

The Lua \TeX -ja package

The Lua \TeX -ja project team

October 25, 2015

Contents

I	User's manual	3
1	Introduction	3
1.1	Backgrounds	3
1.2	Major changes from p \TeX	3
1.3	Notations	4
1.4	About the project	5
2	Getting Started	6
2.1	Installation	6
2.2	Cautions	7
2.3	Using in plain \TeX	7
2.4	Using in \LaTeX	7
3	Changing Fonts	8
3.1	plain \TeX and $\LaTeX 2_{\epsilon}$	8
3.2	luatexja-fontspec package	9
3.3	Presets of Japanese fonts	10
3.4	\backslash CID, \backslash UTF, and macros in japanese-otf package	13
3.5	Changing default Japanese fonts	13
4	Changing Internal Parameters	14
4.1	Range of J Achars	14
4.2	<u>kanjiskip</u> and <u>xkanjiskip</u>	16
4.3	Insertion setting of <u>xkanjiskip</u>	16
4.4	Shifting the baseline	17
4.5	<i>kinsoku</i> parameters and OpenType features	18
II	Reference	19
5	\backslashcatcode in Lua\TeX-ja	19
5.1	Preliminaries: \backslash catcode in p \TeX and up \TeX	19
5.2	Case of Lua \TeX -ja	19
5.3	Non-kanji characters in a control word	20
6	Directions	20
6.1	Boxes in different direction	20
6.2	Getting current direction	22
6.3	Overridden box primitives	22
7	Font Metric and Japanese Font	23
7.1	\backslash jfont	23
7.2	\backslash tfont	25
7.3	Prefix psft	25
7.4	Structure of a JFM file	26
7.5	Math font family	29
7.6	Callbacks	29

8	Parameters	31
8.1	<code>\ltjsetparameter</code>	31
8.2	<code>\ltjgetparameter</code>	33
9	Other Commands for plain \TeX and $\LaTeX 2_{\epsilon}$	34
9.1	Commands for compatibility with $\text{p}\TeX$	34
9.2	<code>\inhibitglue</code>	35
9.3	<code>\ltjdeclarealtfont</code>	35
10	Commands for $\LaTeX 2_{\epsilon}$	35
10.1	Patch for NFSS2	35
10.2	Detail of <code>\fontfamily</code> command	37
11	Addon packages	38
11.1	<code>luatexja-fontspec</code>	38
11.2	<code>luatexja-otf</code>	39
11.3	<code>luatexja-adjust</code>	40
11.4	<code>luatexja-ruby</code>	41
11.5	<code>lltjext.sty</code>	41
III	Implementations	42
12	Storing Parameters	42
12.1	Used dimensions, attributes and whatsit nodes	42
12.2	Stack system of $\text{Lua}\TeX\text{-ja}$	44
12.3	Lua functions of the stack system	45
12.4	Extending Parameters	45
13	Linebreak after a Japanese Character	46
13.1	Reference: behavior in $\text{p}\TeX$	46
13.2	Behavior in $\text{Lua}\TeX\text{-ja}$	46
14	Patch for the listings Package	48
14.1	Notes and additional keys	48
14.2	Class of characters	49
15	Cache Management of $\text{Lua}\TeX\text{-ja}$	50
15.1	Use of cache	50
15.2	Internal	51
	References	52

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

1.1 Backgrounds

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

Recently extensions of p_T_EX, namely up_T_EX (Unicode-implementation of p_T_EX) and ϵ -p_T_EX (merging of p_T_EX and ϵ -_T_EX extension), have developed to fill those gaps to some extent, but gaps still exist.

However, the appearance of Lua_T_EX changed the whole situation. With using Lua “callbacks”, users can customize the internal processing of Lua_T_EX. 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 p_T_EX

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

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

■ **Command names** p_T_EX adds several primitives, such as `\kanjiskip`, `\prebreakpenalty`, and `\ifdir`. They can be used as follows:

```
\kanjiskip=10pt \dimen0=kanjiskip
\tbaselineshift=0.1zw
\dimen0=\tbaselineshift
\prebreakpenalty`あ=100
\ifdir ... \fi
```

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

```
\ltjsetparameter{kanjiskip=10pt} \dimen0=\ltjgetparameter{kanjiskip}
\ltjsetparameter{talbaselineshift=0.1\zw}
\dimen0=\ltjgetparameter{talbaselineshift}
\ltjsetparameter{prebreakpenalty={`あ,100}}
\ifnum\ltjgetparemeter{direction}=4 ... \fi
```

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

■ **Linebreak after a Japanese character** In p_T_EX, 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_T_EX-ja doesn't have this feature completely, because of a specification of Lua_T_EX. For the detail, see Section 13.

■ **Spaces related to Japanese characters** The insertion process of glues/kerns between two Japanese characters and between a Japanese character and other characters (we refer glues/kerns of both kinds as **JAg glue**) is rewritten from scratch.

- As LuaTeX’s internal ligature handling is *node-based* (e.g., of `{ }fice` doesn’t prevent ligatures), the insertion process of **JAg glue** 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 **JAg glue** in pTeX are not effective anymore.* In concrete terms, the following two methods are not effective anymore:

ちよ{}つと ちよ\つと

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

ちよ\hbox{}つと

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

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

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

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

From this version, characters which belongs both ISO 8859-1 and JIS X 0208, such as ¶ and §, are now typeset in alphabetic fonts. This means that without the `\fontspec` (and `luatexja-fontspec`) package, these characters are not typeset correctly.

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).
 - **JAg char**: standing for characters which is used in Japanese typesetting, such as Hiragana, Katakana, Kanji, and other Japanese punctuation marks.
 - **AL char**: standing for all other characters like latin alphabets.

We say *alphabetic fonts* for fonts used in **AL char**, and *Japanese fonts* for fonts used in **JAg char**.

- A word in a sans-serif font with underline (like [prebreakpenalty](#)) means an internal parameter for Japanese typesetting, and it is used as a key in `\ltjsetparameter` command.
- A word in a sans-serif font without underline (like `fontspec`) means a package or a class of L^AT_EX.
- In this document, natural numbers start from zero. ω denotes the set of all natural numbers.

1.4 About the project

■ **Project Wiki** Project Wiki is under construction.

- <https://osdn.jp/projects/luatex-ja/wiki/FrontPage%28en%29> (English)
- <https://osdn.jp/projects/luatex-ja/wiki/FrontPage> (Japanese)
- <https://osdn.jp/projects/luatex-ja/wiki/FrontPage%28zh%29> (Chinese)

This project is hosted by OSDN.

■ **Members**

- Hironori KITAGAWA
- Kazuki MAEDA
- Takayuki YATO
- Yusuke KUROKI
- Noriyuki ABE
- Munehiro YAMAMOTO
- Tomoaki HONDA
- Shuzaburo SAITO
- MA Qiyuan

2 Getting Started

2.1 Installation

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

- LuaTeX beta-0.80.0 (or later)
- luaotfload v2.5 (or later)
- adobemapping (Adobe cmap and pdfmapping files)
- everyisel (if you want to use LuaTeX-ja with L^AT_EX 2_ε)
- fontspec v2.4
- IPAex fonts (<http://ipafont.ipa.go.jp/>)

In summary, this version of LuaTeX-ja no longer supports T_EX Live 2014 (or older version).

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

- MiKTeX (in `luatexja.tar.lzma`); see the next subsection
- T_EX Live (in `texmf-dist/tex/luatex/luatexja`)
- W32T_EX (in `luatexja.tar.xz`)

IPAex fonts are also available in these distributions.

■ Manual installation

1. Download the source, by one of the following method. At the present, LuaTeX-ja has no *stable* release.

- Clone the Git repository:

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

- Download the tar.gz archive of HEAD in the master branch from

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

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

2. Extract the archive. You will see `src/` and several other sub-directories. But only the contents in `src/` are needed to work LuaTeX-ja.
3. If you downloaded this package from CTAN, you have to run following commands to generate classes and `ltj-kinsoku.lua` (the file which stores default “*kinsoku*” parameters):

```
$ cd src
$ lualatex ltjclasses.ins
$ lualatex ltjsclasses.ins
$ lualatex ltjltxdoc.ins
$ luatex ltj-kinsoku_make.tex
```

Do not forget The last line (processing `ltj-kinsoku_make.tex`). `.{dtx,ins}` and `ltj-kinsoku_make.tex` used here are not needed in regular use.*

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 `mktexlsr` is needed to update the file name database, make it so.

2.2 Cautions

For changes from pTeX, 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.
- LuaTeX-ja is very slower than pTeX. Generally speaking, LuaJITTeX processes LuaTeX-ja about 30% faster than LuaTeX, but not always.
- **(Outdated) note for MiKTeX users** LuaTeX-ja requires that several CMap files¹ must be found from LuaTeX. Strictly speaking, those CMaps are needed only in the first run of LuaTeX-ja after installing or updating. But it seems that MiKTeX does not satisfy this condition, so you will encounter an error like the following:

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

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

2.3 Using in plain TeX

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

```
\input luatexja.sty
```

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

- The following 12 Japanese fonts are preloaded:

direction	classification	font name	“10 pt”	“7 pt”	“5 pt”
<i>yoko</i> (horizontal)	<i>mincho</i>	IPAex Mincho	<code>\tenmin</code>	<code>\sevenmin</code>	<code>\fivemin</code>
	<i>gothic</i>	IPAex Gothic	<code>\tengt</code>	<code>\seventgt</code>	<code>\fivegt</code>
<i>tate</i> (vertical)	<i>mincho</i>	IPAex Mincho	<code>\tentmin</code>	<code>\seventmin</code>	<code>\fivetmin</code>
	<i>gothic</i>	IPAex Gothic	<code>\tentgt</code>	<code>\seventgt</code>	<code>\fivetgt</code>

- With `luatexja.cfg`, one can use other fonts as “default” Japanese fonts (Subsection 3.5).
 - 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 **J**Achar and an **A**Lchar (the parameter [xkanjiskip](#)) is set to

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

2.4 Using in L^AT_εX

Using in L^AT_εX 2_ε is basically same. To set up the minimal environment for Japanese, you only have to load `luatexja.sty`:

```
\usepackage{luatexja}
```

It also does minimal settings (counterparts in pL^AT_εX are `plfonts.dtx` and `pldefs.ltx`):

- Font encodings for Japanese fonts is JY3 (for horizontal direction) and JT3 (for vertical direction).
- Traditionally, Japanese documents use two typeface categories: *mincho* (明朝体) and *gothic* (ゴシック体). *mincho* is used in the main text, while *gothic* is used in the headings or for emphasis.

¹UniJIS2004-UTF32-{H,V} and Adobe-Japan1-UCS2.

classification		family name	
<i>mincho</i> (明朝体)	<code>\textmc{...}</code>	<code>{\mcfamily ...}</code>	<code>\mcdefault</code>
<i>gothic</i> (ゴシック体)	<code>\textgt{...}</code>	<code>{\gtfamily ...}</code>	<code>\gtdefault</code>

- By default, the following fonts are used for *mincho* and *gothic*:

classification	family	\mdseries	\bfseries	scale
<i>mincho</i> (明朝体)	mc	IPAex Mincho	IPAex Gothic	0.962216
<i>gothic</i> (ゴシック体)	gt	IPAex Gothic	IPAex Gothic	0.962216

Note that the bold series in both family are same as the medium series of *gothic* family. There is no italic nor slanted shape for these mc and gt.

- Japanese characters in math mode are typeset by the font family mc.
- If you use the beamer class with the default font theme (which uses sans serif fonts) and with Lua \TeX -ja, you might want to change default Japanese fonts to *gothic* family. The following line changes the default Japanese font family to *gothic*:

```
\renewcommand{\kanjifamilydefault}{\gtdefault}
```

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, we have the counterparts of jclasses (standard classes in p \LaTeX) and jsclasses (classes by Haruhiko Okumura), namely, `ltjclasses`² and `ltjsclasses`³.

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

```
! Incompatible direction list can't be unboxed.
\@begindvi ->\unvbox \@begindvibox
\global \let \@begindvi \@empty
```

Now, Lua \TeX -ja automatically applies the patch `lltjp-geometry` to the geometry package, when the direction of the document is *tate* (vertical writing). This patch `lltjp-geometry` also can be used in p \LaTeX ; for the detail, please refer [lltjp-geometry.pdf](#) (Japanese).

3 Changing Fonts

3.1 plain \TeX and $\LaTeX 2_{\epsilon}$

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

■ **$\LaTeX 2_{\epsilon}$ (NFSS2)** For $\LaTeX 2_{\epsilon}$, Lua \TeX -ja adopted most of the font selection system of p $\LaTeX 2_{\epsilon}$ (in `plfonts.dtx`).

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

	encoding	family	series	shape	selection
alphabetic fonts	<code>\romanencoding</code>	<code>\romanfamily</code>	<code>\romanseries</code>	<code>\romanshape</code>	<code>\useroman</code>
Japanese fonts	<code>\kanjiencoding</code>	<code>\kanjifamily</code>	<code>\kanjiserie</code>	<code>\kanjishape</code>	<code>\usekanji</code>
both	—	—	<code>\fontseries</code>	<code>\fontshape</code>	—
auto select	<code>\fontencoding</code>	<code>\fontfamily</code>	—	—	<code>\usefont</code>

²`ltjarticle.cls`, `ltjbook.cls`, `ltjreport.cls`, `ltjtarticle.cls`, `ltjtbook.cls`, `ltjtreport.cls`. The latter `ltjt*.cls` are for vertically written Japanese documents.

³`ltjsarticle.cls`, `ltjsbook.cls`, `ltjskiyou.cls`.

`\fontencoding{<encoding>}` changes the encoding of alphabetic fonts or Japanese fonts depending on the argument. For example, `\fontencoding{JY3}` changes the encoding of Japanese fonts to JY3, and `\fontencoding{T1}` changes the encoding of alphabetic fonts to T1. `\fontfamily` also changes the current Japanese font family, the current alphabetic font family, or both. For the detail, see Subsection 10.1.

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

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

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

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

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 “高温” in above is used as a textual label, and “よって” is used as a conjunction.
- However, the line 3 is correct, since “素” is used as an identifier.

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

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

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 7.5.

3.2 luatexja-fontspec package

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

```
\usepackage[<options>]{luatexja-fontspec}
```

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

In the luatexja-fontspec package, the following seven commands are defined as counterparts of original commands in the fontspec package:

Japanese fonts	<code>\jfontspec</code>	<code>\setmainjfont</code>	<code>\setsansjfont</code>	<code>\setmonojfont</code>
alphabetic fonts	<code>\fontspec</code>	<code>\setmainfont</code>	<code>\setsansfont</code>	<code>\setmonofont</code>
Japanese fonts	<code>\newjfontfamily</code>	<code>\newjfontface</code>	<code>\defaultjfontfeatures</code>	<code>\addjfontfeatures</code>
alphabetic fonts	<code>\newfontfamily</code>	<code>\newfontface</code>	<code>\defaultfontfeatures</code>	<code>\addfontfeatures</code>

The package option of `luatexja-fontspec` are the followings:

`match`

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

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

`pass=<opts>`

(*Obsoleted*) Specify options `<opts>` which will be passed to the `fontspec` package.

`scale=<float>`

Override the ratio of the font size of Japanese fonts to that of alphabetic fonts. The default value is calculated automatically (for example, about 0.924865 when the `ltsarticle` class is used).

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

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

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

Below is an example of `\jfontspec`.

```
1 \jfontspec[CJKShape=NLC]{KozMinPr6N-Regular}
2 JIS-X-0213:2004→辻\par JIS X 0213:2004 →辻
3 \jfontspec[CJKShape=JIS1990]{KozMinPr6N-Regular} JIS X 0208:1990 →辻
4 JIS-X-0208:1990→辻
```

3.3 Presets of Japanese fonts

One can load the `luatexja-preset` package to use several “presets” of Japanese fonts. This package provides functions in a part of `japanese-otf` package and a part of `PXchfon` package by Takayuki Yato.

One can specified other options other than listed in this subsection. These are simply passed to the `luatexja-fontspec`⁴. For example, the line 5 in below example is equivalent to lines 1–3.

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

■ General options

`fontspec`

With this option, Japanese fonts are selected using functionality of the `luatexja-fontspec` package. This means that the `fontspec` package is automatically loaded by this package. *This option is enabled by default.*

If you need to pass some options to `fontspec`, you can load `fontspec` manually before `luatexja-preset`:

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

`nfssonly`

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

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

⁴if `nfssonly` option is *not* specified; in this case these options are simply ignored.

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

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

nodeluxe

Use one-weighted *mincho* and *gothic* font families. This means that `\mcfamily\bfseries`, `\gtfamily\bfseries` and `\gtfamily\mdseries` use the same font. *This option is enabled by default.*

deluxe

Use *mincho* with two weights (medium and bold), *gothic* with three weights (medium, bold and heavy), and *rounded gothic*⁵. The heavy weight of *gothic* can be used by “changing the family” `\gtebfamily`, or `\textgteb{...}`. This is because the `fontspec` package can handle only medium (`\mdseries`) and bold (`\bfseries`).

expert

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

bold

Substitute bold series of *gothic* for bold series of *mincho*.

90jis

Use 90JIS glyph variants if possible.

jis2004

Use JIS2004 glyph variants if possible.

jis

Use the JFM `jfm-jis.lua`, instead of `jfm-ujis.lua`, which is the default JFM of Lua_{TeX}-ja.

Note that `90jis` and `jis2004` only affect with *mincho*, *gothic* (and possibly *rounded gothic*) defined by this package. We didn’t taken account of when both `90jis` and `jis2004` are specified.

■ **Presets for multi weight** Besides `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 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.

family	series	kozuka-pro	kozuka-pr6	kozuka-pr6n
<i>mincho</i>	medium	KozMinPro-Regular	KozMinProVI-Regular	KozMinPr6N-Regular
	bold	KozMinPro-Bold	KozMinProVI-Bold	KozMinPr6N-Bold
<i>gothic</i>	medium	KozGoPro-Regular*	KozGoProVI-Regular*	KozGoPr6N-Regular*
		KozGoPro-Medium	KozGoProVI-Medium	KozGoPr6N-Medium
	bold	KozGoPro-Bold	KozGoProVI-Bold	KozGoPr6N-Bold
	heavy	KozGoPro-Heavy	KozGoProVI-Heavy	KozGoPr6N-Heavy
<i>rounded gothic</i>		KozGoPro-Heavy	KozGoProVI-Heavy	KozGoPr6N-Heavy

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

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

Hiragino fonts are bundled with Mac OS X 10.5 or later. Some editions of a Japanese word-processor “一太郎 2012” includes Hiragino ProN fonts. Note that the heavy weight of *gothic* family only supports Adobe-Japan1-3 character collection (Std/StdN).

⁵Provided by `\mgfamily` and `\textmg`, because *rounded gothic* is called *maru gothic* (丸ゴシック) in Japanese.

family	series	hiragino-pro	hiragino-pron
<i>mincho</i>	medium	Hiragino Mincho Pro W3	Hiragino Mincho ProN W3
	bold	Hiragino Mincho Pro W6	Hiragino Mincho ProN W6
<i>gothic</i>	medium	Hiragino Kaku Gothic Pro W3*	Hiragino Kaku Gothic ProN W3*
		Hiragino Kaku Gothic Pro W6	Hiragino Kaku Gothic ProN W6
	bold heavy	Hiragino Kaku Gothic Pro W6 Hiragino Kaku Gothic Std W8	Hiragino Kaku Gothic ProN W6 Hiragino Kaku Gothic StdN W8
<i>rounded gothic</i>		Hiragino Maru Gothic Pro W4	Hiragino Maru Gothic ProN W4

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

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

family	series	morisawa-pro	morisawa-pr6n
<i>mincho</i>	medium	A-OTF-RyuminPro-Light.otf	A-OTF-RyuminPr6N-Light.otf
	bold	A-OTF-FutoMinA101Pro-Bold.otf	A-OTF-FutoMinA101Pr6N-Bold.otf
<i>gothic</i>	medium	A-OTF-GothicBBBPro-Medium.otf	A-OTF-GothicBBBPr6N-Medium.otf
	bold	A-OTF-FutoGoB101Pro-Bold.otf	A-OTF-FutoGoB101Pr6N-Bold.otf
	heavy	A-OTF-MidashiGoPro-MB31.otf	A-OTF-MidashiGoPr6N-MB31.otf
<i>rounded gothic</i>		A-OTF-Jun101Pro-Light.otf	A-OTF-ShinMGoPr6N-Light.otf

yu-win Yu fonts bundled with Windows 8.1.

yu-osx Yu fonts bundled with OSX Mavericks.

family	series	yu-win	yu-osx
<i>mincho</i>	medium	YuMincho-Regular	YuMincho Medium
	bold	YuMincho-Demibold	YuMincho Demibold
<i>gothic</i>	medium	YuGothic-Regular*	YuGothic Medium*
		YuGothic-Bold	YuGothic Bold
	bold heavy	YuGothic-Bold YuGothic-Bold	YuGothic Bold YuGothic Bold
<i>rounded gothic</i>		YuGothic-Bold	YuGothic Bold

moga-mobo MogaMincho, MogaGothic, and MobaGothic. These fonts can be downloaded from <http://yozvox.web.fc2.com/>.

family	series	default, 90jis option	jis2004 option
<i>mincho</i>	medium	Moga90Mincho	MogaMincho
	bold	Moga90Mincho Bold	MogaMincho Bold
<i>gothic</i>	medium	Moga90Gothic	MogaGothic
		Moga90Gothic	MogaGothic
	bold heavy	Moga90Gothic Bold Moga90Gothic Bold	MogaGothic Bold MogaGothic Bold
<i>rounded gothic</i>		Moba90Gothic	MobaGothic

■ **Presets for single weight** Next, we describe settings for using only single weight.

	noembed	ipa	ipaex	ms
<i>mincho</i>	Ryumin-Light (non-embedded)	IPA Mincho	IPAex Mincho	MS Mincho
<i>gothic</i>	GothicBBB-Medium (non-embedded)	IPA Gothic	IPAex Gothic	MS Gothic

■ **Using HG fonts** We can use HG fonts bundled with Microsoft Office for realizing multiple weights.

	ipa-hg	ipaex-hg	ms-hg
mincho medium	IPA Mincho	IPAex Mincho	MS Mincho
mincho bold	HG Mincho E		
Gothic medium			
without deluxe	IPA Gothic	IPAex Gothic	MS Gothic
with jis2004	IPA Gothic	IPAex Gothic	MS Gothic
otherwise	HG Gothic M		
gothic bold	HG Gothic E		
gothic heavy	HG Soei Kaku Gothic UB		
rounded gothic	HG Maru Gothic PRO		

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.).

90jis by file name (hgrme.ttc, hgrge.ttc, hgrsgu.ttc, hgrsmp.ttf).

jis2004 by file name (hgrme04.ttc, hgrge04.ttc, hgrsgu04.ttc, hgrsmp04.ttf).

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

Under p_lA_TE_X, 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_TE_X-ja supports some of functions in the japanese-otf package, as an external package luatexja-otf.

```

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

```

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

3.5 Changing default Japanese fonts

If luatexja.cfg can be seen from Lua_TE_X, Lua_TE_X-ja automatically reads it. The main use of luatexja.cfg is for changing default Japanese fonts, when IPAex fonts cannot be installed in _TE_X system. One should not overuse this luatexja.cfg; fonts which will be used in a document should be specified in its source.

For example,

```

\def\ltj@stdmcfnt{IPAMincho}
\def\ltj@stdgtfont{IPAGothic}

```

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

For another example, the following two lines makes that non-embedded fonts Ryumin-Light and GothicBBB-Medium as default Japanese fonts (as the earlier version of Lua_TE_X-ja):

```

\def\ltj@stdmcfnt{psft:Ryumin-Light}
\def\ltj@stdgtfont{psft:GothicBBB-Medium}

```

4 Changing Internal Parameters

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

4.1 Range of JAchars

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

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

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

The distinction between **ALchar** and **JAchar** is performed by character ranges. This can be edited by setting the `jacharrange` parameter. For example, the code below is just the default setting of Lua \TeX -ja, and it sets

- a character which belongs character ranges 1, 4, 5, and 8 is **ALchar**,
- a character which belongs character ranges 2, 3, 6, and 7 is **JAchar**.

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

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

Note that characters U+0000–U+007F are always treated as an **ALchar** (this cannot be customized).

■ **Default character ranges** Lua \TeX -ja predefines eight character ranges for convenience. They are determined from the following data:

- Blocks in Unicode 6.0.
- The Adobe-Japan1-UCS2 mapping between a CID Adobe-Japan1-6 and Unicode.
- The PXbase bundle for up \TeX by Takayuki Yato.

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

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

- | | |
|-------------------------------|-----------------------------------|
| • § (U+00A7, Section Sign) | • ´ (U+00B4, Spacing acute) |
| • ¨ (U+00A8, Diaeresis) | • ¶ (U+00B6, Paragraph sign) |
| • ° (U+00B0, Degree sign) | • × (U+00D7, Multiplication sign) |
| • ± (U+00B1, Plus-minus sign) | • ÷ (U+00F7, Division Sign) |

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

Table 1. Unicode blocks in predefined character range 3.

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

Table 2. Unicode blocks in predefined character range 6.

U+2460–U+24FF	Enclosed Alphanumerics	U+2E80–U+2EFF	CJK Radicals Supplement
U+3000–U+303F	CJK Symbols and Punctuation	U+3040–U+309F	Hiragana
U+30A0–U+30FF	Katakana	U+3190–U+319F	Kanbun
U+31F0–U+31FF	Katakana Phonetic Extensions	U+3200–U+32FF	Enclosed CJK Letters and Months
U+3300–U+33FF	CJK Compatibility	U+3400–U+4DBF	CJK Unified Ideographs Extension A
U+4E00–U+9FFF	CJK Unified Ideographs	U+FA90–U+FAFF	CJK Compatibility Ideographs
U+FE10–U+FE1F	Vertical Forms	U+FE30–U+FE4F	CJK Compatibility Forms
U+FE50–U+FE6F	Small Form Variants	U+20000–U+2FFFF	(Supplementary Ideographic Plane)
U+E0100–U+E01EF	Variation Selectors Supplement		

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

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

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

Range 3^J Punctuations and Miscellaneous symbols. The block list is indicated in Table 1.

Range 4^A 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:

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

Range 5^A Surrogates and Supplementary Private Use Areas.

Range 6^J Characters used in Japanese. The block list is indicated in Table 2.

Range 7^J Characters used in CJK languages, but not included in Adobe-Japan1-6. The block list is indicated in Table 3.

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

For example, the codepoint `\textparagraph` which is provided by the `textcomp` package is 182. This codepoint corresponds ¶ (U+00B6) in Unicode. Similarly, `\Frown` which is provided by the `marvosym`

Table 3. Unicode blocks in predefined character range 7.

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

package has the same codepoint as § (U+00A7). Hence, as previous versions of LuaTeX-ja, if these characters are treated as JAchars, then `\textparagraph` produces “`\ljtjchar`¶`” (in a Japanese font), and `\Frownj` produces “`§`” (in a Japanese font).

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

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

```

1 \gtfamily\large % default, ALchar, JAchar
2 ¶, \ltjalchar`¶, \ltjjachar`¶\ % default: ALchar
3 α, \ltjalchar`α, \ltjjachar`α % default: JAchar

```

¶, ¶, ¶
α, α, α

4.2 [kanjiskip](#) and [xkanjiskip](#)

JAgglue is divided into the following three categories:

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

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

```

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

```

Here `\zw` is a internal dimension which stores fullwidth of the current Japanese font. This `\zw` can be used as the unit `zw` in pTeX.

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).

```

1 kanjiskip: \ltjgetparameter{kanjiskip},\ \ kanjiskip: 0.0pt plus 0.4pt minus 0.4pt,
2 xkanjiskip: \ltjgetparameter{xkanjiskip} \ xkanjiskip: 2.40555pt plus 1.0pt minus 1.0pt

```

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 **JAchars** and **ALchars**. 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 [jaxspmode](#) for **JAchars** and [alxspmode](#) parameters **ALchars** respectively.

```

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

```

p あq い! う

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

`jaxspmode` and `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:

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

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

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

4.4 Shifting the baseline

To make a match between a Japanese font and an alphabetic font, sometimes shifting of the baseline of one of the pair is needed. In pTeX, this is achieved by setting `\ybaselineshift` (or `\tbaselineshift`) to a non-zero length (the baseline of **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.

	Horizontal writing (<i>yoko</i> direction) etc.	Vertical writing (<i>tate</i> direction)
Alphabetic fonts	<code>\ybaselineshift</code> parameter	<code>\tbaselineshift</code> parameter
Japanese fonts	<code>\jabaselineshift</code> parameter	<code>\tjabaselineshift</code> parameter

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

```
1 \vrule width 150pt height 0.2pt depth 0.2pt \
   hskip-120pt
2 \ltjsetparameter{yjabaselineshift=0pt,
   yalbaselineshift=0pt}abcあいう
3 \ltjsetparameter{yjabaselineshift=5pt,
   yalbaselineshift=2pt}abcあいう
```

_____ abc あいう abc あいう _____

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):

```
1 \vrule width 150pt height 4.417pt depth -4.217pt%
2 \kern-150pt
3 \large xyz漢字
4 {\scriptsize
5 \ltjsetparameter{yjabaselineshift=-1.757pt,
6 yalbaselineshift=-1.757pt}
7 漢字xyzあいう
8 }あいうabc
```

xyz 漢字 漢字 xyz あいう あいう abc

Note that setting positive `\ybaselineshift` or `\tbaselineshift` parameters does not increase the depth of one-letter *syllable* `p` of **ALchar**, if its left-protrusion (`\lpcode`) and right-protrusion (`\rpcode`) are both non-zero. This is because

- These two parameters are implemented by setting `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 `\jabaselineshift` nor `\tjabaselineshift`, because a *JChar* 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.

```
1 \ltjsetParameter{postbreakpenalty}={`ア, 10}}
2 \ltjsetParameter{postbreakpenalty}={`ア, 20}}
3
4 \newcommand\showpostpena[1]{%
5   \leavevmode\setbox0=\hbox{#1\hbox{}}}%
6   \unhbox0\setbox0=\lastbox\the\lastpenalty}
7
8 \showpostpena{ア},
9 \showpostpena{ア},
10 {\addfontfeatures{CharacterWidth=Half}\showpostpena{ア}}
```

ア10, ア20, ア10

Part II

Reference

5 `\catcode` in Lua \TeX -ja

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

In p \TeX and up \TeX , the value of `\kcatcode` determines whether a Japanese character can be used in a control word. For the detail, see Table 4.

`\kcatcode` can be set by a row of JIS X 0208 in p \TeX , and generally by a Unicode block⁶ 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 `\kcatcode` 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 **J**Achar or **AL**char* is controlled by the character range, see Subsection 4.1.
- *Whether the character can be used in a control word* is controlled by setting `\catcode` to 11 (enabled) or 12 (disabled), as usual.
- *Whether `jcharwidowpenalty` can be inserted before the character* is controlled by the lowermost bit of the `\kcatcode` parameter.
- *Linebreak after a **J**Achar* does not produce a space.

Default setting of `\catcode` of Unicode characters are located in

plain Lua \TeX `luatex-unicode-letters.tex`, which is based on `unicode-letters.tex` (for X \TeX).

Lua \LaTeX now included in \LaTeX kernel as `unicode-letters.def`.

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

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

In other words,

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

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





























This would be inconvenient for p \TeX users to shifting to Lua \TeX -ja, since several control words containing Kanji or other fullwidth characters, such as `\西曆` or `\1年目西曆` are used in p \TeX . Hence, Lua \TeX -ja have a counterpart of `unicode-letters.tex` for Lua \TeX , *to match the `\catcode` setting with that of X \TeX* .

⁶up \TeX divides U+FF00–U+FFEF (Halfwidth and Fullwidth Forms) into three subblocks, and `\kcatcode` can be set by a subblock.

Table 4. `\kcatcode` in `upTeX`

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

Table 5. Difference of the set of non-kanji JIS X 0208 characters which can be used in a control word

	row	col.	<code>pTeX</code>	<code>upTeX</code>	<code>LuaTeX-ja</code>		row	col.	<code>pTeX</code>	<code>upTeX</code>	<code>LuaTeX-ja</code>
 (U+30FB)	1	6	N	Y	N	 (U+FF5C)	1	35	N	N	Y
 (U+309B)	1	11	N	Y	N	 (U+FF0B)	1	60	N	N	Y
 (U+309C)	1	12	N	Y	N	 (U+FF1D)	1	65	N	N	Y
 (U+FF40)	1	14	N	N	Y	 (U+FF1C)	1	67	N	N	Y
 (U+FF3E)	1	16	N	N	Y	 (U+FF1E)	1	68	N	N	Y
 (U+FFE3)	1	17	N	N	Y	 (U+FF03)	1	84	N	N	Y
 (U+FF3F)	1	18	N	N	Y	 (U+FF06)	1	85	N	N	Y
 (U+3003)	1	23	N	N	Y	 (U+FF0A)	1	86	N	N	Y
 (U+4EDD)	1	24	N	Y	Y	 (U+FF20)	1	87	N	N	Y
 (U+3005)	1	25	N	N	Y	 (U+3012)	2	9	N	N	Y
 (U+3006)	1	26	N	N	Y	 (U+3013)	2	14	N	N	Y
 (U+3007)	1	27	N	N	Y	 (U+FFE2)	2	44	N	N	Y
 (U+30FC)	1	28	N	Y	Y	 (U+212B)	2	82	N	N	Y
 (U+FF0F)	1	31	N	N	Y	Greek letters (row 6)			Y	N	Y
 (U+FF3C)	1	32	N	N	Y	Cyrillic letters (row 7)			N	N	Y

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 `pTeX`, in `upTeX`, and in `LuaTeX-ja`. Table 5 shows the difference. Except for four characters “•”, “°”, “””, “=”, `LuaTeX-ja` admits more characters in a control word than `upTeX`.

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

6 Directions

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

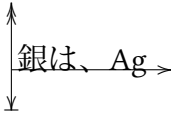



`LuaTeX-ja` supports four directions, as shown in Table 6. The second column (*yoko* direction) is just horizontal writing, and the third column (*tate* direction) is vertical writing. The fourth column (*dtou* direction) is actually a hidden feature of `pTeX`. We implemented this for debugging purpose. The fifth column (*utod* direction) corresponds the “*tate* (math) direction” of `pTeX`.

Directions can be changed by `\yoko`, `\tate`, `\dtou`, `\utod`, only when the current list is null. Also, the direction of a math formula is changed to *utod*, when the direction outside the math formula is *tate* (vertical writing).

6.1 Boxes in different direction

As in `pTeX`, one can use boxes of different direction in one document. The below is an example.

Table 6. Directions supported by LuaTeX-ja

	horizontal (<i>yoko</i> direction)	vertical (<i>tate</i> direction)	<i>dtou</i> direction	<i>utod</i> direction
Commands	<code>\yoko</code>	<code>\tate</code>	<code>\dtou</code>	<code>\utod</code>
Beginning of the page	Top	Right	Left	Right
Beginning of the line	Left	Top	Bottom	Top
Used Japanese font	horizontal (<code>\jfont</code>)	vertical (<code>\tfont</code>)	horizontal (90°)	rotated
Example				
(Notation used in Ω)	TLT	RTR, RTT	LBL	RTR

```

1 ここは横組% yoko
2 \hbox{\tate % tate
3 \hbox{縦組}% tate
4 の中に
5 \hbox{\yoko 横組の内容}% yoko
6 を挿入する
7 }
8 また横組に戻る% yoko

```

ここは横組 縦組の中に横組の内容を挿入する また横組に戻る

Table 7 shows how a box is arranged when the direction inside the box and that outside the box differ.

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

```

1 \setbox0=\hbox to 20pt{foo}
2 \the\wd0,~\hbox{\tate\vrule\the\wd0}
3 \wd0=100pt
4 \the\wd0,~\hbox{\tate \the\wd0}

```

20.0pt, 100.0pt

To access box dimensions *with respect to current direction*, one have 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.

```

1 \parindent0pt
2 \setbox32767=\hbox{\yoko よこぐみ}
3 \fboxsep=0mm\fbox{\copy32767}
4 \vbox{\hsize=20mm
5 \yoko YOKO \the\ltjgetwd32767, \
6 \the\ltjgetht32767, \ \the\ltjgetdp32767.}
7 \vbox{\hsize=20mm\raggedleft
8 \tate TATE \the\ltjgetwd32767, \
9 \the\ltjgetht32767, \ \the\ltjgetdp32767.}
10 \vbox{\hsize=20mm\raggedleft
11 \dtou DTOU \the\ltjgetwd32767, \
12 \the\ltjgetht32767, \ \the\ltjgetdp32767.}

```

YOKO
 38.48877pt,
 8.46753pt,
 1.15466pt.
 TATE
 9.6222pt,
 19.24438pt,
 19.24438pt.
 DTOU
 9.6222pt,
 19.24438pt,
 19.24438pt.

Table 7. Boxes in different direction

typeset in <i>yoko</i> direction	typeset in <i>tate</i> or <i>utod</i> direction	typeset in <i>dtou</i> direction
$W_Y = w_T + d_T,$ $H_Y = w_T,$ $D_Y = 0\text{pt}$	$W_T = w_Y + d_Y,$ $H_T = w_Y/2,$ $D_T = w_Y/2$	$W_D = w_Y + d_Y,$ $H_D = w_Y,$ $D_D = 0\text{pt}$
$W_Y = w_D + d_D,$ $H_Y = w_D,$ $D_Y = 0\text{pt}$	$W_T = w_D + d_D,$ $H_T = d_D,$ $D_T = D$	$W_D = w_T,$ $H_D = d_T,$ $D_D = w_T$

`\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*:

Direction	<i>yoko</i>	<i>tate</i>	<i>dtou</i>	<i>utod</i>	(empty)
Returned value	4	3	1	11	0

```

1 \leavevmode\def\DIR{\ltjgetparameter{direction}}
2 \hbox{\yoko \DIR}, \hbox{\tate\DIR},
3 \hbox{\dtou\DIR}, \hbox{\utod\DIR},
4 \hbox{\tate$\hbox{\tate math: \DIR}$}
5
6 \setbox2=\hbox{\tate}\ltjgetparameter{boxdir}{2}

```

tate math: 11
 4, ∞, ↑, ↓
 3

6.3 Overridden box primitives

To cope with multiple directions, the following primitives are overridden by Lua_{TeX}-ja, using `\protected\def`.

Table 8. Differences between horizontal JFM's shipped with Lua \TeX -ja



```
\unhbox<num>, \unvbox<num>, \unhcopy<num>, \unvcopy<num>
\vadjust{<material>}
\insert<number>{<material>}
\lastbox
\raise<dimen><box>, \lower<dimen><box> etc., \vcenter
\vcenter
```

7 Font Metric and Japanese Font

7.1 `\jfont`

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

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

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

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

`jfm=<name>`

Specify the name of (horizontal) JFM. If specified JFM has not been loaded, Lua \TeX -ja search and load a file named `jfm-<name>.lua`.

The following JFM's are shipped with Lua \TeX -ja:

`jfm-ujis.lua` A standard JFM in Lua \TeX -ja. This JFM is based on `upnmlminr-h.tfm`, a metric for UTF/OTF package that is used in `up \TeX` . When you use the `luatexja-otf` package, you should use this JFM.

`jfm-jis.lua` A counterpart for `jis.tfm`, “JIS font metric” which is widely used in `p \TeX` . A major difference between `jfm-ujis.lua` and this `jfm-jis.lua` is that most characters under `jfm-ujis.lua` are square-shaped, while that under `jfm-jis.lua` are horizontal rectangles.


```

1 \ltjsetParameter{differentjfm=both}
2 \jfont\F=file:KozMinPr6N-Regular.otf:jfm=ujis
3 \jfont\G=file:KozGoPr6N-Medium.otf:jfm=ujis
4 \jfont\H=file:KozGoPr6N-Medium.otf:jfm=ujis;jfmvar=hoge
5 \F ) {\G 【】 } ( % halfwidth space
6   ) {\H 『』 } ( % fullwidth space
7
8 ほげ, {\G 「ほげ」 } (ほげ) \par
9 ほげ, {\H 「ほげ」 } (ほげ) % pTeX-like
10
11 \ltjsetParameter{differentjfm=paverage}

```

) 【】 () 『』 ()
ほげ, 「ほげ」 (ほげ)
ほげ, 「ほげ」 (ほげ)

Figure 1. Example of jfmvar key

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

```

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

```

Figure 2. Kerning information and [kanjiskip](#)

`jfm-min.lua` A counterpart for `min10.tfm`, which is one of the default Japanese font metric shipped with pTeX.

The difference among these three JFM's is shown in Table 8.

`jfmvar=<string>`
Sometimes there is a need that

■ **Using kerning information in a font** Some fonts have information for inter-glyph spacing. This version of LuaTeX-ja treats kerning spaces like an italic correction; any glue and/or kern from the JFM and a kerning space can coexist. See Figure 2 for detail.

Note that in `\setmainjfont` etc. which are provided by `luatexja-fontspec` package, kerning option is set *off* (`Kerning=0ff`) by default, because of the compatibility with previous versions of LuaTeX-ja.

■ **extend and slant** The following setting can be specified as OpenType font features:

`extend=<extend>` expand the font horizontally by `<extend>`.

`slant=<slant>` slant the font.

Table 9. Differences between vertical JFMs shipped with LuaTeX-ja

ま	子	ん	あ	◆	ち	
し	が	る	る	◆	よ	
た	な	お	日	◆	つ	漢
.	っ	使	モ	◆	と	
	て	い	モ	◆	!	つ
	泣	で	ち	◆		
き	迷	や		◆	何	

Blue: `jfm-ujisv.lua`
Red: `jfm-tmin.lua`

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

```

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

```

あいうえお
あいうABC

7.2 \tfont

...

7.3 Prefix psft

Besides “file:” and “name:” prefixes which are introduced in the `luaotfload` package, LuaTeX-ja adds “psft:” prefix in `\font` (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 “+jp90”, have no meaning in name-only fonts using “psft:” prefix, because we can't expect what fonts are actually used by the PDF reader. Note that `extend` and `slant` settings (see above) are supported with `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-Japan1-6 CID-keyed font. One can specify `cid key` to use other CID-keyed non-embedded fonts for Chinese or Korean typesetting.

```

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

```

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 4 values written in the sample code above. Specifying other values, e.g.,

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

produces the following error:

```

1 ! Package luatexja Error: bad cid key `Adobe-Japan2'.
2
3 See the luatexja package documentation for explanation.
4 Type H <return> for immediate help.
5 <to be read again>

```

```

6          \par
7 1.78
8
9 ? h
10 I couldn't find any non-embedded font information for the CID
11 `Adobe-Japan2'. For now, I'll use `Adobe-Japan1-6'.
12 Please contact the LuaTeX-ja project team.
13 ?

```

7.4 Structure of a JFM file

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

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

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

`dir`= \langle *direction* \rangle (required)

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

`zw`= \langle *length* \rangle (required)

The amount of the length of the “full-width”.

`zh`= \langle *length* \rangle (required)

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

`kanjiskip`={ \langle *natural* \rangle , \langle *stretch* \rangle , \langle *shrink* \rangle } (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 0pt). Note that \langle *stretch* \rangle and \langle *shrink* \rangle fields are in design-size unit too.

`xkanjiskip`={ \langle *natural* \rangle , \langle *stretch* \rangle , \langle *shrink* \rangle } (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`={ \langle *character* \rangle , ... } (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 **J**A**char** which do not belong any character classes other than 0 are in the character class 0 (hence, the character class 0 contains most of **J**A**chars**). In the list, character(s) can be specified in the following form:

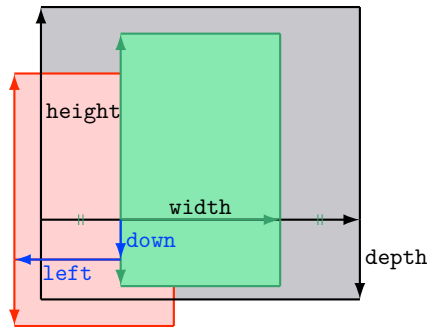
- a Unicode code point
- the character itself (as a Lua string, like 'あ')
- a string like 'あ*' (the character followed by an asterisk)
- several “imaginary” characters (We will describe these later.)

`width`= \langle *length* \rangle , `height`= \langle *length* \rangle , `depth`= \langle *length* \rangle , `italic`= \langle *length* \rangle (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 10.

Direction of JFM	'yoko' (horizontal)	'tate' (vertical)
width field	the width of the “real” glyph	1.0 (full-width)
height field	the height of the “real” glyph	0.5 (half-width)
depth field	the depth of the “real” glyph	0.5 (half-width)
italic field	0.0	

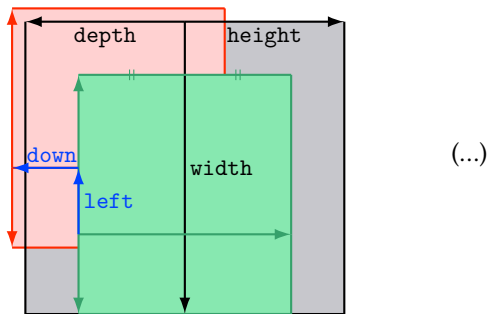
Table 10. Default values of width field and other fields



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 3. The position of the real glyph (horizontal Japanese fonts)



(...)

Figure 4. The position of the real glyph (vertical Japanese fonts)

`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 Figures 3 and 4.

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 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 `<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 $[-2, 2]$ (treated as 0 if omitted), and this is used only in line adjustment with `priority` by `luatexja-adjust` (see Subsection 11.3). Higher value means the glue is easy to stretch, and is also easy to shrink.

`kanjiskip_natural=<num>`, `kanjiskip_stretch=<num>`, `kanjiskip_shrink=<num>`

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

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

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

Hence we have the following result:

```

1 \leavevmode
2 \ltjsetparameter{kanjiskip=0pt plus 3\zw}          あ 「い」 う, え お|
3 \vrule\hbox to 15\zw{あ 「い」 う, えお}\vrule

```

`end_stretch=<kern>`, `end_shrink=<kern>`

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

```

[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 = { '。', '匕' },
  align = 'left', left = 0.0, down = 0.0,
  width = 0.5, height = 0.88, depth = 0.12, italic=0.0,
},

```

Now consider the following input/output:

```

1 \jfont\fontfile:KozMinPr6N-Regular.otf:jfm=test;+hwid
2 \setbox0\hbox{\a 匕漢}                                15.0pt
3 \the\wd0

```

Now we look why the above source outputs 15 pt.

1. The character “匕” is converted to its half width form “匕” by `hwid` feature.
2. According to the JFM, the character class of “匕” 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 *after applying font features by luaotfload*.

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

```

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

```

Table 11. Commands for Japanese math fonts

Japanese fonts	alphabetic fonts
$\backslash\text{jfam} \in [0, 256)$	$\backslash\text{fam}$
$\text{jatextfont}=\langle\text{jfam}\rangle,\langle\text{jfont_cs}\rangle$	$\text{textfont}\langle\text{fam}\rangle=\langle\text{font_cs}\rangle$
$\text{jascriptfont}=\langle\text{jfam}\rangle,\langle\text{jfont_cs}\rangle$	$\text{scriptfont}\langle\text{fam}\rangle=\langle\text{font_cs}\rangle$
$\text{jascriptscriptfont}=\langle\text{jfam}\rangle,\langle\text{jfont_cs}\rangle$	$\text{scriptscriptfont}\langle\text{fam}\rangle=\langle\text{font_cs}\rangle$

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

1. As the case of “ㄣ”, the ideographic full stop “。” is converted to its vertical form “ㄣ” (U+FE12) by `vert` feature.
2. The character class of “ㄣ”, according to the JFM is *zero*.
3. However, Lua \TeX -ja remembers that this “ㄣ” is obtained from “。” by font features. The character class of “。” is *non-zero value*, namely, 2000.
4. Hence the ideographic full stop “。” in above belongs the character class 2000.

■ **Imaginary characters** As described before, you 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.

'parbdd'

The beginning of an (indented) paragraph.

'jcharbdd'

A boundary between **J**Achar and anything else (such as **AL**char, kern, glue, ...).

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

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

7.5 Math font family

\TeX handles fonts in math formulas by 16 font families⁷, and each family has three fonts: `\textfont`, `\scriptfont` and `\scriptscriptfont`.

Lua \TeX -ja’s handling of Japanese fonts in math formulas is similar; Table 11 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, you can set both `\fam` and `\jfam` to the same value. Here $\langle\text{jfont_cs}\rangle$ in the argument of `jatextfont` etc. is a control sequence which is defined by `\jfont`, i.e., a *horizontal* Japanese font.

7.6 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.

`luatexja.load_jfm` callback

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

⁷Omega, Aleph, Lua \TeX and ϵ - μ p \TeX can handles 256 families, but an external package is needed to support this in plain \TeX and \LaTeX .

```

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

```

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

An example of this callback is the `ltjarticle` class, with forcefully assigning character class 0 to 'parbdd' in the JFM `jfm-min.lua`.

luatexja.define_jfont callback

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

```

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

```

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

size The font size specified at `\jfont` in scaled points ($1\text{ sp} = 2^{-16}\text{ pt}$).

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 $i \in \omega$, `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:

$$\begin{cases} 0 & \text{'left' (default)} \\ 0.5 & \text{'middle'} \\ 1 & \text{'right'} \end{cases}$$

- For $j \in \omega$, `[j]` stores a kern or a glue which will be inserted between character class i and class j .

If a kern will be inserted, the value of this field is `[j]={false, ⟨kern_node⟩, ⟨ratio⟩}`, where `⟨kern_node⟩` is a node⁸. If a glue will be inserted, we have `[j]={false, ⟨spec_node⟩, ⟨ratio⟩, ⟨icflag⟩}`, where `⟨spec_node⟩` is also a node, and `⟨icflag⟩ = from_jfm + ⟨priority⟩`.

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:

```

1 function (<number> char_class, <table> jfont_info, <number> chr_code)
2   if char_class~=0 then return char_class
3   else

```

⁸This version of `LuaTeX-ja` uses "direct access model" for accessing nodes, if possible.

```

4     ....
5     return (<number> new_char_class or 0)
6 end
7 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 **J**Achar *glyph_node*, to adjust its dimension and position.

```

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

```

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

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

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

Then, the position of glyphs is shifted up by

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

8 Parameters

8.1 \ltjsetparameter

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

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

The following is the list of parameters which can be specified by the `\ltjsetparameter` command. `[<cs>]` indicates the counterpart in pTeX, and symbols beside each parameter has the following meaning:

- “ ”: values at the end of a paragraph or a hbox are adopted in the whole paragraph or the whole hbox.
- “ † ”: assignments are always global.

`\jcharwidowpenalty=<penalty>` [`\jcharwidowpenalty`]

Penalty value for suppressing orphans. This penalty is inserted just after the last **J**Achar which is not regarded as a (Japanese) punctuation mark.

`kcatcode` = { $\langle chr_code \rangle$, $\langle natural\ number \rangle$ }

An additional attributes which each character whose character code is $\langle chr_code \rangle$ has. At the present version, the lowermost bit of $\langle natural\ number \rangle$ indicates whether the character is considered as a punctuation mark (see the description of [jcharwidowpenalty](#) above).

`prebreakpenalty` = { $\langle chr_code \rangle$, $\langle penalty \rangle$ } [`\prebreakpenalty`]

Set a penalty which is inserted automatically before the character $\langle chr_code \rangle$, to prevent a line starts from this character. For example, a line cannot started with one of closing brackets “””, so LuaTeX-ja sets

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

by default.

pTeX has following restrictions on `\prebreakpenalty` and `\postbreakpenalty`, but they don't exist in LuaTeX-ja:

- Both `\prebreakpenalty` and `\postbreakpenalty` cannot be set for the same character.
- We can set `\prebreakpenalty` and `\postbreakpenalty` up to 256 characters.

`postbreakpenalty` = { $\langle chr_code \rangle$, $\langle penalty \rangle$ } [`\postbreakpenalty`]

Set a penalty which is inserted automatically after the character $\langle chr_code \rangle$, to prevent a line ends with this character.

`jatextfont` = { $\langle jfam \rangle$, $\langle jfont_cs \rangle$ } [`\textfont` in TeX]

`jascriptfont` = { $\langle jfam \rangle$, $\langle jfont_cs \rangle$ } [`\scriptfont` in TeX]

`jascriptscriptfont` = { $\langle jfam \rangle$, $\langle jfont_cs \rangle$ } [`\scriptscriptfont` in TeX]

`yjabaselineshift` = $\langle dimen \rangle$

`yalbaselineshift` = $\langle dimen \rangle$ [`\ybaselineshift`]

`tjabaselineshift` = $\langle dimen \rangle$

`talbaselineshift` = $\langle dimen \rangle$ [`\tbaselineshift`]

`jaxspmode` = { $\langle chr_code \rangle$, $\langle mode \rangle$ }

Set whether inserting [xkanjiskip](#) is allowed before/after a **J**Achar whose character code is $\langle chr_code \rangle$. The followings are allowed for $\langle mode \rangle$:

- 0, inhibit** Insertion of [xkanjiskip](#) is inhibited before the character, nor after the character.
- 1, preonly** Insertion of [xkanjiskip](#) is allowed before the character, but not after.
- 2, postonly** Insertion of [xkanjiskip](#) is allowed after the character, but not before.
- 3, allow** Insertion of [xkanjiskip](#) is allowed both before the character and after the character. This is the default value.

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

`alxspmode` = { $\langle chr_code \rangle$, $\langle mode \rangle$ } [`\xspcode`]

Set whether inserting [xkanjiskip](#) is allowed before/after a **A**Lchar whose character code is $\langle chr_code \rangle$. The followings are allowed for $\langle mode \rangle$:

- 0, inhibit** Insertion of [xkanjiskip](#) is inhibited before the character, nor after the character.
- 1, preonly** Insertion of [xkanjiskip](#) is allowed before the character, but not after.
- 2, postonly** Insertion of [xkanjiskip](#) is allowed after the character, but not before.
- 3, allow** Insertion of [xkanjiskip](#) is allowed before the character and after the character. This is the default value.

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

`autospacing` = $\langle bool \rangle$ [`\autospacing`]

`autoxspacing` = $\langle bool \rangle$ [`\autoxspacing`]

`kanjiskip` = $\langle skip \rangle$ [`\kanjiskip`]

The default glue which inserted between two **J**Achars. Changing current Japanese font does not alter this parameter, as p \TeX .

If the natural width of this parameter is `\maxdimen`, Lua \TeX -ja uses the value which is specified in the JFM for current Japanese font (See Subsection 7.4).

`xkanjiskip` = $\langle skip \rangle$ [`\xkanjiskip`]

The default glue which inserted between a **J**Achar and an **AL**char. Changing current font does not alter this parameter, as p \TeX .

As `kanjiskip`, if the natural width of this parameter is `\maxdimen`, Lua \TeX -ja uses the value which is specified in the JFM for current Japanese font (See Subsection 7.4).

`differentjfm` = $\langle mode \rangle^\dagger$

Specify how glues/kerns between two **J**Achars 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` = $\langle ranges \rangle$

`kansujichar` = $\{ \langle digit \rangle, \langle chr_code \rangle \}$ [`\kansujichar`]

`direction` = $\langle dir \rangle$ (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.

8.2 `\ltjgetparameter`

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

```

1 \ltjgetparameter{differentjfm},
2 \ltjgetparameter{autospacing},
3 \ltjgetparameter{kanjiskip},
4 \ltjgetparameter{prebreakpenalty}{`} }.

```

paverage, 1, 0.0pt plus 0.4pt minus 0.4pt, 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.

- `\ltjgetparameter{jacharrange}{<range>}` returns 0 if “characters which belong to the character range *<range>* are **J**Achar”, 1 if “... are **AL**char”. Although there is no character range `—1`, specifying `—1` to *<range>* does not cause an error (returns 1).
- For an integer *<digit>* between 0 and 9, `\ltjgetparameter{kansujichar}{<digit>}` returns the character code of the result of `\kansuji<digit>`.
- `\ltjgetparameter{adjustdir}` returns a integer which represents the direction of the surrounding vertical list. As [direction](#), the return value 1 means *down-to-up* direction, 3 means *tate* direction (vertical typesetting), and 4 means *yoko* direction (horizontal typesetting).
- For an integer *<reg_num>* between 0 and 65535, `\ltjgetparameter{boxdim}{<reg_num>}` returns the direction of `\box<reg_num>`. If this box register is void, the returned value is zero.
- The following parameter names *cannot be specified* in `\ltjgetparameter`.

`jatextfont, jascriptfont, jascriptscriptfont, jacharrange`

- `\ltjgetparameter{chartorange}{<chr_code>}` returns the range number which *<chr_code>* belongs to (although there is no parameter named “chartorange”).

If *<chr_code>* is between 0 and 127, this *<chr_code>* does not belong to any character range. In this case, `\ltjgetparameter{chartorange}{<chr_code>}` returns `—1`.

Hence, one can know whether *<chr_code>* is **J**Achar or not by the following:

```
\ltjgetparameter{jacharrange}{\ltjgetparameter{chartorange}{<chr_code>}}
% 0 if JAchar, 1 if ALchar
```

- Because the returned value is string, the following conditionals do not work if [kanjiskip](#) (or [xkanjiskip](#)) has the stretch part or the shrink part.

```
\ifdim\ltjgetparameter{kanjiskip}>\z@ ... \fi
\ifdim\ltjgetparameter{xkanjiskip}>\z@ ... \fi
```

The correct way is using a temporary register.

```
\@tempkipa=\ltjgetparameter{kanjiskip} \ifdim\@tempkipa>\z@ ... \fi
\@tempkipa=\ltjgetparameter{xkanjiskip} \ifdim\@tempkipa>\z@ ... \fi
```

9 Other Commands for plain T_EX and L^AT_EX 2_ε

9.1 Commands for compatibility with pT_EX

The following commands are implemented for compatibility with pT_EX. 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, \sjis, \jis, \kansuji`

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

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

To change characters of Chinese numerals for each digit, set [kansujichar](#) parameter:

```
1 \ltjsetparameter{kansujichar={1,`壹}}
2 \ltjsetparameter{kansujichar={7,`漆}}
3 \ltjsetparameter{kansujichar={0,`零}}
4 \kansuji1701           壹漆零壹
```

9.2 `\inhibitglue`

`\inhibitglue` suppresses the insertion of **JAg**lue. The following is an example, using a special JFM that there will be a glue between the beginning of a box and “あ”, and also between “あ” and “ウ”.

<pre>1 \jfont\g=file:KozMinPr6N-Regular.otf:jfm=test \g 2 \fbox{\hbox{あウあ\inhibitglue ウ}} 3 \inhibitglue\par\noindent あ1 4 \par\inhibitglue\noindent あ2 5 \par\noindent\inhibitglue あ3 6 \par\hrule\noindent あoff\inhibitglue ice</pre>	<p style="text-align: center;">あ ウあウ</p> <hr/> <p>あ 1 あ 2 あ 3</p> <hr/> <p>あ office</p>
---	---

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

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

9.3 `\ltjdeclarealtfont`

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

$$\ltjdeclarealtfont\langle base_font_cs \rangle \langle alt_font_cs \rangle \{ \langle range \rangle \}$$

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

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

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

For example, if `\hoge` uses `jfm-ujis.lua`, the standard JFM of Lua \TeX -ja, then

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

does

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

10 Commands for $\LaTeX 2_{\epsilon}$

10.1 Patch for NFSS2

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

additional dimensions

Like p $\LaTeX 2_{\epsilon}$, Lua \TeX -ja defines the following dimensions for information of current Japanese font:

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

and its `\normalsize` version:

`\Cht` (height), `\Cdp` (depth), `\Cwd` (width),
`\Cvs` (equals to `\baselineskip`), `\Chs` (equals to `\c wd`).

Note that `\c wd` and `\c HT` may differ from `\zw` and `\zh` respectively. On the one hand the former dimensions are determined from the character “あ”, 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 Lua \TeX -ja, distinction between alphabetic fonts and Japanese fonts are only made by their encodings. For example, encodings OT1 and T1 are encodings for alphabetic fonts, and Japanese fonts cannot have these encodings. These command define a new encoding scheme for Japanese font families.

`\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`

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`. Like `\fontfamily`, `\selectfont` is required to take an effect.

`\adjustbaseline`

In p \TeX 2 ϵ , `\adjustbaseline` sets `\tbaselineshift` to match the vertical center of “M” and that of “あ” in vertical typesetting:

$$\tbaselineshift \leftarrow \frac{(\text{M} + d_M) - (\text{あ} + d_{\text{あ}})}{2} + d_{\text{あ}} - d_M,$$

where a and d_a denote the height of “a” and the depth, respectively. In Lua \TeX -ja, this `\adjustbaseline` does same task, namely setting the [talbaselineshift](#) parameter.

`\fontfamily{⟨family⟩}`

As in \TeX 2 ϵ , this command changes current font family (alphabetic, Japanese, or both) to `⟨family⟩`. See Subsection 10.2 for detail.

`\DeclareAlternateKanjiFont{⟨base-encoding⟩}{⟨base-family⟩}{⟨base-series⟩}{⟨base-shape⟩}`

`{⟨alt-encoding⟩}{⟨alt-family⟩}{⟨alt-series⟩}{⟨alt-shape⟩}{⟨range⟩}`

As `\ltjdeclarealtfont` (Subsection 9.3), 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 5.

- 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.
- ...

As closing this subsection, we shall introduce an example of `\SetRelationFont` and `\userelfont`:

```

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

```

日本国民は、正当に選挙された国会における代表者を通じて行動し、……

Figure 5. An example of `\DeclareAlternateKanjiFont`

```

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

```

10.2 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 `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 `family`, and another case that `family` isn't recognized as a Japanese/alphabetic font family either.

■ **Recognition as Japanese font family** First, Whether Japanese font family will be changed is determined in following order. This order is very similar to `\fontfamily` in $\text{p}\mathcal{L}\text{T}\mathcal{E}\text{X}_2\epsilon$, but we re-implemented in Lua. We use an auxiliary list N_J .

1. If the family `family` has been defined already by `\DeclareKanjiFamily`, `family` is recognized as a Japanese font family. Note that `family` need not be defined under *current* Japanese font encoding.
2. If the family `family` has been listed in a list N_J , this means that `family` is not a Japanese font family.
3. If the `luatexja-fontspec` package is loaded, we stop here, and `family` is not recognized as a Japanese font family.

If the `luatexja-fontspec` package is *not* loaded, now `LuaTeX-ja` looks whether there exists a Japanese font encoding `enc` such that a font definition named `encfamily`.`fd` (the file name is all lowercase) exists. If so, `family` is recognized as a Japanese font family (the font definition file won't be loaded here). If not, `family` is not a Japanese font family, and `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 `family` has been listed in a list F_A , `family` is recognized as an alphabetic font family.
2. If the family `family` has been listed in a list N_A , this means that `family` is not an alphabetic font family.
3. If there exists an alphabetic font encoding such that the family `family` has been defined under it, `family` is recognized as an alphabetic font family, and to memorize this, `family` is appended to the list F_A .
4. Now `LuaTeX-ja` looks whether there exists an alphabetic font encoding `enc` such that a font definition named `encfamily`.`fd` (the file name is all lowercase) exists. If so, current alphabetic font family will be changed to `family` (the font definition file won't be loaded here). If not, current alphabetic font family won't be changed, and `family` is appended to the list N_A .

```

1 \jfontspec[
2   YokoFeatures={Color=007F00}, TateFeatures={Color=00007F},
3   TateFont=KozGoPr6N-Regular
4 ]{KozMinPr6N-Regular}
5 \hbox{\yoko 横組のテスト}\hbox{\tate 縦組のテスト}
6 \addjfontfeatures{Color=FF0000}
7 \hbox{\yoko 横組}\hbox{\tate 縦組}

```

横組のテスト

縦
組
の
テ
ス
ト

横組

縦
組

Figure 6. An example of TateFeatures etc.

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

The above order is very similar to `\fontfamily` in $\text{p}\text{L}\text{A}\text{T}\text{E}\text{X} 2_\epsilon$, but more complicated (clause 3.). This is because $\text{p}\text{L}\text{A}\text{T}\text{E}\text{X} 2_\epsilon$ is a *format* however `LuaTeX-ja` is not, hence `LuaTeX-ja` does not know calls of `\DeclareFontFamily` before itself is loaded.

■ **Remarks** Of course, there is a case that *family* is not recognized as a Japanese font family, nor an alphabetic font family. In this case, `LuaTeX-ja` treats “the argument *family* is wrong”, so set both current alphabetic and Japanese font family to *family*, to use the default family for font substitution.

11 Addon packages

`LuaTeX-ja` has several addon packages. These addons are written as $\text{L}\text{A}\text{T}\text{E}\text{X}$ packages, but `luatexja-otf` and `luatexja-adjust` can be loaded in plain `LuaTeX` by `\input`.

11.1 luatexja-fontspec

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

`CID=<name>`, `JFM=<name>`, `JFM-var=<name>`

These 3 keys correspond to `cid`, `jfm` and `jfmvar` keys for `\jfont` and `\tfont` respectively. See Subsections 7.1 and 7.3 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 7.3.

`Kanjiskip=<bool>`

`TateFeatures={<features>}`, `TateFont=`

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 demonstrarion is shown in Figure 6.

`YokoFeatures={<features>}`

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

```

1 \fontspec[
2   AltFont={
3     {Range="4E00-"67FF, Color=007F00},
4     {Range="6800-"9EFF, Color=0000FF},
5     {Range="3040-"306F, Font=KozGoPr6N-Regular},
6   }
7 ]{KozMinPr6N-Regular}
8 日本国民は、正当に選挙された国会における代表者を通じて行動し、われらとわれらの子孫のために、
9 諸国民との協和による成果と、わが国全土にわたつ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、わが国全土にわたつて自由のもたらす恵沢を確保し、……

Figure 7. An example of AltFont

AltFont

As `\ltjdeclarealtfont` (Subsection 9.3) and `\DeclareAlternateKanjiFont` (Subsection 10.1), with this key, one can typeset some Japanese characters by a different font and/or using different features. The `AltFont` feature takes a comma-separated list of comma-separated lists, as the following:

```

AltFont = {
  ...
  { Range=<range>, <features>},
  { Range=<range>, Font=<font name>, <features> },
  { Range=<range>, Font=<font name> },
  ...
}

```

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

■ **Remark on AltFont, YokoFeatures, TateFeatures keys** In `AltFont`, `YokoFeatures`, `TateFeatures` keys, one cannot specify per-shape settings such as `BoldFeatures`. For example,

```

AltFont = {
  { Font=HogeraMin-Light, BoldFont=HogeraMin-Bold,
    Range="3000-"30FF, BoldFeatures={Color=007F00} }
}

```

does *not* work. Instead, one have to write

```

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

```

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

11.2 luatexja-otf

This optional package supports typesetting characters in Adobe-Japan1 character collection (or other CID character collection, if the font is supported). The package `luatexja-otf` offers the following 2 low-level commands:

`\CID{⟨number⟩}`

Typeset a character whose CID number is $\langle number \rangle$.

`\UTF{⟨hex_number⟩}`

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

This package automatically loads `luatexja-ajmacros.sty`, which is slightly modified version of `ajmacros.sty`⁹. Hence one can use macros which sre defined in `ajmacros.sty`, such as `\aj半角`.

■ **Remarks** Characters by `\CID` and `\UTF` commands are different from ordinary characters in the following points:

- Always treated as **J**Achars.
- Processing codes for supporting OpenType features (e.g., glyph replacement and kerning) by the `luaotfload` package is not performed to these characters.

■ **Additional syntax of JFM** The package `luatexja-otf` extends the syntax of JFM; the entries of `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.

■ **IVS support** Recent fonts support Ideographic Variation Selector (IVS). It seems that `luaotfload` and `fontspec` packages do not support IVS, so we implemented IVS support in `luatexja-otf`. *IVS support is experimental; if you want to enable this, load `luatexja-otf` and execute the following:*

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

After executing the command above, you can use IVS like the following:

<pre>1 \Large 2 \fontspec{KozMinPr6N-Regular} 3 奈良県葛⁰⁰⁰⁰城市と，東京都葛⁰⁰⁰⁰飾区。 \ 4 こんにちは，渡 5 邊⁰⁰⁰⁰邊⁰⁰⁰¹邊⁰⁰⁰²邊⁰⁰⁰³邊⁰⁰⁰⁴ 6 邊⁰⁰⁰⁵邊⁰⁰⁰⁶邊⁰⁰⁰⁷邊⁰⁰⁰⁸邊⁰⁰⁰⁹ 7 邊^{000A}邊^{000B}邊^{000C}邊^{000D}邊^{000E} 8 さん。</pre>	<p>奈良県葛城市と，東京都葛飾区。 こんにちは，渡邊邊邊邊邊邊邊邊邊邊邊邊邊邊邊邊邊さん。</p>
--	--

Specifying glyph variants by IVS precedes glyph replacement by font features. For example, only “葛” in “葛西” is changed by font features `jp78` or `jp90`, which does not followed by any variation selector.

<pre>1 \def\TEST#1{% 2 {\fontspec[#1]{KozMinPr6N-Regular}% 3 葛⁰⁰⁰⁰城市，葛⁰⁰⁰⁰飾区，葛西}\ 4 指定なし：\TEST{}} 5 \texttt{jp78}：\TEST{CJKShape=JIS1978} 6 \texttt{jp90}：\TEST{CJKShape=JIS1990}</pre>	<p>指定なし：葛城市，葛飾区，葛西 <code>jp78</code>：葛城市，葛飾区，葛西 <code>jp90</code>：葛城市，葛飾区，葛西</p>
---	--

11.3 luatexja-adjust

(see Japanese version of this manual)

⁹Useful macros by iNOUE Koich!, for the `japanese-otf` package.

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

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

Figure 8. Line adjustment

11.4 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](#).)

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

1 東西線\ruby{妙典}{みょうでん}駅は……\	東西線 ^{みょうでん} 妙典駅は……
2 東西線の\ruby{妙典}{みょうでん}駅は……\	東西線 ^{みょうでん} の ^{妙典} 駅は……
3 東西線の\ruby{妙典}{みょうでん}という駅……\	東西線の ^{みょうでん} の ^{妙典} という駅……
4 東西線\ruby{葛西}{かさい}駅は……	東西線 ^{かさい} 葛西駅は……

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

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

1 東西線の\ruby{妙}{みょう}\ruby{典}{でん}駅は……	東西線 ^{みょう} の ^妙 ^典 駅は……
-------------------------------------	---

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

1 \ruby{妙 典}{みょう でん}\	
2 \ruby{葛 西}{か さい}\	^{みょうでん} ^か ^{さい} ^か ^{ぐら} ^ざ ^か
3 \ruby{神楽 坂}{かぐら ざか}	妙典 葛西 神楽坂

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

1 \vbox{\hspace=6\zw\noindent	
2 \hbox to 2.5\zw{\ruby{京 急 蒲 田}{けい きゆう かま た}}	
3 \hbox to 2.5\zw{\ruby{京 急 蒲 田}{けい きゆう かま た}}	
4 \hbox to 3\zw{\ruby{京 急 蒲 田}{けい きゆう かま た}}	
5 }	けい きゆう かま 京 急 蒲 た 田 京 急 か ま た 蒲 田 京 きゆう かま た 急 蒲 田

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.

1 \vbox{\hspace=8\zw\noindent	
2 \null\kern3\zw ……を\ruby{承}{うけたまわる}	うけたまわ ……を 承
3 \kern1\zw ……を\ruby{承}{うけたまわる}\	る ……を 承る
4 \null\kern5\zw ……を\ruby{承}{うけたまわる}	……を
5 }	うけたまわ 承 る

11.5 lltjext.sty

pl \TeX supplies additional macros for vertical writing in the `plext` package. The `lltjext` package which we want to describe here is the Lua \TeX -ja counterpart of the `plext` package.

tabular, array, minipage environments

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

```
\begin{tabular}<dir>[pos]{table spec} ... \end{tabular}
\begin{array}<dir>[pos]{table spec} ... \end{array}
\begin{minipage}<dir>[pos]{width} ... \end{minipage}
```

This option permits one of the following five values. If none of them is specified, the direction inside the environment is same as that outside the environment.

y *yoko* direction (horizontal writing)
t *tate* direction (vertical writing)
z *utod* direction if direction outside the env. is *tate*.
d *dtou* direction
u *utod* direction

```
\parbox<dir>[pos]{width}{contents}
\parbox command is also extended by <dir>.
```

```
\pbox<dir>[width][pos]{contents}
This commands typeset contents in LR-mode, in dir direction. If width is positive, the width of the box becomes this width. In this case, contents will be aligned ...
```

picture environment

```
\rensuji[pos]{contents}, \rensujiskip
```

```
\Kanji{counter_name}
```

```
\kasen{contents}, \bou{contents}, \boutenchar
```

参照番号

Part III

Implementations

12 Storing Parameters

12.1 Used dimensions, attributes and whatsit nodes

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

`\jQ` (dimension) `\jQ` is equal to $1\text{ Q} = 0.25\text{ mm}$, 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 “齒” which equals to 0.25 mm and used in Japanese phototypesetting. This `\jH` is the same `\dimen` register as `\jQ`.

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

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

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

`\ltj@curjfont` (attribute) The font index of current Japanese font for horizontal direction.

`\ltj@curtfnt` (attribute) The font index of current Japanese font for vertical direction.

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

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

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

`\ltj@tblshift` (attribute)

`\ltj@tkblshift` (attribute)

`\ltj@autospc` (attribute) Whether the auto insertion of [kanjiskip](#) is allowed at the node.

`\ltj@autoxspc` (attribute) Whether the auto insertion of [xkanjiskip](#) is allowed at the node.

`\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 ($\backslash/$), or from kerning information of a Japanese font. These kerns are “ignored” in the insertion process of **JAgglue**, unlike explicit $\backslash\kern$.
- packed* (2)**
- kinsoku* (3)** Penalties inserted for the word-wrapping process (*kinsoku shori*) of Japanese characters.
- (from_jfm - 2)*–*(from_jfm + 2)* (4–8)** Glues/kerns from JFM.
- kanji_skip* (9), *kanji_skip_jfm* (10)** Glues from [kanjiskip](#).
- xkanji_skip* (11), *xkanji_skip_jfm* (12)** Glues from [xkanjiskip](#).
- processed* (13)** Nodes which is already processed by
- ic_processed* (14)** Glues from an italic correction, but already processed in the insertion process of **JAgglues**.
- boxbdd* (15)** Glues/kerns that inserted just the beginning or the ending of an hbox or a paragraph.

`\ltj@kcat i` (attribute) Where *i* is a natural number which is less than 7. These 7 attributes store bit vectors indicating which character block is regarded as a block of **JAchars**.

`\ltj@dir` (attribute) ***dir_node_auto* (128)**
***dir_node_manual* (256)**

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

inhibitglue Nodes for indicating that $\backslash\text{inhibitglue}$ 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 **JChar** which the callback process of `luaotfload` 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 the callback process of `luaotfload`. Nodes of this type is used in $\backslash\text{CID}$, $\backslash\text{UTF}$ and $\backslash\text{IVS}$ support.

replace_vs Similar to *char_by_cid* whatsits above. These nodes are for **ALchar** which the callback process of luaotfload 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 whatsits will be removed during the process of inserting **JAg**lues.

12.2 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 \setbox0=\hbox{%
3   \ltjsetparameter{kanjiskip=5pt}ほげほげ}   ふがふが.ほげほげ.ひよひよ
4 \box0.ひよひよ\par

```

As described in Subsection 8.1, 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:

```

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

```

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 list¹⁰.

These are two TeX count registers for maintaining information: `\ltj@@stack` for the stack level, and `\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 `\ltjsetparameter`, all data of the previous level is copied.

To resolve the problem mentioned in above paragraph “Background”, LuaTeX-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:

- 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*.

¹⁰ [Dev-luatex] `tex.currentgrouplevel`, a post at 2008/8/19 by Jonathan Sauer.

```

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

```

Figure 9. Definition of parameter setting commands

- 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 $s + 1$.
- 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 .

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

12.3 Lua functions of the stack system

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

The following function can be used to store data into a stack:

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

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

Stored data can be obtained as the return value of

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

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

12.4 Extending Parameters

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

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

Hence, to add a key in `\ltjsetparameter`, one only have to add a key whose prefix is `ltj` and whose family is `japaram`, as the following.

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

`\ltjsetparameter` and `\ltjglobalsetparameter` automatically sets `luatexja.isglobal`. Its meaning is the following.

$$\text{luatexja.isglobal} = \begin{cases} \text{'global'} & \text{global} \\ \text{' ' } & \text{local} \end{cases} \quad (1)$$

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.

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

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

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

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

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

For parameters that need an additional argument, one also have to execute the \TeX code like

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

to indicate that “the parameter `fuga` needs an additional argument”.

13 Linebreak after a Japanese Character

13.1 Reference: behavior in p \TeX

In p \TeX , a line break after a Japanese character doesn't emit a space, since words are not separated by spaces in Japanese writings. However, this feature isn't fully implemented in Lua \TeX -ja due to the specification of callbacks in Lua \TeX . To clarify the difference between p \TeX and Lua \TeX , We briefly describe the handling of a line break in p \TeX , in this subsection.

p \TeX 's input processor can be described in terms of a finite state automaton, as that of \TeX in Section 2.5 of [1]. The internal states are as follows:

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

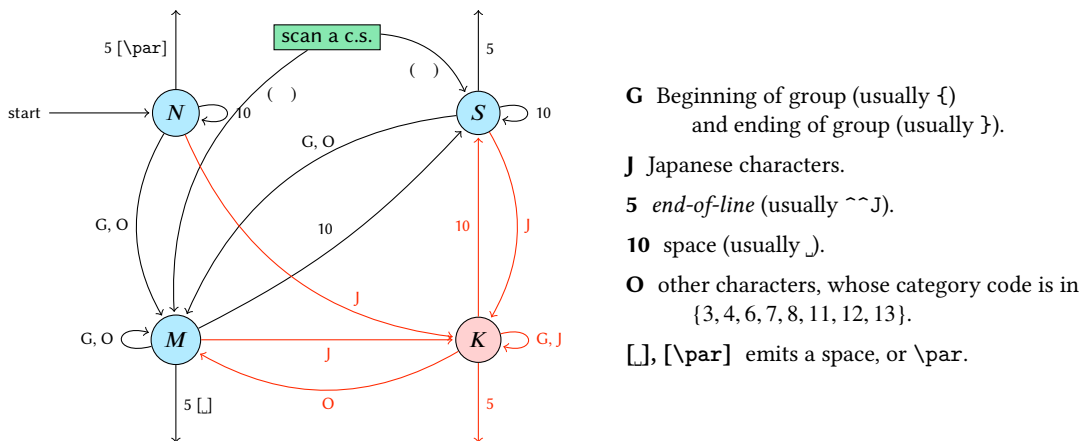
The first three states— N , S , and M —are as same as \TeX 's input processor. State K is similar to state M , and is entered after Japanese characters. The diagram of state transitions are indicated in Figure 10. Note that p \TeX doesn't leave state K after “beginning/ending of a group” characters.

13.2 Behavior in Lua \TeX -ja

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

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

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



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

Figure 10. State transitions of $\text{p}\TeX$'s input processor

A character U+FFFFF (its category code is set to 14 (*comment*) by $\text{Lua}\TeX$ -ja) is appended to an input line, *before LuaTeX actually process it*, if and only if the following three conditions are satisfied:

1. The category code of `\endlinechar`¹¹ is 5 (*end-of-line*).
2. The category code of U+FFFFF itself is 14 (*comment*).
3. The input line matches the following “regular expression”:

(any char) (**J**Achar)({catcode = 1} \cup {catcode = 2})

■ **Remark** The following example shows the major difference from the behavior of $\text{p}\TeX$.

```

1 \fontspec[Ligatures=TeX]{Linux Libertine O}
2 \ltjsetparameter{autospacing=false}
3 \ltjsetparameter{jacharrange=-6}xあ          xyz\`u
4 y\ltjsetparameter{jacharrange=+6}z\`
5 u

```

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 **JA**char.

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 `process_input_buffer` callback, “あ” is considered as an **JA**char. Since line 3 ends with an **JA**char, the comment character U+FFFFF is appended to this line, and hence the linebreak immediately after this line is ignored.
- When line 4 is processed by `process_input_buffer` callback, “\`” is considered as an **AL**char. Since line 4 ends with an **AL**char, the linebreak immediately after this line emits a space.

¹¹Usually, it is `<return>` (whose character code is 13).

14 Patch for the listings Package

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

In LuaTeX-ja, the problem is resolved by using the `process_input_buffer` callback. The callback function inserts the output command (active character `U+FFFFF`) before each letter above `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 LuaTeX-ja were loaded, then the patch `lltjp-listings` is loaded automatically at `\begin{document}`.

14.1 Notes and additional keys

■ **Escaping to \LaTeX** We used the `process_input_buffer` callback to output **J**Achars. But it has a drawback; any commands whose name contains a **J**Achar cannot be used in any “escape to \LaTeX ”.

Consider the following input:

```
\begin{lstlisting}[escapechar=\#]
#\ほげ_x ひよ#
\end{lstlisting}
```

The line 2 is transformed by the callback to

```
#\ほげ_xひよ#
```

before the line is actually processed. In the escape (between the character “#”), the category code of `U+FFFFF` is set to 9 (*ignored*). Hence the control symbol “`\ほげ`” will be executed, instead of “`\ほげ`”.

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

`vsraw` is a key which takes a boolean value, and its default value is false.

- If the `vsraw` key is true, then variation selectors are “combined” with the previous character.

```
1 \begin{lstlisting}[vsraw=true]
2 葛080城市, 葛091飾区, 葛西           1 葛080城市, 葛091飾区, 葛西
3 \end{lstlisting}
```

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

```
1 \begin{lstlisting}[vsraw=false,
2   vscmd=\ltjlistingsvsstdcmd]
3 葛080城市, 葛091飾区, 葛西           1 葛080城市, 葛091飾区, 葛西
4 \end{lstlisting}
```

For example, the following code is the setting of the `vscmd` key in this document.

```
1 \def\IVSA#1#2#3#4#5{%
2   \textcolor{blue}{\raisebox{3.5pt}{\tt%
3     \fboxsep=0.5pt\fbox{\tiny \oalign{0#1#2\cr#3#4#5\cr}}}}%
4 }
5 {\catcode`\%=11
6   \gdef\IVSB#1{\expandafter\IVSA\directlua{
7     local cat_str = luatexbase.catcodetables['string']
8     tex.sprint(cat_str, string.format('%X', 0xE00EF+#1))
9   }}}
10 \lstset{vscmd=\IVSB}
```

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

■ **The doubleletterspace key** Even the column format is [c]fixed, sometimes characters are not vertically aligned. The following example is typeset with `basewidth=2em`, and you'll see the leftmost “H” are not vertically aligned.

```
1 : H :
2 : H H H H :
```

`lltjp-listing` adds the `doubleletterspace` key (not activated by default, for compatibility) to improve the situation, namely doubles inter-character space in each output unit. With this key, the above input now produces better output.

```
1 : H :
2 : H H H H :
```

14.2 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 `lltjp-listings` introduces a new flag `\lst@ifkanji`, which indicates whether the previous character is a Japanese character or not. For illustration, we introduce following classes of characters:

	Letter	Other	Kanji	Open	Close
<code>\lst@ifletter</code>	T	F	T	F	T
<code>\lst@ifkanji</code>	F	F	T	T	F
Meaning	char in an identifier	other alphabet	most of Japanese char	opening brackets	closing brackets

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:

		Next				
		Letter	Other	Kanji	Open	Close
Prev	Letter	collects	_____ outputs _____	collects		
	Other	outputs	collects	_____ outputs _____	collects	
	Kanji	_____ outputs _____	collects	_____	collects	
	Open	_____ collects _____	_____ outputs _____	collects	_____	
	Close	_____ outputs _____	_____ outputs _____	_____ outputs _____	collects	collects

In the above table,

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

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

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

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

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

15 Cache Management of Lua \TeX -ja

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

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

15.1 Use of cache

Lua \TeX -ja uses the following cache:

```
ltj-cid-auto-adobe-japan1.lua
```

The font table of a CID-keyed non-embedded Japanese font. This is loaded in every run. It is created from three CMaps, `UniJIS2004-UTF32-{H,V}` and `Adobe-Japan1-UCS2`, and this is why these two CMaps are needed in the first run of Lua \TeX -ja.

Similar caches are created as Table 12, if you specified `cid` key in `\jfont` to use other CID-keyed non-embedded fonts for Chinese or Korean, as in Page 25.

```
extra_***.lua
```

This file stores the table which stores the following.

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

The following is the structure of the that table.

```
return {
  {
    [10955]={ -- U+2ACB "Subset Of Above Not Equal To"
      [65024]=983879, -- <2ACB FE00>
      ["vwidth"]=0.98, -- vertical width
    },
    [37001]={ -- U+9089 "邊"
      [0]=37001, -- <9089 E0100>
```

Table 12. cid key and corresponding files

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

```

    991049,    -- <9089 E0101>
    ...
    ["vert"]=995025, -- vertical variant
  },
  ...
},
["chksum"]="FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF", -- checksum of the fontfile
["version"]=2, -- version of the cache
}

```

ltj-jisx0208.{luc|lub}

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

15.2 Internal

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

`save_cache(filename, data)`

Save a non-nil table *data* into a cache *filename*. Both the text form *filename*.lua and its binary version are created or updated.

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

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

`load_cache(filename, outdate)`

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

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

References

- [1] Victor Eijkhout. *T_EX by Topic, A T_EXnician's Reference*, Addison-Wesley, 1992.
- [2] C. Heinz, B. Moses. The Listings Package.
- [3] Takuji Tanaka. upTeX—Unicode version of pTeX with CJK extensions, TUG 2013, October 2013.
http://tug.org/tug2013/slides/TUG2013_upTeX.pdf
- [4] Thor Watanabe. Listings - MyTeXpert. <http://mytexpert.osdn.jp/index.php?Listings>
- [5] W3C Japanese Layout Task Force (ed). Requirements for Japanese Text Layout (W3C Working Group Note), 2011, 2012. <http://www.w3.org/TR/jlreq/>
- [6] 乙部 厳己. min10 フォントについて.
<http://argent.shinshu-u.ac.jp/~otobe/tex/files/min10.pdf>
- [7] 日本工業規格 (Japanese Industrial Standard). JIS X 4051, 日本語文書の組版方法 (Formatting rules for Japanese documents), 1993, 1995, 2004.
- [8] 濱野尚人, 田村明史, 倉沢良一. T_EX の出版への応用—縦組み機能の組み込み—.
.../texmf-dist/doc/ptex/base/ptexdoc.pdf
- [9] Hisato Hamano. *Vertical Typesetting with T_EX*, TUGBoat **11**(3), 346–352, 1990.
- [10] International Organization for Standardization. ISO 32000-1:2008, *Document management – Portable document format – Part 1: PDF 1.7*, 2008.
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=51502