
<td>Adobe-Korea1-*</td>
<td>ltj-cid-auto-adobe-korea1.{lua.gz,luc}</td>
<td>UniKS-UTF32-* Adobe-Korea1-UCS2</td>
</tr>
<tr>
<td>Adobe-GB1-*</td>
<td>ltj-cid-auto-adobe-gb1.{lua.gz,luc}</td>
<td>UniGB-UTF32-* Adobe-GB1-UCS2</td>
</tr>
<tr>
<td>Adobe-CNS1-*</td>
<td>ltj-cid-auto-adobe-cns1.{lua.gz,luc}</td>
<td>UniCNS-UTF32-* Adobe-CNS1-UCS2</td>
</tr>
</tbody>
</table>

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 \texttt{ALchar}; the width of the other \texttt{JAchar} is double the width of \texttt{ALchar}.

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

17 Cache Management of \texttt{Lua\TeX}-ja

\texttt{Lua\TeX}-ja creates some cache files to reduce the loading time. in a similar way to the \texttt{luaotfload} package:

- Cache files are usually stored in (and loaded from) $\texttt{TEXMFVAR/luatexja/}$.
- In addition to caches of the text form (the extension is ".lua.gz", because they are compressed by gzip), caches of the binary (bytecode) form are supported.
  - In loading a cache, the binary cache precedes the text form.
  - When \texttt{Lua\TeX}-ja updates a compressed text cache \texttt{hoge.lua.gz}, its binary version is also updated.

17.1 Use of cache

\texttt{Lua\TeX}-ja uses the following cache:

\texttt{ltj-cid-auto-adobe-japan1.{lua.gz,luc}}

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 \texttt{Lua\TeX}-ja.

Similar caches are created as Table 17, if you specified \texttt{cid} key in \texttt{\jfont} to use other CID-keyed non-embedded fonts for Chinese or Korean, as in Page 27.

\texttt{ltj-kinsoku.luc}

The bytecode cache which default \texttt{kinsoku} parameters are stored.

\texttt{ltj-jisx0208.lua}

The bytecode version of \texttt{ltj-jisx0208.luc}. This is the conversion table between JIS X 0208 and Unicode which is used in Kanji-code conversion commands for compatibility with \texttt{p\TeX}.

\texttt{ltj-ivd_aj1.luc}

The bytecode version of \texttt{ltj-ivd_aj1.lua}.

\texttt{extra_{***}.{lua.gz,luc}}

This file contains some information (especially for vertical typesetting) about the font \texttt{***}.
17.2 Internal

Cache management system of Lua\TeX-ja is stored in luatexja.base (ltj-base.lua). There are four public functions for cache management in luatexja.base, where \(\langle\text{filename}\rangle\) stands for the file name without suffix:

\begin{verbatim}
save_cache(\langle\text{filename}\rangle, \langle\text{data}\rangle)
    Save a non-nil table \(\langle\text{data}\rangle\) into a cache \(\langle\text{filename}\rangle\). Both the compressed text form \(\langle\text{filename}\rangle\).lua.gz and its binary version are created or updated.

save_cache_luc(\langle\text{filename}\rangle, \langle\text{data}\rangle[, \langle\text{serialized_data}\rangle])
    Same as save_cache, except that only the binary cache is updated. The third argument \(\langle\text{serialized_data}\rangle\) is not usually given. But if this is given, it is treated as a string representation of \(\langle\text{data}\rangle\).

load_cache(\langle\text{filename}\rangle, \langle\text{outdate}\rangle)
    Load the cache \(\langle\text{filename}\rangle\). \(\langle\text{outdate}\rangle\) 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 \(\langle\text{filename}\rangle\).luc. 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 compressed text form \(\langle\text{filename}\rangle\).lua.gz. Hence, the return value of load_cache is non-nil, if and only if the updated cache is found.

remove_cache(\langle\text{filename}\rangle)
    Remove the cache \(\langle\text{filename}\rangle\).
\end{verbatim}
References