

The fontspec package

Font selection for Xe_ΛT_EX and Lua_ΛT_EX

WILL ROBERTSON

With contributions by Khaled Hosny,
Philipp Gesang, Joseph Wright, and others.

<http://latex3.github.io/fontspec/>

2026/08/11 v2.9h

Contents

I	Getting started	5
1	History	5
2	Introduction	5
2.1	Acknowledgements	5
3	Package loading and options	6
3.1	Font encodings	6
3.2	Maths fonts adjustments	6
3.3	Configuration	6
3.4	Warnings	7
4	Interaction with $\text{\LaTeX} 2_{\epsilon}$ and other packages	7
4.1	Commands for old-style and lining numbers	7
4.2	Italic small caps	7
4.3	Emphasis and nested emphasis	7
4.4	Strong emphasis	7
II	General font selection	9
1	Main commands	9
2	Font selection	10
2.1	By font name	10
2.2	By file name	11
2.3	By custom file name using a .fontspec file	12
2.4	Querying whether a font ‘exists’	13

3	Commands to select font families	14
4	Commands to select single font faces	14
4.1	More control over font shape selection	15
4.2	Specifically choosing the NFSS family	17
4.3	Choosing additional NFSS font faces	18
4.4	Math(s) fonts	19
5	Miscellaneous font selecting details	20
III	Selecting font features	21
1	Default settings	21
2	Working with the currently selected features	22
2.1	Priority of feature selection	23
3	Different features for different font shapes	23
4	Selecting fonts from TrueType Collections (TTC files)	24
5	Different features for different font sizes	24
6	Font independent options	26
6.1	Colour	26
6.2	Scale	28
6.3	Interword space	29
6.4	Post-punctuation space	29
6.5	The hyphenation character	29
6.6	Optical font sizes	31
6.7	Font transformations	31
6.8	Letter spacing	33
7	Variable fonts	34
7.1	Optical font sizes	34
7.2	Weight	34
7.3	Width	34
7.4	Slant	34
7.5	Other axes	35
7.6	Instances	35
IV	OpenType	36
1	Introduction	36
1.1	How to select font features	36
1.2	How do I know what font features are supported by my fonts?	37
2	OpenType scripts and languages	38

2.1	Script and Language examples	38
3	OpenType font features	41
3.1	Tag-based features	41
3.2	CJK features	51
V	Commands for accents and symbols (‘encodings’)	55
1	A new Unicode-based encoding from scratch	55
2	Adjusting a pre-existing encoding	56
3	Summary of commands	58
VI	LuaTeX-only font features	59
1	Different font technologies and shapers	59
2	Custom font features	59
VII	Fonts and features with XeTeX	61
1	XeTeX-only font features	61
1.1	Mapping	61
1.2	Different font technologies: AAT, OpenType, and Graphite	61
1.3	Vertical typesetting	62
2	The Graphite renderer	62
3	macOS’s AAT fonts	62
3.1	Ligatures	63
3.2	Letters	63
3.3	Numbers	63
3.4	Contextuals	63
3.5	Vertical position	63
3.6	Fractions	63
3.7	Variants	64
3.8	Alternates	64
3.9	Style	64
3.10	CJK shape	64
3.11	Character width	64
3.12	Diacritics	64
3.13	Annotation	65

VIII	Customisation and programming interface	66
1	Defining new features	66
2	Defining new scripts and languages	67
3	Going behind fontspec's back	67
4	Renaming existing features & options	68
5	Programming interface	68
5.1	Variables	68
5.2	Functions for loading new fonts and families	69
5.3	Conditionals	69

Part I

Getting started

1 History

This package began life as a $\LaTeX$ interface to select system-installed macOS fonts in Jonathan Kew's $X_{\TeX}$, the first widely-used Unicode extension to $\TeX$. Over time, $X_{\TeX}$ was extended to support OpenType fonts and then was ported into a cross-platform program to run also on Windows and Linux.

More recently, $\text{Lua}\TeX$ is fast becoming the $\TeX$ engine of the day; it supports Unicode encodings and OpenType fonts and opens up the internals of $\TeX$ via the Lua programming language. Hans Hagen's $\text{Con}\TeX$ t Mk. IV is a re-write of his powerful typesetting system, taking full advantage of $\text{Lua}\TeX$'s features including font support; a kernel of his work in this area has been extracted to be useful for other $\TeX$ macro systems as well, and this has enabled `fontspec` to be adapted for $\LaTeX$ when run with the $\text{Lua}\TeX$ engine.

2 Introduction

The `fontspec` package allows users of either $X_{\TeX}$ or $\text{Lua}\TeX$ to load OpenType fonts in a $\LaTeX$ document. No font installation is necessary, and font features can be selected and used as desired throughout the document.

Without `fontspec`, it is necessary to write cumbersome font definition files for $\LaTeX$, since $\LaTeX$'s font selection scheme (known as the 'NFSS') has a lot going on behind the scenes to allow easy commands like `\emph` or `\bfseries`. With an uncountable number of fonts now available for use, however, it becomes less desirable to have to write these font definition (`.fd`) files for every font one wishes to use.

Because `fontspec` is designed to work in a variety of modes, this user documentation is split into separate sections that are designed to be relatively independent. Nonetheless, the basic functionality all behaves in the same way, so previous users of `fontspec` under $X_{\TeX}$ should have little or no difficulty switching over to $\text{Lua}\TeX$.

This manual can get rather in-depth, as there are a lot of details to cover. See the documents `fontspec-example.tex` for a complete minimal example to get started quickly.

2.1 Acknowledgements

This package could not have been possible without the early and continued support the author of $X_{\TeX}$, Jonathan Kew. When I started this package, he steered me many times in the right direction.

I've had great feedback over the years on feature requests, documentation queries, bug reports, font suggestions, and so on from lots of people all around the world. Many thanks to you all.

Thanks to David Perry and Markus Böhning for numerous documentation improvements and David Perry again for contributing the text for one of the sections of this manual.

Special thanks to Khaled Hosny, who was the driving force behind the support for Lua $\LaTeX$, ultimately leading to version 2.0 of the package.

3 Package loading and options

For basic use, no package options are required:

```
\usepackage{fontspec}
```

Package options will be introduced below; some preliminary details are discussed first. Package options are setup with the in-built $\LaTeX$ keyval options handler. This means that the package can be loaded more than once with different options without triggering an option clash error. The `config` and `no-config` option must be used in the first loading and are ignored later.

3.1 Font encodings

The package switches the `nfss` font encoding to `TU`. `TU` is a new Unicode font encoding, intended for both $\XTeX$ and $\LuaTeX$ engines, and automatically contains support for symbols covered by $\LaTeX$'s traditional `T1` and `TS1` font encodings (for example, `\%`, `\textbullet`, `\u`, and so on). Some additional features are provided by `fontspec` to customise some encoding details; see Part V on page 55 for further details.

`euenc` Pre-2017 behaviour is now obsolete. The `euenc` and `tuenc` package options are ig-
`tuenc` nored. Package authors and users who have referred explicitly to the encoding names `EU1` or `EU2` should update their code or documents. (See internal variable names described in Section 5 on page 68 for how to do this properly.)

3.2 Maths fonts adjustments

By default, `fontspec` adjusts $\LaTeX$'s default maths setup in order to maintain the correct Computer Modern symbols when the roman font changes. However, it will attempt to avoid doing this if another maths font package is loaded (such as `mathpazo` or the `unicode-math` package).

If you find that `fontspec` is incorrectly changing the maths font when it shouldn't be, `no-math` apply the `no-math` package option to manually suppress its behaviour here.

3.3 Configuration

If you wish to customise any part of the `fontspec` interface, this should be done by creating your own `fontspec.cfg` file, which will be automatically loaded if it is found by $\XTeX$ or $\LuaTeX$. A `fontspec.cfg` file is distributed with `fontspec` with a small number of defaults set up within it.

To customise `fontspec` to your liking, use the standard `.cfg` file as a starting point or write your own from scratch, then either place it in the same folder as the main document for isolated cases, or in a location that $\XTeX$ or $\LuaTeX$ searches by default; *e.g.* in $\MacTeX$: `~/Library/texmf/tex/latex/`.

`no-config` The package option `no-config` will suppress the loading of the `fontspec.cfg` file

under all circumstances. Both options must be used the first time `fontspec` is loaded and are ignored in later calls.

3.4 Warnings

This package can give some warnings that can be harmless if you know what you're doing. Use the `quiet` package option to write these warnings to the transcript (`.log`) file instead.

`silent` Use the `silent` package option to completely suppress these warnings if you don't even want the `.log` file cluttered up.

`verbose` Both options can also be used with `\Setkeys` in the document. Use the `verbose` option to get activate the warnings again.

4 Interaction with L^AT_EX 2_ε and other packages

This section documents some areas of adjustment that `fontspec` makes to improve default behaviour with L^AT_EX 2_ε and third-party packages.

4.1 Commands for old-style and lining numbers

`\oldstylenums` L^AT_EX's definition of `\oldstylenums` relies on strange font encodings. We provide a `fontspec`-compatible alternative and while we're at it also throw in the reverse option as well. Use `\oldstylenums{<text>}` to explicitly use old-style (or lowercase) numbers in `<text>`, and the reverse for `\liningnums{<text>}`.

4.2 Italic small caps

Support now provided by L^AT_EX 2_ε in 2020.

4.3 Emphasis and nested emphasis

Support now provided by L^AT_EX 2_ε in 2020.

4.4 Strong emphasis

`\strong` The `\strong` macro is used analogously to `\emph` but produces variations in weight. If you need it in environment form, use `\begin{strongenv}... \end{strongenv}`.

As with emphasis, this font-switching command is intended to move through a range of font weights. For example, if the fonts are set up correctly it allows usage such as `\strong{... \strong{...}}` in which each nested `\strong` macro increases the weight of the font.

`\strongfontdeclare` Currently this feature set is somewhat experimental and there is no syntactic sugar to easily define a range of font weights using `fontspec` commands. Use, say, the following to define first bold and then black (`k`) font faces for `\strong`:

```
\strongfontdeclare{\bfseries,\fontseries{k}\selectfont}
```

`\strongreset` If too many levels of `\strong` are reached, `\strongreset` is inserted. By default this is a no-op and the font will simply remain the same. Use `\renewcommand\strongreset{\mdseries}` to start again from the beginning if desired.

 An example for setting up a font family for use with `\strong` is discussed in [4.3.1 on page 19](#).

Part II

General font selection

1 Main commands

This section concerns the variety of commands that can be used to select fonts.

```
\setmainfont{<font>}[<font features>]  
\setsansfont{<font>}[<font features>]  
\setmonofont{<font>}[<font features>]
```

These are the main font-selecting commands of this package which select the standard fonts used in a document, as shown in Example 1. Here, the scales of the fonts have been chosen to equalise their lowercase letter heights. The `Scale` font feature will be discussed further in [Section 6 on page 26](#), including methods for automatic scaling. Note that further options may need to be added to select appropriate bold/italic fonts, but this shows the main idea.

Note that while these commands all look and behave largely identically, the default setup for font loading automatically adds the `Ligatures=TeX` feature for the `\setmainfont` and `\setsansfont` commands. These defaults (and further customisations possible) are discussed in [Section 1 on page 21](#).

```
\newfontfamily<cmd>{<font>}[<font features>]  
\setfontfamily<cmd>{<font>}[<font features>]  
\renewfontfamily<cmd>{<font>}[<font features>]  
\providefontfamily<cmd>{<font>}[<font features>]
```

These commands define new font family commands (like `\rmfamily`). The `new` command checks if `<cmd>` has been defined, and issues an error if so. The `renew` command checks if `<cmd>` has been defined, and issues an error if not. The `provide` command checks if `<cmd>` has been defined, and silently aborts if so. The `set` command never checks; use at your own risk.

```
\fontspec{<font>}[<font features>]
```

The plain `\fontspec` command is not generally recommended for document use. It is an ad hoc command best suited for testing and loading fonts on a one-off basis.

All of the commands listed above accept comma-separated `<font feature>=<option>` lists; these are described later:

- For general font features, see [Section 6 on page 26](#)
- For OpenType fonts, see [Part IV on page 36](#)
- For $\text{\XeTeX}$ -only general font features, see [Part VII on page 61](#)
- For $\text{\LuaTeX}$ -only general font features, see [Part VI on page 59](#)
- For features for AAT fonts in $\text{\XeTeX}$, see [Section 3 on page 62](#)

Example 1: Loading the default, sans serif, and monospaced fonts.

```
\setmainfont{texgyrebonum-regular.otf}
\setsansfont{lmsans10-regular.otf}[Scale=MatchLowercase]
\setmonofont{Inconsolatazi4-Regular.otf}[Scale=MatchLowercase]
```

Pack my box with five dozen liquor jugs	\rmfamily Pack my box with five dozen liquor jugs\par
Pack my box with five dozen liquor jugs	\sffamily Pack my box with five dozen liquor jugs\par
Pack my box with five dozen liquor jugs	\ttfamily Pack my box with five dozen liquor jugs

2 Font selection

In both Lua $\TeX$ and X $\TeX$, fonts can be selected (using the $\langle font \rangle$ argument in [Section 1](#)) either by ‘font name’ or by ‘file name’, but there are some differences in how each engine finds and selects fonts — don’t be too surprised if a font invocation in one engine needs correction to work in the other.

2.1 By font name

Fonts known to Lua $\TeX$ or X $\TeX$ may be loaded by their standard names as you’d speak them out loud, such as *Times New Roman* or *Adobe Garamond*. ‘Known to’ in this case generally means ‘exists in a standard fonts location’ such as `~/Library/Fonts` on macOS, or `C:\Windows\Fonts` on Windows. In Lua $\TeX$, fonts found in the `TEXMF` tree can also be loaded by name. In X $\TeX$, fonts found in the `TEXMF` tree can be loaded in Windows and Linux, but not on macOS.

The simplest example might be something like

```
\setmainfont{Cambria}[ ... ]
```

in which the bold and italic fonts will be found automatically (if they exist) and are immediately accessible with the usual `\textit` and `\textbf` commands.

The ‘font name’ can be found in various ways, such as by looking in the name listed in an application like *Font Book* on Mac OS X. Alternatively, $\TeX$ Live contains the `otfinfo` command line program, which can query this information; for example:

```
otfinfo -i `kpsewhich lmrroman10-regular.otf`
```

results in a line that reads:

```
Preferred family:    Latin Modern Roman
```

(The ‘preferred family’ name is usually better than the ‘family’ name.)

Lua $\TeX$ users only In order to load fonts by their name rather than by their filename (e.g., ‘Latin Modern Roman’ instead of ‘ec-lmr10’), you may need to run the script `luaotfload-tool`, which is distributed with the `luaotfload` package. Note that if you do not execute this script beforehand, the first time you attempt to typeset the process will pause for (up to) several minutes. (But only the first time.) Please see the `luaotfload` documentation for more information.

2.2 By file name

X_YTeX and LuaTeX also allow fonts to be loaded by file name instead of font name. When you have a very large collection of fonts, you will sometimes not wish to have them all installed in your system's font directories. In this case, it is more convenient to load them from a different location on your disk. This technique is also necessary in X_YTeX when loading OpenType fonts that are present within your TeX distribution, such as `/usr/local/texlive/2013/texmf-dist/fonts/opentype/public`. Fonts in such locations are visible to X_YTeX but cannot be loaded by font name, only file name; LuaTeX does not have this restriction. (If you for some reason want to restrict the fonts to the ones provided by your TeX distribution even though you are using LuaTeX you can use `KpseOnly` the `KpseOnly` option)

When selecting fonts by file name, any font that can be found in the default search paths may be used directly (including in the current directory) without having to explicitly define the location of the font file on disk.

Fonts selected by filename must include bold and italic variants explicitly, unless a `.fontspec` file is supplied for the font family (see [Section 2.3](#)). We'll give some first examples specifying everything explicitly:

```
\setmainfont{texgyrepagella-regular.otf}[
  BoldFont      = texgyrepagella-bold.otf ,
  ItalicFont     = texgyrepagella-italic.otf ,
  BoldItalicFont = texgyrepagella-bolditalic.otf ]
```

`fontspec` knows that the font is to be selected by file name by the presence of the `' .otf '` extension. An alternative is to specify the extension separately, as shown following:

```
\setmainfont{texgyrepagella-regular}[
  Extension      = .otf ,
  BoldFont       = texgyrepagella-bold ,
  ... ]
```

If desired, an abbreviation can be applied to the font names based on the mandatory `'font name'` argument:

```
\setmainfont{texgyrepagella}[
  Extension      = .otf ,
  UprightFont    = *-regular ,
  BoldFont       = *-bold ,
  ... ]
```

In this case `'texgyrepagella'` is no longer the name of an actual font, but is used to construct the font names for each shape; the `*` is replaced by `'texgyrepagella'`. Note in this case that `UprightFont` is required for constructing the font name of the normal font to use.

To load a font that is not in one of the default search paths, its location in the filesystem must be specified with the `Path` feature:

```
\setmainfont{texgyrepagella}[
  Path           = /Users/will/Fonts/ ,
  UprightFont    = *-regular ,
```

```

    BoldFont      = *-bold ,
    ... ]

```

Note that X_YTeX and LuaTeX are able to load the font without giving an extension, but fontspec must know to search for the file; this can be indicated by using the Path feature without an argument:

```

\setmainfont{texgyrepagella-regular}[
    Path, BoldFont = texgyrepagella-bold,
    ... ]

```

My preference is to always be explicit and include the extension; this also allows fontspec to automatically identify that the font should be loaded by filename.

In previous versions of the package, the Path feature was also provided under the alias ExternalLocation, but this latter name is now deprecated and should not be used for new documents.

2.3 By custom file name using a .fontspec file

When fontspec is first asked to load a font, a font settings file is searched for with the name '*fontname*.fontspec'.¹ If you want to *disable* this feature on a per-font basis, use the IgnoreFontspecFile font option.

The contents of this file can be used to specify font shapes and font features without having to have this information present within each document. Therefore, it can be more flexible than the alternatives listed above.

When searching for this .fontspec file, *fontname* is stripped of spaces and file extensions are omitted. For example, given `\setmainfont{TeX Gyre Adventor}`, the .fontspec file would be called `TeXGyreAdventor.fontspec`. If you wanted to transparently load options for `\setmainfont{texgyreadventor-regular.otf}`, the configuration file would be `texgyreadventor-regular.fontspec`.

N.B. that while spaces are stripped, the lettercase of the names should match.

This mechanism can be used to define custom names or aliases for your font collections. The syntax within this file follows from the `\defaultfontfeatures`, defined in more detail later but mirroring the standard fontspec font loading syntax. As an example, suppose we're defining a font family to be loaded with `\setmainfont{My Charis}`. The corresponding `MyCharis.fontspec` file would contain, say,

```

\defaultfontfeatures[My Charis]
{
    Extension = .ttf ,
    UprightFont    = CharisSILR,
    BoldFont       = CharisSILB,
    ItalicFont     = CharisSILI,
    BoldItalicFont = CharisSILBI,
    % <any other desired options>
}

```

¹Located in the current folder or within a standard texmf location.

The optional argument to `\defaultfontfeatures` must exactly match that requested by the font loading command (`\setmainfont`, etc.) — in particular note that spaces are significant here, so `\setmainfont{MyCharis}` will not ‘see’ the default font feature setting within the `.fontspec` file.

Finally, note that options for individual font faces can also be defined in this way. To continue the example above, here we colour the different faces:

```
\defaultfontfeatures[CharisSILR]{Color=blue}
\defaultfontfeatures[CharisSILB]{Color=red}
```

Such configuration lines could be stored either inline inside `MyCharis.fontspec` or within their own `.fontspec` files; in this way, `fontspec` is designed to handle ‘nested’ configuration options.

Where `\defaultfontfeatures` is being used to specify font faces by a custom name, the `Font` feature is used to set the filename of the font face. For example:

```
\defaultfontfeatures[charis]
{
  UprightFont = charis-regular,
  % <other desired options for all font faces in the family>
}

\defaultfontfeatures[charis-regular]
{
  Font = CharisSILR
  % <other desired options just for the ‘upright’ font>
}
```

The `fontspec` interface here is designed to be flexible to accommodate a variety of use cases; there is more than one way to achieve the same outcome when font faces are collected together into a larger font family.

2.4 Querying whether a font ‘exists’

`\IfFontExistsTF{<font name>}{<true branch>}{<>false branch>}`

The conditional `\IfFontExistsTF` is provided to test whether the `<font name>` exists or is loadable. If it is, the `<true branch>` code is executed; otherwise, the `<>false branch>` code is.

This command can be slow since the engine may resort to scanning the filesystem for a missing font. Nonetheless, it has been a popular request for users who wish to define ‘fallback fonts’ for their documents for greater portability.

In this command, the syntax for the `<font name>` is a restricted/simplified version of the font loading syntax used for `\fontspec` and so on. Fonts to be loaded by filename are detected by the presence of an appropriate extension (`.otf`, etc.), and paths should be included inline. E.g.:

```
\IfFontExistsTF{cmr10}{T}{F}
\IfFontExistsTF{Times New Roman}{T}{F}
\IfFontExistsTF{texgyrepagella-regular.otf}{T}{F}
\IfFontExistsTF{/Users/will/Library/Fonts/CODE2000.TTF}{T}{F}
```

The `\IfFontExistsTF` command is a synonym for the programming interface function `\fontspec_font_if_exist:nTF` (Section 5 on page 68).

3 Commands to select font families

For cases when a specific font with a specific feature set is going to be re-used many times in a document, it is inefficient to keep calling `\fontspec` for every use. While the `\fontspec` command does not define a new font instance after the first call, the feature options must still be parsed and processed.

For this reason, new commands can be created for loading a particular font family with the `\newfontfamily` command and variants, outlined in Section 1 on page 9 and demonstrated in Example 2. This macro should be used to create commands that would be used in the same way as `\rmfamily`, for example. If you would like to create a command that only changes the font inside its argument (i.e., the same behaviour as `\emph`) define it using regular L^AT_EX commands:

```
\newcommand\textnote[1]{\fontspec{\notefont #1}}
\textnote{This is a note.}
```

Note that the double braces are intentional; the inner pair is used to delimit the scope of the font change.

Comment for advanced users: The commands defined by `\newfontfamily` (and `\newfontface`; see next section) include their encoding information, so even if the document is set to use a legacy T_EX encoding, such commands will still work correctly. For example,

```
\documentclass{article}
\usepackage{fontspec}
\newfontfamily\unicodefont{Lucida Grande}
\usepackage{mathpazo}
\usepackage[T1]{fontenc}
\begin{document}
A legacy TeX font. {\unicodefont A unicode font.}
\end{document}
```

4 Commands to select single font faces

```
\newfontface<cmd>{\font}{[font features]}
\setfontface<cmd>{\font}{[font features]}
\renewfontface<cmd>{\font}{[font features]}
\providefontface<cmd>{\font}{[font features]}
```

Example 2: Defining new font families.

This is a <i>note</i> .	<pre>\newfontfamily\notefont{Kurier} \notefont This is a \emph{note}.</pre>
-------------------------	---

Sometimes only a specific font face is desired, without accompanying italic or bold variants being automatically selected. This is common when selecting a font for a very particular context within the document. For instance, say that a particular swash font is desired that isn't part of the document font setup. `\newfontface` could be used for this purpose, shown in Example 3.

4.1 More control over font shape selection

```

BoldFont = <font name>
ItalicFont = <font name>
BoldItalicFont = <font name>
SlantedFont = <font name>
BoldSlantedFont = <font name>
SwashFont = <font name>
BoldSwashFont = <font name>
SmallCapsFont = <font name>
UprightFont = <font name>

```

The automatic bold, italic, and bold italic font selections will not be adequate for the needs of every font: while some fonts mayn't even have bold or italic shapes, in which case a skilled (or lucky) designer may be able to choose well-matching accompanying shapes from a different font altogether, others can have a range of bold and italic fonts to choose among. The `BoldFont` and `ItalicFont` features are provided for these situations. If only one of these is used, the bold italic font is requested as the default from the *new* font. See Example 4.

If a bold italic shape is not defined, or you want to specify *both* custom bold and italic shapes, the `BoldItalicFont` feature is provided.

4.1.1 Small caps shapes

For modern OpenType fonts, small caps glyphs are included within a fontface and `fontspec` will automatically detect them for use with the `\textsc` and `\scshape` commands. Pre-OpenType, it was common for font families to be distributed with small caps glyphs in separate fonts, due to the limitations on the number of glyphs allowed in the PostScript Type 1 format. Such fonts may be used by declaring the `SmallCapsFont` for each font of the family you are specifying:

```

\setmainfont{ <upright> }[
  UprightFeatures      = { SmallCapsFont={ <sc> } } ,
  BoldFeatures         = { SmallCapsFont={ <bf sc> } } ,
  ItalicFeatures       = { SmallCapsFont={ <it sc> } } ,

```

Example 3: Defining a single font face.

QED	<pre> \newfontface\qedfont{EBGaramond-Regular.otf}[Style=Swash] \qedfont QED % \emph, \textbf, etc., all don't work </pre>
-----	--

Example 4: Explicit selection of the bold font.

	<code>\setmainfont{Ysabeau-Hairline.otf}%</code>
	<code>[BoldFont={Ysabeau-Thin.otf}]</code>
Hairline	<code>Hairline \\\</code>
Hairline Italic	<code>{\itshape Hairline Italic} \\\</code>
Thin	<code>{\bfseries Thin } \\\</code>
Thin Italic	<code>{\bfseries\itshape Thin Italic} \\\</code>

```

    BoldItalicFeatures = { SmallCapsFont={ <bf it sc> } } ,
]
Roman 123 \\\ \textsc{Small caps 456}

```

For most modern fonts that have small caps as a font feature, this level of control isn't generally necessary.

All of the bold, italic, and small caps fonts can be loaded with different font features from the main font. See [Section 3](#) for details. When an OpenType font is selected for `SmallCapsFont`, the small caps font feature is *not* automatically enabled. In this case, users should write instead, if necessary,

```

\setmainfont{...}[
    SmallCapsFont={...},
    SmallCapsFeatures={Letters=SmallCaps},
]

```

4.1.2 Slanted font shapes

When a font family has both slanted *and* italic shapes, these may be specified separately using the analogous features `SlantedFont` and `BoldSlantedFont`. Without these, however, the $\text{\LaTeX}$ font switches for slanted (`\textsl`, `\slshape`) will default to the italic shape.

4.1.3 Swash font shapes

Swash font shapes in a family is supported by $\text{\LaTeX}$'s commands `\textsw` and `\swshape`. These commands assume that swash shapes are in a sense 'parallel' to italic shapes — for instance, writing both `\swshape` and `\itshape` would not result in an italic swash shape (you would get whichever was declared last). The `fontspec` package adopts this approach, while recognising that OpenType fonts in theory could have any crazy combination of shapes such as 'italic swash small caps'. Attempting to support arbitrarily complex situations makes setup (and the code) more difficult with let's say infrequent benefit — `fontspec`'s alternate feature selection mechanisms (such as `\addfontfeature{Style=Swash}`) can be used in such situations.

Therefore, setup is quite simple:

```

\setmainfont{...}[
    SwashFont = {...} ,
    BoldSwashFont = {...} ,
]

```

No assumptions are made about the +swsh OpenType feature availability, and if desired the ‘Swash’ feature needs to be explicitly requested as in:

```
\setmainfont{...}[
    SwashFont = {...} ,
    SwashFeatures = {Style=Swash} ,
    ...
]
```

This may become more automatic in the future.

4.2 Specifically choosing the NFSS family

In L^AT_EX’s NFSS, font families are defined with names such as ‘ppl’ (Palatino), ‘lmr’ (Latin Modern Roman), and so on, which are selected with the `\fontfamily` command:

```
\fontfamily{ppl}\selectfont
```

In `fontspec`, the family names are auto-generated based on the fontname of the font; for example, writing `\fontspec{Times New Roman}` for the first time would generate an internal font family name of ‘TimesNewRoman(1)’. Please note that you should not rely on the name that is generated.

In certain cases it is desirable to be able to choose this internal font family name so it can be re-used elsewhere for interacting with other packages that use the L^AT_EX’s font selection interface; an example might be

```
\usepackage{fancyvrb}
\fvset{fontfamily=myverbatimfont}
```

To select a font for use in this way in `fontspec` use the NFSSFamily feature:²

```
\newfontfamily\verbatimfont{Inconsolata}[NFSSFamily=myverbatimfont]
```

It is then possible to write commands such as:

```
\fontfamily{myverbatimfont}\selectfont
```

which is essentially the same as writing `\verbatimfont`, or to go back to the original example:

```
\fvset{fontfamily=myverbatimfont}
```

Only use this feature when necessary; the in-built font switching commands that `fontspec` generates (such as `\verbatimfont` in the example above) are recommended in all other cases.

If you don’t wish to explicitly set the NFSS family but you would like to know what it is, an alternative mechanism for package writers is introduced as part of the `fontspec` programming interface; see the function `\fontspec_set_family:Nnn` for details ([Section 5 on page 68](#)).

²Thanks to Luca Fascione for the example and motivation for finally implementing this feature.

4.3 Choosing additional NFSS font faces

L^AT_EX's font selection scheme (NFSS) is more flexible than the fontspec interface discussed up until this point. It assigns to each font face a *family* (discussed above), a *series* such as bold or light or condensed, and a *shape* such as italic or slanted or small caps. The fontspec features such as BoldFont and so on all assign faces for the default series and shapes of the NFSS, but it's not uncommon to have font families that have multiple weights and shapes and so on.

If you set up a regular font family with the 'standard four' (upright, bold, italic, and bold italic) shapes and then want to use, say, a light font for a certain document element, many users will be perfectly happy to use `\newfontface\⟨switch⟩` and use the resulting font `\⟨switch⟩`. In other cases, however, it is more convenient or even necessary to load additional fonts using additional NFSS specifiers.

```
FontFace = {⟨series⟩}{⟨shape⟩} { Font = ⟨font name⟩ , ⟨features⟩ }
FontFace = {⟨series⟩}{⟨shape⟩}{⟨font name⟩}
```

The font thus specified will inherit the font features of the main font, with optional additional `⟨features⟩` as requested. (Note that the optional `{⟨features⟩}` argument is still surrounded with curly braces.) Multiple FontFace commands may be used in a single declaration to specify multiple fonts. As an example:

```
\setmainfont{font1.otf}[
  FontFace = {c}{\shapedefault}{ font2.otf } ,
  FontFace = {c}{m}{ Font = font3.otf , Color = red }
]
```

Writing `\fontseries{c}\selectfont` will result in font2 being selected, which then followed by `\fontshape{m}\selectfont` will result in font3 being selected (in red). A font face that is defined in terms of a different series but an upright shape (`\shapedefault`, as shown above) will attempt to find a matching small caps feature and define that face as well. Conversely, a font face defined in terms of a non-standard font shape will not.

There are some standards for choosing shape and series codes; the L^AT_EX 2_ε font selection guide³ has a comprehensive listing.

The FontFace command also interacts properly with the SizeFeatures command as follows:

```
FontFace = {c}{n}{
  Font = lmsans10-oblique.otf ,
  SizeFeatures = {
    { Size = -10 , Font = lmsans8-oblique.otf } ,
    { Size = 10-15 } ,
    { Size = 15- , Font = lmsans17-oblique.otf } ,
  },
},
```

Note that if the first Font feature is omitted then each size needs its own inner Font declaration.

³`texdoc fntguide`

4.3.1 An example for `\strong`

If you wanted to set up a font family to allow nesting of the `\strong` to easily access increasing font weights, you might use a declaration along the following lines:

```
\setmonofont{SourceCodePro}[  
  Extension = .otf ,  
  UprightFont = *-Light ,  
  BoldFont = *-Regular ,  
  FontFace = {k}{n}{*-Black} ,  
]  
\strongfontdeclare{\bfseries,\fontseries{k}\selectfont}
```

Further ‘syntactic sugar’ is planned to make this process somewhat easier.

4.4 Math(s) fonts

When `\setmainfont`, `\setsansfont` and `\setmonofont` are used in the preamble, they also define the fonts to be used in maths mode inside the `\mathrm`-type commands. This only occurs in the preamble because $\LaTeX$ freezes the maths fonts after this stage of the processing. The `fontspec` package must also be loaded after any maths font packages (e.g., `euler`) to be successful. (Actually, it is *only* `euler` that is the problem.⁴)

Note that `fontspec` will not change the font for general mathematics; only the upright and bold shapes will be affected. To change the font used for the mathematical symbols, see either the `mathspec` package or the `unicode-math` package.

Note that you may find that loading some maths packages won’t be as smooth as you expect since `fontspec` (and $\XeTeX$ in general) breaks many of the assumptions of $\TeX$ as to where maths characters and accents can be found. Contact me if you have troubles, but I can’t guarantee to be able to fix any incompatibilities. The Lucida and Euler maths fonts should be fine; for all others keep an eye out for problems.

```
\setmathrm{<font name>}[<font features>]  
\setmathsf{<font name>}[<font features>]  
\setmathtt{<font name>}[<font features>]  
\setboldmathrm{<font name>}[<font features>]
```

However, the default text fonts may not necessarily be the ones you wish to use when typesetting maths (especially with the use of fancy ligatures and so on). For this reason, you may optionally use the commands above (in the same way as our other `\fontspec`-like commands) to explicitly state which fonts to use inside such commands as `\mathrm`. Additionally, the `\setboldmathrm` command allows you define the font used for `\mathrm` when in bold maths mode (which is activated with, among others, `\boldmath`).

For example, if you were using Optima with the Euler maths font, you might have this in your preamble:

```
\usepackage{mathpazo}  
\usepackage{fontspec}  
\setmainfont{Optima}
```

⁴Speaking of `euler`, if you want to use its `[mathbf]` option, it won’t work, and you’ll need to put this after `fontspec` is loaded instead: `\AtBeginDocument{\DeclareMathAlphabet\mathbf{U}{eur}{b}{n}}`

```
\setmathrm{Optima}
\setboldmathrm[BoldFont={Optima ExtraBlack}]{Optima Bold}
```

These commands are compatible with the `unicode-math` package. Having said that, `unicode-math` also defines a more general way of defining fonts to use in maths mode, so you can ignore this subsection if you’re already using that package.

5 Miscellaneous font selecting details

The optional argument — from v2.4 For the first decade of `fontspec`’s life, optional font features were selected with a bracketed argument before the font name, as in:

```
\setmainfont[
  lots and lots ,
  and more and more ,
  an excessive number really ,
  of font features could go here
]{myfont.otf}
```

This always looked like ugly syntax to me, because the most important detail — the name of the font — was tucked away at the end. The order of these arguments has now been reversed:

```
\setmainfont{myfont.otf}[
  lots and lots ,
  and more and more ,
  an excessive number really ,
  of font features could go here
]
```

I hope this doesn’t cause any problems.

1. Backwards compatibility has been preserved, so either input method works.
2. In fact, you can write

```
\fontspec[Ligatures=Rare]{myfont.otf}[Color=red]
```

if you really felt like it and both sets of features would be applied.

Spaces `\fontspec` and `\addfontfeatures` ignore trailing spaces as if it were a ‘naked’ control sequence; e.g., ‘M. `\fontspec{...}` N’ and ‘M. `\fontspec{...}`N’ are the same.

Part III

Selecting font features

The commands discussed so far such as `\fontspec` each take an optional argument for accessing the font features of the requested font. Commands are provided to set default features to be applied for all fonts, and even to change the features that a font is presently loaded with. Different font shapes can be loaded with separate features, and different features can even be selected for different sizes that the font appears in. This part discusses these options.

1 Default settings

```
\defaultfontfeatures{<font features>}
```

It is sometimes useful to define font features that are applied to every subsequent font selection command. This may be defined with the `\defaultfontfeatures` command, shown in Example 5. New calls of `\defaultfontfeatures` overwrite previous ones, and defaults can be reset by calling the command with an empty argument.

```
\defaultfontfeatures[<font name>]{<font features>}
```

Default font features can be specified on a per-font and per-face basis by using the optional argument to `\defaultfontfeatures` as shown.

```
\defaultfontfeatures[tegyreadventor-regular.otf]{Color=blue}  
\setmainfont{tegyreadventor-regular.otf}% will be blue
```

Multiple fonts may be affected by using a comma separated list of font names.

```
\defaultfontfeatures[<\font-switch>]{<font features>}
```

New in v2.4. Defaults can also be applied to symbolic families such as those created with the `\newfontfamily` command and for `\rmfamily`, `\sffamily`, and `\ttfamily`:

```
\defaultfontfeatures[\rmfamily,\sffamily]{Ligatures=TeX}  
\setmainfont{tegyreadventor-regular.otf}% will use standard TeX ligatures
```

Example 5: A demonstration of the `\defaultfontfeatures` command.

	<pre>\fontspec{tegyreadventor-regular.otf} Some default text 0123456789 \\ \defaultfontfeatures{ Numbers=OldStyle, Color=888888 } \fontspec{tegyreadventor-regular.otf}</pre>
Some default text 0123456789	Now grey, with old-style figures: 0123456789
Now grey, with old-style figures: 0123456789	0123456789

The line above to set T_EX-like ligatures is now activated by *default* in `fontspec.cfg`. To reset default font features, simply call the command with an empty argument:

```
\defaultfontfeatures[\rmfamily,\sffamily]{}
\setmainfont{texgyreadventor-regular.otf}% will no longer use standard TeX ligatures
```

```
\defaultfontfeatures+{<font features>}
\defaultfontfeatures+[<font name>]{<font features>}
```

New in v2.4. Using the + form of the command appends the `<font features>` to any already-selected defaults.

2 Working with the currently selected features

```
\IfFontFeatureActiveTF{<font feature>}{<>true code>}{<>false code>}
```

This command queries the currently selected font face and executes the appropriate branch based on whether the `<font feature>` as specified by `fontspec` is currently active.

For example, the following will print ‘True’:

```
\setmainfont{texgyreagella-regular.otf}[Numbers=OldStyle]
\IfFontFeatureActiveTF{Numbers=OldStyle}{True}{False}
```

Note that there is no way for `fontspec` to know what the default features of a font will be. For example, by default the `texgyreagella` fonts use lining numbers. But in the following example, querying for lining numbers returns false since they have not been explicitly requested:

```
\setmainfont{texgyreagella-regular.otf}
\IfFontFeatureActiveTF{Numbers=Lining}{True}{False}
```

Please note: At time of writing this function only supports OpenType fonts; AAT/Graphite fonts under the X_YT_EX engine are not supported.

```
\addfontfeatures{<font features>}
```

This command allows font features in an entire font family to be changed without knowing what features are currently selected or even what font family is being used. A good example of this could be to add a hook to all tabular material to use monospaced numbers, as shown in Example 6. If you attempt to *change* an already-selected feature, `fontspec` will try to de-activate any features that clash with the new ones. *E.g.*, the following two invocations are mutually exclusive:

```
\addfontfeatures{Numbers=OldStyle}...
\addfontfeatures{Numbers=Lining}...
123
```

Since `Numbers=Lining` comes last, it takes precedence and deactivates the call `Numbers=OldStyle`.

If you wish to apply the change to only one of the fonts of a family (say, italics only) you can write

```
\addfontfeatures{ItalicFeatures={Numbers=Lowercase}}
```

`\addfontfeature` This command may also be executed under the alias `\addfontfeature`.

Example 6: A demonstration of the `\addfontfeatures` command.

<pre> \fontspec{texgyreadventor-regular.otf}% [Numbers={Proportional,OldStyle}] `In 1842, 999 people sailed 97 miles in 13 boats. In 1923, 111 people sailed 54 miles in 56 boats.' \bigskip `In 1842, 999 people sailed 97 miles in 13 boats. In 1923, 111 people sailed 54 miles in 56 boats.' </pre>	<pre> {\addfontfeatures{Numbers={Monospaced,Lining}} \begin{tabular}{@{} cccc @{}} Year & People & Miles & Boats & \\ \hline 1842 & 999 & 75 & 13 & \\ 1923 & 111 & 54 & 56 & \\ \end{tabular}} </pre>
---	--

2.1 Priority of feature selection

Features defined with `\addfontfeatures` override features specified by `\fontspec`, which in turn override features specified by `\defaultfontfeatures`. If in doubt, whenever a new font is chosen for the first time, an entry is made in the transcript (`.log`) file displaying the font name and the features requested.

3 Different features for different font shapes

```

BoldFeatures={\features}
ItalicFeatures={\features}
BoldItalicFeatures={\features}
SlantedFeatures={\features}
BoldSlantedFeatures={\features}
SwashFeatures={\features}
BoldSwashFeatures={\features}
SmallCapsFeatures={\features}
UprightFeatures={\features}

```

It is entirely possible that separate fonts in a family will require separate options; *e.g.*, certain italic fonts contains various swash feature options that are usually unavailable in the upright (‘roman’) shapes.

The font features defined at the top level of the optional `\fontspec` argument are applied to *all* shapes of the family. Using the `xxFeatures` options shown above, separate font features may be defined to their respective shapes *in addition* to, and with precedence over, the ‘global’ font features. See Example 7.

Note that because most fonts include their small caps glyphs within the main font, features specified with `SmallCapsFeatures` are applied *in addition* to any other shape-specific features as defined above, and hence `SmallCapsFeatures` can be nested within `ItalicFeatures` and friends. Every combination of upright, italic, bold, (etc.), and small

Example 7: Features for, say, just italics.

	<code>\fontspec{EBGaramond-Regular.otf}%</code>
	<code>[ItalicFont=EBGaramond-Italic.otf]</code>
<i>Don't Ask Victoria!</i>	<code>\itshape Don't Ask Victoria! \</code>
<i>Don't Ask Victoria!</i>	<code>\addfontfeature{ItalicFeatures={Style=Swash}}</code>
	<code>Don't Ask Victoria! \</code>

caps can thus be assigned individual features, as shown in the somewhat ludicrous Example 8.

4 Selecting fonts from TrueType Collections (TTC files)

TrueType Collections are multiple fonts contained within a single file. Each font within a collection must be explicitly chosen using the `FontIndex` command. Since TrueType Collections are often used to contain the italic/bold shapes in a family, `fontspec` automatically selects the italic, bold, and bold italic fontfaces from the same file. For example, to load the macOS system font Optima:

```
\setmainfont{Optima.ttc}[
  Path = /System/Library/Fonts/ ,
  UprightFeatures = {FontIndex=0} ,
  BoldFeatures = {FontIndex=1} ,
  ItalicFeatures = {FontIndex=2} ,
  BoldItalicFeatures = {FontIndex=3} ,
]
```

Support for TrueType Collections has only been tested in X_YTeX, but should also work with an up-to-date version of LuaTeX and the `luaotfload` package.

5 Different features for different font sizes

```
SizeFeatures = {
  ...
  { Size = <size range>, <font features> },
  { Size = <size range>, Font = <font name>, <font features> },
  ...
}
```

The `SizeFeatures` feature is a little more complicated than the previous features discussed. It allows different fonts and different font features to be selected for a given font family as the point size varies.

It takes a comma separated list of braced, comma separated lists of features for each size range. Each sub-list must contain the `Size` option to declare the size range, and optionally `Font` to change the font based on size. Other (regular) `fontspec` features that are

Example 8: An example of setting the `SmallCapsFeatures` separately for each font shape.

	<pre> \fontspec{texgyretermes}[Extension = {.otf}, UprightFont = {*-regular}, ItalicFont = {*-italic}, BoldFont = {*-bold}, BoldItalicFont = {*-bolditalic}, UprightFeatures={Color = 220022, SmallCapsFeatures = {Color=115511}}, ItalicFeatures={Color = 2244FF, SmallCapsFeatures = {Color=112299}}, BoldFeatures={Color = FF4422, SmallCapsFeatures = {Color=992211}}, BoldItalicFeatures={Color = 888844, SmallCapsFeatures = {Color=444422}},]</pre>
Upright SMALL CAPS	<code>\Upright {\scshape Small Caps}\</code>
<i>Italic</i> <i>ITALIC SMALL CAPS</i>	<code>\itshape Italic {\scshape Italic Small Caps}\</code>
Bold BOLD SMALL CAPS	<code>\upshape\bfseries Bold {\scshape Bold Small Caps}\</code>
<i>Bold Italic</i> <i>BOLD ITALIC SMALL CAPS</i>	<code>\itshape Bold Italic {\scshape Bold Italic Small Caps}</code>

added are used on top of the font features that would be used anyway. A demonstration to clarify these details is shown in Example 9. A less trivial example is shown in the context of optical font sizes in Section 6.6 on page 31.

To be precise, the `Size` sub-feature accepts arguments in the form shown in Table 1 on the following page. Braces around the size range are optional. For an exact font size (`Size=X`) font sizes chosen near that size will ‘snap’. For example, for size definitions at exactly 11pt and 14pt, if a 12pt font is requested *actually* the 11pt font will be selected. This is a remnant of the past when fonts were designed in metal (at obviously rigid sizes) and later when bitmap fonts were similarly designed for fixed sizes.

If additional features are only required for a single size, the other sizes must still be specified. As in:

```

SizeFeatures={
  {Size=-10,Numbers=Uppercase},
  {Size=10-}}

```

Example 9: An example of specifying different font features for different sizes of font with `SizeFeatures`.

	<pre> \fontspec{texgyrechorus-mediumitalic.otf}[SizeFeatures={ {Size={-8}, Font=texgyrebonum-italic.otf, Color=AA0000}, {Size={8-14}, Color=00AA00}, {Size={14-}, Color=0000AA}}] </pre>
<i>Small</i>	<code>{\scriptsize Small\par}</code>
<i>Normal size</i>	<code>Normal size\par</code>
<i>Large</i>	<code>{\Large Large\par}</code>

Otherwise, the font sizes greater than 10 won't be defined at all!

Interaction with other features For `SizeFeatures` to work with `ItalicFeatures`, `BoldFeatures`, etc., and `SmallCapsFeatures`, a strict hierarchy is required:

```
UprightFeatures =
{
  SizeFeatures =
  {
    {
      Size = -10,
      Font = ..., % if necessary
      SmallCapsFeatures = {...},
      ... % other features for this size range
    },
    ... % other size ranges
  }
}
```

Suggestions on simplifying this interface welcome.

6 Font independent options

Features introduced in this section may be used with any font.

6.1 Colour

`Color` (or `Colour`) uses font specifications to set the colour of the text. You should think of this as the literal glyphs of the font being coloured in a certain way. Notably, this mechanism is different to that of the `color`/`xcolor`/`hyperref`/etc. packages, and in fact using `fontspec` commands to set colour will prevent your text from changing colour using those packages at all! (For example, if you set the colour in a `\setmainfont` command, `\color{...}` and related commands, including hyperlink colouring, will no longer have any effect on text in this font.) Therefore, `fontspec`'s colour commands are best used to set explicit colours in specific situations, and the `xcolor` package is recommended for more general colour functionality.

Table 1: Syntax for specifying the size to apply custom font features.

Input	Font size, s
<code>Size = X-</code>	$s \geq X$
<code>Size = -Y</code>	$s < Y$
<code>Size = X-Y</code>	$X \leq s < Y$
<code>Size = X</code>	$s = X$

Example 10: Selecting colour with transparency.



```
\fontsize{48}{48}  
\fontspec{texgyrebonum-bold.otf}  
{\addfontfeature{Color=FF000099}W}\kern-0.4ex  
{\addfontfeature{Color=0000FF99}S}\kern-0.4ex  
{\addfontfeature{Color=DDBB2299}P}\kern-0.5ex  
{\addfontfeature{Color=00BB3399}R}
```

The colour can be defined as a triplet of two-digit Hex RGB values, with optionally another value for the transparency (where 00 is completely transparent and FF is opaque.)

If you load the `xcolor` package, you may use any named colour instead of writing the colours in hexadecimal.

```
\usepackage{xcolor}  
...  
\fontspec[Color=red]{Montserrat-Medium.otf} ...  
\definecolor{Foo}{rgb}{0.3,0.4,0.5}  
\fontspec[Color=Foo]{Montserrat-Medium.otf} ...
```

You may also use named colours defined with the `color` commands of the L₃ programming layer:

```
\ExplSyntaxOn  
\color_set:nnn{Foo}{rgb}{0.3,0.4,0.5}  
\ExplSyntaxOff  
...  
\fontspec[Color=Foo]{Montserrat-Medium.otf} ...
```

Color expressions (like `red!50!blue`) are not supported. The `color` package is *not* supported neither.

The code will at first test for color names of the L₃ layer, then for `xcolor` names, and at last try to use the argument as a hexadecimal value.

You may specify the transparency with a named colour using the `Opacity` feature, which takes an decimal from zero to one corresponding to transparent to opaque respectively:

```
\fontspec[Color=red,Opacity=0.7]{Montserrat-Medium.otf} ...
```

It is still possible to specify a colour in six-char hexadecimal form while defining opacity in this way, if you like.

6.1.1 Color models

With $\text{\XeTeX}$ color are always written in the `rgb` color model into the PDF. When using $\text{\LuaTeX}$, colors with the commands of the L₃ layer can be written as `rgb` or `cmyk` or as spot color depending on their definition and of the value of the variable `\l_color_fixed_model_tl`.

6.1.2 Spot colors

With Lua_T_EX it is possible to use spot colors. This requires the use of the PDF management:

```
\DocumentMetadata{}
\documentclass{article}
\usepackage{fontspec}
\ExplSyntaxOn
\color_model_new:nnn { sepblue } { Separation }
{
  name = PANTONE~3005~U ,
  alternative-model = cmyk ,
  alternative-values = {1, 0.56, 0,0},
}
\color_set:nnn{spotblue}{sepblue}{1}
\ExplSyntaxOff
...
\fontspec[Color=spotblue]{texgyreheros}
```

6.2 Scale

```
Scale = <number>
Scale = MatchLowercase
Scale = MatchUppercase
Scale = MatchAveragcase
```

In its explicit form, `Scale` takes a single numeric argument for linearly scaling the font, as demonstrated in Example 1.

As well as a numerical argument, the `Scale` feature also accepts options `MatchLowercase`, `MatchUppercase`, and `MatchAveragcase`, which will scale the font being selected to match the current default roman font to either the height of the lowercase, the height of the uppercase letters, or the average of the two, respectively; these features are shown in Example 11. The amount of scaling used in each instance is reported in the `.log` file.

Additional calls to the `Scale` feature overwrite the settings of the former. If you want to accumulate scale factors (useful perhaps to fine-tune the settings of `MatchLowercase`), the `ScaleAgain` feature can be used as many times as necessary. For example:

Example 11: Automatically calculated scale values.

The perfect match is hard to find. LOGO FONT Lower and UPPER CASE	<pre>\setmainfont{texgyrepagella-regular.otf} \newfontfamily\lc[Scale=MatchLowercase]{texgyreadventor-regular.otf} The perfect match {\lc is hard to find.}\ \newfontfamily\uc[Scale=MatchUppercase]{texgyreheros-regular.otf} L O G O {\uc F O N T}\ \newfontfamily\ac[Scale=MatchAveragcase]{FiraMath-Regular.otf} Lower {\ac and UPPER} CASE</pre>
---	---

```
[ Scale = 1.1 , Scale = 1.2 ]      % -> scale of 1.2
[ Scale = 1.1 , ScaleAgain = 1.2 ] % -> scale of 1.32
```

Note that when `Scale=MatchLowercase`, `Scale=MatchUppercase`, or `Scale=MatchAverageCase` is used with `\setmainfont`, the new ‘main’ font of the document will be scaled to match the old default. If you wish to automatically scale all fonts except have the main font use ‘natural’ scaling, you may write

```
\defaultfontfeatures{ Scale = MatchLowercase }
\defaultfontfeatures[\rmfamily]{ Scale = 1}
```

One or both of these lines may be placed into a local `fontspec.cfg` file (see [Section 3.3 on page 6](#)) for this behaviour to be effected in your own documents automatically. (Also see [Section 1 on page 21](#) for more information on setting font defaults.)

6.3 Interword space

While the space between words can be varied with the $\TeX$ primitive `\spaceskip` command, `fontspec` also supports changing the interword spacing when a given font is loaded.

The space in between words in a paragraph will be chosen automatically, and generally will not need to be adjusted. For those times when the precise details are important, the `WordSpace` feature is provided, which takes either a single scaling factor to scale the default value, or a triplet of comma-separated values to scale the nominal value, the stretch, and the shrink of the interword space by, respectively. (`WordSpace={x}` is the same as `WordSpace={x,x,x}`.)

Note that $\TeX$ ’s optimisations in how it loads fonts means that you cannot use this feature in `\addfontfeatures`.

6.4 Post-punctuation space

If `\frenchspacing` is *not* in effect (which is the default), $\TeX$ will allow extra space after some punctuation in its goal of justifying the lines of text.

The `PunctuationSpace` feature takes a scaling factor by which to adjust the nominal value chosen for the font; this is demonstrated in [Example 13](#). Note that `PunctuationSpace=0` is *not* equivalent to `\frenchspacing`, although the difference will only be apparent when a line of text is under-full.

Note that $\TeX$ ’s optimisations in how it loads fonts means that you cannot use this feature in `\addfontfeatures`.

6.5 The hyphenation character

The letter used for hyphenation may be chosen with the `HyphenChar` feature. With one exception (`HyphenChar = None`), this is a $\text{\XeTeX}$ -only feature since $\text{\LuaTeX}$ cannot set the hyphenation character on a per-font basis; see its `\prehyphenchar` primitive for further details.

`HyphenChar` takes three types of input, which are chosen according to some simple rules. If the input is the string `None`, then hyphenation is suppressed for this font.

Example 12: Scaling the default interword space. An exaggerated value has been chosen to emphasise the effects here.

	<pre> \fontspec{texgyretermes-regular.otf} Some text for our example to take up some space, and to demonstrate the default interword space. \bigskip </pre>
Some text for our example to take up some space, and to demonstrate the default interword space.	<pre> \fontspec{texgyretermes-regular.otf}% [WordSpace = 0.3] Some text for our example to take up some space, and to demonstrate the default interword space. </pre>
Some text for our example to take up some space, and to demonstrate the default interword space.	

Example 13: Scaling the default post-punctuation space.

	<pre> \nonfrenchspacing \fontspec{texgyreschola-regular.otf} Letters, Words. Sentences. \par \fontspec{texgyreschola-regular.otf}[PunctuationSpace=2] Letters, Words. Sentences. \par \fontspec{texgyreschola-regular.otf}[PunctuationSpace=0] Letters, Words. Sentences. </pre>
Letters, Words. Sentences.	
Letters, Words. Sentences.	
Letters, Words. Sentences.	

As part of `fontspec.cfg`, the default monospaced family (e.g., `\ttfamily`) is set up to automatically set `HyphenChar = None`.

If the input is a single character, then this character is used. Finally, if the input is longer than a single character it must be the UTF-8 slot number of the hyphen character you desire.

Note that $\text{\TeX}$'s optimisations in how it loads fonts means that you cannot use this feature in `\addfontfeatures`.

6.6 Optical font sizes

Optically scaled fonts thicken out as the font size decreases in order to make the glyph shapes more robust (less prone to losing detail), which improves legibility. Conversely, at large optical sizes the serifs and other small details may be more delicately rendered.

OpenType fonts with optical scaling can exist in several discrete sizes (in separate font files). When loading fonts by name, $\text{\XeTeX}$ and $\text{\LuaTeX}$ engines will attempt to *automatically* load the appropriate font as determined by the current font size. An example of this behaviour is shown in Example 15, in which some larger text is mechanically scaled down to compare the difference for equivalent font sizes.

The `OpticalSize` feature may be used to specify a different optical size. With `OpticalSize` set (Example 16) to zero, no optical size font substitution is performed.

The `SizeFeatures` feature (Section 5 on page 24) can be used to specify exactly which optical sizes will be used for ranges of font size. For example, something like:

```
\fontspec{Latin Modern Roman}[
  UprightFeatures = { SizeFeatures = {
    {Size=-10,      OpticalSize=8 },
    {Size= 10-14,   OpticalSize=10},
    {Size= 14-18,   OpticalSize=14},
    {Size= 18-,     OpticalSize=18}}}
]
```

6.7 Font transformations

In rare situations users may want to mechanically distort the shapes of the glyphs in the current font such as shown in Example 17. Please don't overuse these features; they are *not* a good alternative to having the real shapes.

Example 14: Explicitly choosing the hyphenation character.

EXAMPLE HYPHENATION

```
\def\text{\fbox{\parbox{1.55cm}{%
  EXAMPLE HYPHENATION%
}}\qqquad\qqquad\null\par\bigskip}
```

EXAMPLE HYPHEN+ ATION

```
\fontspec{LinLibertine_R.otf}[HyphenChar=None]
\text
\fontspec{LinLibertine_R.otf}[HyphenChar={+}]
\text
```

Example 15: A demonstration of automatic optical size selection.		
	<code>\fontspec{Latin Modern Roman}</code>	
Automatic optical size	Automatic optical size	<code>\\</code>
Automatic optical size	<code>\scalebox{0.4}{\Huge</code>	
	Automatic optical size}	

Example 16: Explicit optical size substitution for the Latin Modern Roman family.		
	<code>\fontspec{Latin Modern Roman}[OpticalSize=5]</code>	
	Latin Modern optical sizes	<code>\\</code>
	<code>\fontspec{Latin Modern Roman}[OpticalSize=8]</code>	
Latin Modern optical sizes	Latin Modern optical sizes	<code>\\</code>
Latin Modern optical sizes	<code>\fontspec{Latin Modern Roman}[OpticalSize=12]</code>	
Latin Modern optical sizes	Latin Modern optical sizes	<code>\\</code>
Latin Modern optical sizes	<code>\fontspec{Latin Modern Roman}[OpticalSize=17]</code>	
	Latin Modern optical sizes	

Example 17: Artificial font transformations.		
	<code>\fontspec{Quattrocento-Regular.ttf} \emph{ABCxyz} \quad</code>	
	<code>\fontspec{Quattrocento-Regular.ttf}[FakeSlant=0.2] ABCxyz</code>	
	<code>\fontspec{Quattrocento-Regular.ttf} ABCxyz \quad</code>	
	<code>\fontspec{Quattrocento-Regular.ttf}[FakeStretch=1.2] ABCxyz</code>	
ABCxyz	ABCxyz	
ABCxyz	ABCxyz	<code>\fontspec{Quattrocento-Regular.ttf} \textbf{ABCxyz} \quad</code>
ABCxyz	ABCxyz	<code>\fontspec{Quattrocento-Regular.ttf}[FakeBold=1.5] ABCxyz</code>

If values are omitted, their defaults are as shown above.

FakeBold fakes a bold font by stroking the glyph outlines with an outline of a constant width. This is typographically not ideal, but can give acceptable output when the factor is small and is rendered correctly by the PDF viewer. However, many PDF viewers—most notably Chrome—completely misrender the effect at low zoom levels, so you should only use this feature on fonts typeset at large sizes, if you know for certain which PDF viewers will be used to view the document, or if you’re printing to paper.

Internally, the value of the FakeBold feature is divided by 10, multiplied by the font size in T_EX points (pt), and then directly written to the PDF file as the stroke width (therefore treating T_EX points as PostScript points (bp)). As an additional feature not present in X_YT_EX, if you set the FakeBold parameter to exactly 0, LuaT_EX will stroke the font without setting the stroke width, meaning that the stroke width will be inherited from the outer PDF graphics state.

Setting FakeBold to values below 0.5 is only useful for print documents, since many PDF viewers will always render the stroke widths at some greater minimum value; setting FakeBold to values above 20 is rarely useful since if the stroke overlaps with itself, there will be significant visual artifacts. Useful values are generally between 1 and 10.

If you want the bold shape to be faked automatically, or the italic shape to be slanted automatically, use the AutoFakeBold and AutoFakeSlant features. For example, the following two invocations are equivalent:

```
\fontspec[AutoFakeBold=1.5]{Charis SIL}  
\fontspec[BoldFeatures={FakeBold=1.5}]{Charis SIL}
```

If both of the AutoFake... features are used, then the bold italic font will also be faked.

6.8 Letter spacing

Letter spacing, or tracking, is the term given to adding (or subtracting) a small amount of horizontal space in between adjacent characters. It is specified with the LetterSpace, which takes a numeric argument, shown in Example 18.

The letter spacing parameter is a normalised additive factor (not a scaling factor); it is defined as a percentage of the font size. That is, for a 10 pt font, a letter spacing parameter of ‘1.0’ will add 0.1 pt between each letter.

This functionality is not generally used for lowercase text in modern typesetting but does have historic precedent in a variety of situations. In particular, small amounts of letter spacing can be very useful, when setting small caps or all caps titles. Also see the OpenType Uppercase option of the Letters feature (3.1.7 on page 45).

Example 18: The LetterSpace feature.

	<code>\setmainfont{Ysabeau-Light.otf}</code>
	<code>\addfontfeature{LetterSpace=0.0}</code>
	<code>USE TRACKING FOR DISPLAY CAPS TEXT \</code>
<code>USE TRACKING FOR DISPLAY CAPS TEXT</code>	<code>\addfontfeature{LetterSpace=3.0}</code>
<code>USE TRACKING FOR DISPLAY CAPS TEXT</code>	<code>USE TRACKING FOR DISPLAY CAPS TEXT</code>

7 Variable fonts

OpenType variable fonts and Multiple Master fonts are parameterised over orthogonal font axes, allowing continuous selection along such features as weight, width, and optical size.

Currently OpenType variable fonts are only supported in LuaTeX, while Multiple Master fonts only work with XeTeX.

7.1 Optical font sizes

Whereas traditional OpenType fonts will have only a few separate optical sizes, a Variable or Multiple Master font's optical size can be specified over a continuous range. Unfortunately, this flexibility makes it harder to create an automatic interface through L^AT_EX, and the optical size for a Variable or Multiple Master font must always be specified explicitly.

```
\fontspec{Minion MM Roman}[OpticalSize=11]
MM optical size test                \\
\fontspec{Minion MM Roman}[OpticalSize=47]
MM optical size test                \\
\fontspec{Minion MM Roman}[OpticalSize=71]
MM optical size test                \\
```

7.2 Weight

For fonts with a variable weight axis, the weight can be specified through the `Weight` feature. The value should be between 0 and 1000, where typically 400 corresponds to regular weight and 700 is a bold font.

```
\fontspec{Source Serif Variable}[Weight=700]
Bold                               \\
\fontspec{Source Serif Variable}[Weight=200]
Extra Light                        \\
```

7.3 Width

Similarly, the `Width` feature allows specifying the value of the width axis, where the value is a percentage of normal width.

```
\fontspec{Noto Serif}[Width=100]
Normal Width                       \\
\fontspec{Noto Serif}[Width=75]
Condensed                          \\
```

7.4 Slant

Also fonts with a slant axis can be controlled with the `Slant` feature. In a standard compliant font the value should specify the clockwise angle in degree the glyphs are slanted. Therefore for a typical forward leaning slanted font, a negative value should be passed.

Many fonts use this feature in non-standard ways, so you might have to experiment a bit with the value.

```

\fontspec{Roboto Flex}[Slant=0]
Upright                               \\\
\fontspec{Roboto Flex}[Slant=-5]
Slanted                               \\\

```

7.5 Other axes

For OpenType variable fonts, additional axis values can be specified if the four letter tag of these axes is known. Then their value can be set with the `RawAxis` feature:

```

\fontspec{Noto Serif}[RawAxis={CTGR=100}]
Maximal contrast                               \\\
\fontspec{Noto Serif}[RawAxis={CTGR=0}]
Regular contrast                               \\\

```

7.6 Instances

Instead of manually setting axis values, many fonts contain named instances which are predefined settings of all axes.

To select such an instance, the `Instance` feature can be used:

```

\fontspec{Noto Serif}[Instance=ExtraCondensed Bold]
This is in extra condensed bold.

```

Part IV

OpenType

1 Introduction

OpenType fonts (and other ‘smart’ font technologies such as AAT and Graphite) can change the appearance of text in many different ways. These changes are referred to as font features. When the user applies a feature — for example, small capitals — to a run of text, the code inside the font makes appropriate substitutions and small capitals appear in place of lowercase letters. However, the use of such features does not affect the underlying text. In our small caps example, the lowercase letters are still stored in the document; only the appearance has been changed by the OpenType feature. This makes it possible to search and copy text without difficulty. If the user selected a different font that does not support small caps, the ‘plain’ lowercase letters would appear instead.

Some OpenType features are required to support particular scripts, and these features are often applied automatically. The Indic scripts, for example, often require that characters be reshaped and reordered after they are typed by the user, in order to display them in the traditional ways that readers expect. Other features can be applied to support a particular language. The Junicode font for medievalists uses by default the Old English shape of the letter thorn, while in modern Icelandic thorn has a more rounded shape. If a user tags some text as being in Icelandic, Junicode will automatically change to the Icelandic shape through an OpenType feature that localises the shapes of letters.

There are a large group of OpenType features, designed to support high quality typography a multitude of languages and writing scripts. Examples of some font features have already been shown in previous sections; the complete set of OpenType font features supported by fontspec is described below in [Section 3](#).

The OpenType specification provides four-letter codes (e.g., `smcp` for small capitals) for each feature. The four-letter codes are given below along with the fontspec names for various features, for the benefit of people who are already familiar with OpenType. You can ignore the codes if they don’t mean anything to you.

1.1 How to select font features

Font features are selected by a series of $\langle feature \rangle = \langle option \rangle$ selections. Features are (usually) grouped logically; for example, all font features relating to ligatures are accessed by writing `Ligatures={...}` with the appropriate argument(s), which could be `TeX`, `Rare`, etc., as shown below in [3.1.8](#).

Multiple options may be given to any feature that accepts non-numerical input, although doing so will not always work. Some options will override others in generally obvious ways; `Numbers={OldStyle,Lining}` doesn’t make much sense because the two options are mutually exclusive, and `XYTeX` will simply use the last option that is specified (in this case using `Lining` over `OldStyle`).

If a feature or an option is requested that the font does not have, a warning is given in the console output. As mentioned in [Section 3.4 on page 7](#) these warnings can be suppressed by selecting the `[quiet]` package option.

1.2 How do I know what font features are supported by my fonts?

Although I’ve long desired to have a feature within `fontspec` to display the OpenType features within a font, it’s never been high on my priority list. One reason for that is the existence of the document `opentype-info.tex`, which is available on CTAN or typing `kpsewhich opentype-info.tex` in a Terminal window. Make a copy of this file and place it somewhere convenient. Then open it in your regular T_EX editor and change the font name to the font you’d like to query; after running through plain X_YT_EX, the output PDF will look something like this:

OpenType Layout features found in ‘[Asana-Math.otf]’

```
script = 'DFLT'
  language = <default>
    features = 'onum' 'salt' 'kern'
script = 'cher'
  language = <default>
    features = 'onum' 'salt' 'kern'
script = 'grek'
  language = <default>
    features = 'onum' 'salt' 'ssty' 'kern'
script = 'latn'
  language = <default>
    features = 'dtls' 'onum' 'salt' 'ssty' 'kern'
script = 'math'
  language = <default>
    features = 'dtls' 'onum' 'salt' 'ssty' 'kern'
```

I intentionally picked a font above that by design contains few font features; ‘regular’ text fonts such as Latin Modern Roman contain many more, and I didn’t want to clutter up the document too much. After finding the scripts, languages, and features contained within the font, you’ll then need to cross-check the OpenType tags with the ‘logical’ names used by `fontspec`.

otfinfo Alternatively, and more simply, you can use the command line tool `otfinfo`, which is distributed with T_EXLive. Simply type in a Terminal window, say:

```
otfinfo -f `kpsewhich lmromandunh10-oblique.otf`
```

which results in:

aalt	Access All Alternates
csp	Capital Spacing
dlig	Discretionary Ligatures
frac	Fractions
kern	Kerning
liga	Standard Ligatures
lnum	Lining Figures

onum	Oldstyle Figures
pnum	Proportional Figures
size	Optical Size
tnum	Tabular Figures
zero	Slashed Zero

2 OpenType scripts and languages

Fonts that include glyphs for various scripts and languages may contain different font features for the different character sets and languages they support, and different font features may behave differently depending on the script or language chosen. When multilingual fonts are used, it is important to select which language they are being used for, and more importantly what script is being used.

The ‘script’ refers to the alphabet in use; for example, both English and French use the Latin script. Similarly, the Arabic script can be used to write in both the Arabic and Persian languages.

The Script and Language features are used to designate this information. The possible options are tabulated in [Table 2 on the following page](#) and [Table 3 on page 40](#), respectively. When a script or language is requested that is not supported by the current font, a warning is printed in the console output. See [Section 2 on page 67](#) for methods to create new Script or Language options if required.

Because these font features can change which features are able to be selected for the font, the Script and Language settings are automatically selected by `fontspec` before all others, and, if $\text{X}\text{\TeX}$ is being used, will specifically select the OpenType renderer for this font, as described in [Section 1.2 on page 61](#).

OpenType fonts can make available different font features depending on the Script and Language chosen. In addition, these settings can also set up their own font behaviour and glyph selection (one example is differences in style between some of the letters in the alphabet used for Bulgarian, Serbian, and Russian). The `fontspec` feature `LocalForms = Off` will disable some of these substitutions if desired for some reason. It is important to note that `LocalForms = On` is a default not of `fontspec` but of the underlying font shaping engines in both $\text{X}\text{\TeX}$ and $\text{Lua}\text{\TeX}/\text{otfload}$.

2.1 Script and Language examples

In the examples shown in [Example 19](#), the Code2000 font⁵ is used to typeset various input texts with and without the OpenType Script applied for various alphabets. The text is only rendered correctly in the second case; many examples of incorrect diacritic spacing as well as a lack of contextual ligatures and rearrangement can be seen. Thanks to Jonathan Kew, Yves Codet and Gildas Hamel for their contributions towards these examples.

⁵<http://www.code2000.net/>

Example 19: An example of various Scripts and Languages.

العربي	العربي	
हिन्दी	हिन्दी	
લઢ	લઢ	
ਮਧਾਦ-ਸੂਧਕ ਨਵਿਦਨ	ਮਧਾਦ-ਸੂਧਕ ਨਵਿਦਨ	
നമ്മുടറ പാരബരയ	നമ്മുടറ പാരബരയ	
ਆਦਿ ਸਚੁ ਜੁਗਾਦਿ ਸਚੁ	ਆਦਿ ਸਚੁ ਜੁਗਾਦਿ ਸਚੁ	
தமிழ் துடி	தமிழ் துடி	
קאָפּ סוֹ מוֹי	קאָפּ סוֹ מוֹי	
cáp số mõi	cáp số mõi	
		<pre> \testfeature{Script=Arabic}{\arabictext} \testfeature{Script=Devanagari}{\devanagaritext} \testfeature{Script=Bengali}{\bengalitext} \testfeature{Script=Gujarati}{\gujaratitext} \testfeature{Script=Malayalam}{\malayalamtext} \testfeature{Script=Gurmukhi}{\gurmukhitext} \testfeature{Script=Tamil}{\tamilttext} \testfeature{Script=Hebrew}{\hebrewtext} \def\examplefont{DoulosSILR.ttf} \testfeature{Language=Vietnamese}{\vietnamesetext} </pre>

Table 2: Defined Scripts for OpenType fonts. Aliased names are shown in adjacent positions marked with red pilcrows (¶).

Adlam	Glagolitic	Marchen	Rejang
Ahom	Gothic	¶Math	Runic
Anatolian Hieroglyphs	Grantha	¶Maths	Samaritan
Arabic	Greek	Meitei Mayek	Saurashtra
Armenian	Gujarati	Mende Kikakui	Sharada
Avestan	Gurmukhi	Meroitic Cursive	Shavian
Balinese	Hangul Jamo	Meroitic Hieroglyphs	Siddham
Bamum	Hangul	Miao	Sign Writing
Bassa Vah	Hanunoo	Modi	Sinhala
Batak	Hatran	Mongolian	Sora Sompeng
Bengali	Hebrew	Mro	Sumero-Akkadian
Bhaiksuki	¶Hiragana and Katakana	Multani	Cuneiform
Bopomofo	¶Kana	Musical Symbols	Sundanese
Brahmi	Imperial Aramaic	Myanmar	Syloti Nagri
Braille	Inscriptional Pahlavi	¶N'Ko	Syriac
Buginese	Inscriptional Parthian	¶N'ko	Tagalog
Buhid	Javanese	Nabataean	Tagbanwa
Byzantine Music	Kaithi	Newa	Tai Le
Canadian Syllabics	Kannada	Ogham	Tai Lu
Carian	Kayah Li	Ol Chiki	Tai Tham
Caucasian Albanian	Kharosthi	Old Italic	Tai Viet
Chakma	Khmer	Old Hungarian	Takri
Cham	Khojki	Old North Arabian	Tamil
Cherokee	Khudawadi	Old Permic	Tangut
¶CJK	Lao	Old Persian Cuneiform	Telugu
¶CJK Ideographic	Latin	Old South Arabian	Thaana
Coptic	Lepcha	Old Turkic	Thai
Cypriot Syllabary	Limbu	¶Oriya	Tibetan
Cyrillic	Linear A	¶Odia	Tifinagh
Default	Linear B	Osage	Tirhuta
Deseret	Lisu	Osmanya	Ugaritic Cuneiform
Devanagari	Lycian	Pahawh Hmong	Vai
Duployan	Lydian	Palmyrene	Warang Citi
Egyptian Hieroglyphs	Mahajani	Pau Cin Hau	Yi
Elbasan	Malayalam	Phags-pa	
Ethiopic	Mandaic	Phoenician	
Georgian	Manichaean	Psalter Pahlavi	

Table 3: Defined Languages for OpenType fonts. Aliased names are shown in adjacent positions marked with red pilcrows (¶).

Abaza	Default	Igbo	Koryak	Norway House Cree	Serer
Abkhazian	Dogri	Ijo	Ladin	Nisi	South Slavey
Adyghe	Divehi	Ilokano	Lahuli	Niuean	Southern Sami
Afrikaans	Djerma	Indonesian	Lak	Nkole	Suri
Afar	Dangme	Ingush	Lambani	N'ko	Svan
Agaw	Dinka	Inuktitut	Lao	Dutch	Swedish
Altai	Dungan	Irish	Latin	Nogai	Swadaya Aramaic
Amharic	Dzongkha	Irish Traditional	Laz	Norwegian	Swahili
Arabic	Ebira	Icelandic	L-Cree	Northern Sami	Swazi
Aari	Eastern Cree	Inari Sami	Ladakhi	Northern Tai	Sutu
Arakanese	Edo	Italian	Lezgi	Esperanto	Syriac
Assamese	Efik	Hebrew	Lingala	Nynorsk	Tabasaran
Athapaskan	Greek	Javanese	Low Mari	Oji-Cree	Tajiki
Avar	English	Yiddish	Limbu	Ojibway	Tamil
Awadhi	Erzya	Japanese	Lomwe	Oriya	Tatar
Aymara	Spanish	Judezmo	Lower Sorbian	Oromo	TH-Cree
Azeri	Estonian	Jula	Lule Sami	Ossetian	Telugu
Badaga	Basque	Kabardian	Lithuanian	Palestinian Aramaic	Tongan
Baghelkhandi	Evenki	Kachchi	Luba	Pali	Tigre
Balkar	Even	Kalenjin	Luganda	Punjabi	Tigrinya
Baule	Ewe	Kannada	Luhya	Palpa	Thai
Berber	French Antillean	Karachay	Luo	Pashto	Tahitian
Bench	¶Farsi	Georgian	Latvian	Polytonic Greek	Tibetan
Bible Cree	¶Parsi	Kazakh	Majang	Pilipino	Turkmen
Belarussian	¶Persian	Kebena	Makua	Palauing	Temne
Bemba	Finnish	Khutsuri Georgian	Malayalam	Polish	Tswana
Bengali	Fijian	Khakass	Traditional	Provençal	Tundra Nenets
Bulgarian	Flemish	Khanty-Kazim	Mansi	Portuguese	Tonga
Bhili	Forest Nenets	Khmer	Marathi	Chin	Todo
Bhojpuri	Fon	Khanty-Shurishkar	Marwari	Rajasthani	Turkish
Bikol	Faroeese	Khanty-Vakhi	Mbundu	R-Cree	Tsonga
Bilen	French	Khowar	Manchu	Russian Buriat	Turoyo Aramaic
Blackfoot	Frisian	Kikuyu	Moose Cree	Riang	Tulu
Balochi	Friulian	Kirghiz	Mende	Rhaeto-Romanic	Tuvin
Balante	Futa	Kisii	Me'en	Romanian	Twi
Balti	Fulani	Kokni	Mizo	Romany	Udmurt
Bambara	Ga	Kalmyk	Macedonian	Rusyn	Ukrainian
Bamileke	Gaelic	Kamba	Male	Ruanda	Urdu
Breton	Gagauz	Kumaoni	Malagasy	Russian	Upper Sorbian
Brahui	Galician	Komo	Malinke	Sadri	Uyghur
Braj Bhasha	Garshuni	Komso	Malayalam	Sanskrit	Uzbek
Burmese	Garhwali	Kanuri	Reformed	Santali	Venda
Bashkir	Ge'ez	Kodagu	Malay	Sayisi	Vietnamese
Beti	Gilyak	Korean Old Hangul	Mandinka	Sekota	Wa
Catalan	Gumuz	Konkani	Mongolian	Selkup	Wagdi
Cebuano	Gondi	Kikongo	Manipuri	Sango	West-Cree
Chechen	Greenlandic	Komi-Permyak	Maninka	Shan	Welsh
Chaha Gurage	Garo	Korean	Manx Gaelic	Sibe	Wolof
Chattisgarhi	Guarani	Komi-Zyrian	Moksha	Sidamo	Tai Lue
Chichewa	Gujarati	Kpelle	Moldavian	Silte Gurage	Xhosa
Chukchi	Haitian	Krio	Mon	Skolt Sami	Yakut
Chipewyan	Halam	Karakalpak	Moroccan	Slovak	Yoruba
Cherokee	Haraui	Karelian	Maori	Slavey	Y-Cree
Chuvash	Hausa	Karaim	Maithili	Slovenian	Yi Classic
Comorian	Hawaiian	Karen	Maltese	Somali	Yi Modern
Coptic	Hammer-Banna	Koorete	Mundari	Samoan	Chinese Hong Kong
Cree	Hiligaynon	Kashmiri	Naga-Assamese	Sena	Chinese Phonetic
Carrier	Hindi	Khasi	Nanai	Sindhi	Chinese Simplified
Crimean Tatar	High Mari	Kildin Sami	Naskapi	Sinhalese	Chinese Traditional
Church Slavonic	Hindko	Kui	N-Cree	Soninke	Zande
Czech	Ho	Kulvi	Ndebele	Sodo Gurage	Zulu
Danish	Harari	Kumyk	Ndonga	Sotho	
Dargwa	Croatian	Kurdish	Nepali	Albanian	
Woods Cree	Hungarian	Kurukh	Newari	Serbian	
German	Armenian	Kuy	Nagari	Saraiki	

3 OpenType font features

There are a finite set of OpenType font features, and `fontspec` provides an interface to around half of them. Full documentation will be presented in the following sections, including how to enable and disable individual features, and how they interact.

A brief reference is provided ([Table 4 on the following page](#)) but note that this is an incomplete listing — only the ‘enable’ keys are shown, and where alternative interfaces are provided for convenience only the first is shown. (E.g., `Numbers=OldStyle` is the same as `Numbers=Lowercase`.)

For completeness, the complete list of OpenType features *not* provided with a `fontspec` interface is shown in [Table 5 on page 43](#). Features omitted are partially by design and partially by oversight; for example, the `aalt` feature is largely useless in $\TeX$ since it is designed for providing a GUI interface for selecting ‘all alternates’ of a glyph. Others, such as optical bounds for example, simply haven’t yet been considered due to a lack of fonts available for testing. Suggestions welcome for how/where to add these missing features to the package.

3.1 Tag-based features

3.1.1 Alternates — `salt`

The `Alternate` feature, alias `StylisticAlternates`, is used to access alternate font glyphs when variations exist in the font, such as in [Example 20](#). It uses a numerical selection, starting from zero, that will be different for each font. Note that the `Style=Alternate` option is equivalent to `Alternate=0` to access the default case.

Note that the indexing starts from zero. With the $\text{Lua}\TeX$ engine, `Alternate=Random` selects a random alternate.

See [Section 1 on page 66](#) for a way to assign names to alternates if desired.

3.1.2 Character Variants — `cvNN`

‘Character Variations’ are selected numerically to adjust the output of (usually) a single character for the particular font. These correspond to the OpenType features `cv01` to `cv99`.

For each character that can be varied, it is possible to select among possible options for that particular glyph. For example, in the hypothetical example below, variants are chosen for glyphs ‘4’ and ‘5’, and the trailing `:<n>` corresponds to which variety to choose.

```
\fontspec{CV Font}[CharacterVariant={4,5:2}] \& violet
```

Example 20: The `Alternate` feature.

A & h	<code>\fontspec{LinLibertine_R.otf}</code>
A & h	<code>\textsc{a} \& h \\\</code>
A & h	<code>\addfontfeature{Alternate=0}</code>
A & h	<code>\textsc{a} \& h</code>

Table 4: Summary of OpenType features in fontspec, alphabetic by feature tag.

ABVM	Diacritics = AboveBase	<i>Above-base Mark Positioning</i>	NLCK	CJKShape = NLC	<i>NLC Kanji Forms</i>
AFRC	Fractions = Alternate	<i>Alternative Fractions</i>	NUMR	VerticalPosition = Numerator	<i>Numerators</i>
BLWM	Diacritics = BelowBase	<i>Below-base Mark Positioning</i>	ONUM	Numbers = Lowercase	<i>Oldstyle Figures</i>
CALT	Contextuals = Alternate	<i>Contextual Alternates</i>	ORDN	VerticalPosition = Ordinal	<i>Ordinals</i>
CASE	Style = Uppercase	<i>Case-Sensitive Forms</i>	ORNM	Ornament = <i>N</i>	<i>Ornaments</i>
CLIG	Ligatures = Contextual	<i>Contextual Ligatures</i>	PALT	CharacterWidth = AlternateProportional	<i>Proportional Alternate Widths</i>
CPSP	Kerning = Uppercase	<i>Capital Spacing</i>	PCAP	Letters = PetiteCaps	<i>Petite Capitals</i>
CSWH	Contextuals = Swash	<i>Contextual Swash</i>	PKNA	Style = ProportionalKana	<i>Proportional Kana</i>
CVNN	CharacterVariant = <i>N:M</i>	<i>Character Variant N</i>	PNUM	Numbers = Proportional	<i>Proportional Figures</i>
C2PC	Letters = UppercasePetiteCaps	<i>Petite Capitals From Capitals</i>	PWID	CharacterWidth = Proportional	<i>Proportional Widths</i>
C2SC	Letters = UppercaseSmallCaps	<i>Small Capitals From Capitals</i>	QWID	CharacterWidth = Quarter	<i>Quarter Widths</i>
DLIG	Ligatures = Rare	<i>Discretionary Ligatures</i>	RAND	Letters = Random	<i>Randomize</i>
DNOM	VerticalPosition = Denominator	<i>Denominators</i>	RLIG	Ligatures = Required	<i>Required Ligatures</i>
EXPT	CJKShape = Expert	<i>Expert Forms</i>	RUBY	Style = Ruby	<i>Ruby Notation Forms</i>
FALT	Contextuals = LineFinal	<i>Final Glyph on Line Alternates</i>	SALT	Alternate = <i>N</i>	<i>Stylistic Alternates</i>
FINA	Contextuals = WordFinal	<i>Terminal Forms</i>	SINF	VerticalPosition = ScientificInferior	<i>Scientific Inferiors</i>
FRAC	Fractions = On	<i>Fractions</i>	SMCP	Letters = SmallCaps	<i>Small Capitals</i>
FWID	CharacterWidth = Full	<i>Full Widths</i>	SMPL	CJKShape = Simplified	<i>Simplified Forms</i>
HALT	CharacterWidth = AlternateHalf	<i>Alternate Half Widths</i>	ssNN	StylisticSet = <i>N</i>	<i>Stylistic Set N</i>
HIST	Style = Historic	<i>Historical Forms</i>	SSTY	Style = MathScript	<i>Math script style alternates</i>
HKNA	Style = HorizontalKana	<i>Horizontal Kana Alternates</i>	SUBS	VerticalPosition = Inferior	<i>Subscript</i>
HLIG	Ligatures = Historic	<i>Historical Ligatures</i>	SUPS	VerticalPosition = Superior	<i>Superscript</i>
HWID	CharacterWidth = Half	<i>Half Widths</i>	SWSH	Style = Swash	<i>Swash</i>
INIT	Contextuals = WordInitial	<i>Initial Forms</i>	TITL	Style = Titling	<i>Titling</i>
ITAL	Style = Italic	<i>Italics</i>	TNUM	Numbers = Monospaced	<i>Tabular Figures</i>
JP78	CJKShape = JIS1978	<i>JIS78 Forms</i>	TRAD	CJKShape = Traditional	<i>Traditional Forms</i>
JP83	CJKShape = JIS1983	<i>JIS83 Forms</i>	TWID	CharacterWidth = Third	<i>Third Widths</i>
JP90	CJKShape = JIS1990	<i>JIS90 Forms</i>	UNIC	Letters = Unicase	<i>Unicase</i>
JPO4	CJKShape = JIS2004	<i>JIS2004 Forms</i>	VALT	Vertical = AlternateMetrics	<i>Alternate Vertical Metrics</i>
KERN	Kerning = On	<i>Kerning</i>	VERT	Vertical = Alternates	<i>Vertical Writing</i>
LIGA	Ligatures = Common	<i>Standard Ligatures</i>	VHAL	Vertical = HalfMetrics	<i>Alternate Vertical Half Metrics</i>
LNUM	Numbers = Uppercase	<i>Lining Figures</i>	VKNA	Style = VerticalKana	<i>Vertical Kana Alternates</i>
LOCL	LocalForms = On	<i>Localized Forms</i>	VKRN	Vertical = Kerning	<i>Vertical Kerning</i>
MARK	Diacritics = MarkToBase	<i>Mark Positioning</i>	VPAL	Vertical = ProportionalMetrics	<i>Proportional Alternate Vertical Metrics</i>
MEDI	Contextuals = Inner	<i>Medial Forms</i>	VRT2	Vertical = RotatedGlyphs	<i>Vertical Alternates and Rotation</i>
MKMK	Diacritics = MarkToMark	<i>Mark to Mark Positioning</i>	VRTR	Vertical = AlternatesForRotation	<i>Vertical Alternates for Rotation</i>
NALT	Annotation = <i>N</i>	<i>Alternate Annotation Forms</i>	ZERO	Numbers = SlashedZero	<i>Slashed Zero</i>

Table 5: List of *unsupported* OpenType features.

AALT <i>Access All Alternates</i>	HNGL <i>Hangul</i>	RCLT <i>Required Contextual Alternates</i>
ABVF <i>Above-base Forms</i>	HOJO <i>Hojo Kanji Forms</i>	RKRF <i>Rakar Forms</i>
ABVS <i>Above-base Substitutions</i>	ISOL <i>Isolated Forms</i>	RPHF <i>Reph Forms</i>
AKHN <i>Akhands</i>	JALT <i>Justification Alternates</i>	RTBD <i>Right Bounds</i>
BLWF <i>Below-base Forms</i>	LFBD <i>Left Bounds</i>	RTL A <i>Right-to-left alternates</i>
BLWS <i>Below-base Substitutions</i>	LJMO <i>Leading Jamo Forms</i>	RTL M <i>Right-to-left mirrored forms</i>
CCMP <i>Glyph Composition / Decomposition</i>	LTRA <i>Left-to-right alternates</i>	RVRN <i>Required Variation Alternates</i>
CFAR <i>Conjunct Form After Ro</i>	LTRM <i>Left-to-right mirrored forms</i>	SIZE <i>Optical size</i>
CJCT <i>Conjunct Forms</i>	MED2 <i>Medial Forms #2</i>	STCH <i>Stretching Glyph Decomposition</i>
CPCT <i>Centered CJK Punctuation</i>	MGRK <i>Mathematical Greek</i>	TJMO <i>Trailing Jamo Forms</i>
CURS <i>Cursive Positioning</i>	MSET <i>Mark Positioning via Substitution</i>	TNAM <i>Traditional Name Forms</i>
DIST <i>Distances</i>	NUKT <i>Nukta Forms</i>	VATU <i>Vattu Variants</i>
DTLS <i>Dotless Forms</i>	OPBD <i>Optical Bounds</i>	VJMO <i>Vowel Jamo Forms</i>
FIN2 <i>Terminal Forms #2</i>	PREF <i>Pre-Base Forms</i>	
FIN3 <i>Terminal Forms #3</i>	PRES <i>Pre-base Substitutions</i>	
FLAC <i>Flattened accent forms</i>	PSTF <i>Post-base Forms</i>	
HALF <i>Half Forms</i>	PSTS <i>Post-base Substitutions</i>	
HALN <i>Halant Forms</i>		

The numbering is entirely font-specific. Glyph ‘5’ might be the character ‘v’, for example. Character variants are specifically designed not to conflict with each other, so you can enable them individually per character. (Unlike stylistic alternates, say.) Note that the indexing starts from zero.

3.1.3 Contextuals

This feature refers to substitutions of glyphs that vary ‘contextually’ by their relative position in a word or string of characters; features such as contextual swashes are accessed via the options shown in [Table 6](#).

Historic forms are accessed in OpenType fonts via the feature `Style=Historic`; this is generally *not* contextual in OpenType, which is why it is not included in this feature.

3.1.4 Diacritics

Specifies how combining diacritics should be placed. These will usually be controlled automatically according to the Script setting.

3.1.5 Fractions — frac

Activates the construction of ‘vulgar’ fractions using precomposed glyphs and/or subscript and superscript characters from within the font. Coverage will vary by font; see [Example 21](#). Some (Asian fonts predominantly) also provide for the `Alt` option.

Table 6: Options for the OpenType font feature ‘Contextuals’.

Feature	Option	Tag
Contextuals	= Swash	cswh †
	Alternate	calt †
	WordInitial	init †
	WordFinal	fini †
	LineFinal	falt †
	Inner	medi †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Table 7: Options for the OpenType font feature ‘Diacritics’.

Feature	Option	Tag
Diacritics	= MarkToBase	mark †
	MarkToMark	mkmk †
	AboveBase	abvm †
	BelowBase	blwm †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Table 8: Options for the OpenType font feature ‘Fractions’.

Feature	Option	Tag
Fractions	= On	+frac
	Off	-frac
	Reset	
Alternate		afrc †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 21: The Fractions feature.

	<code>\setsansfont{IBMPlexSans-Regular.otf}[Fractions=On]</code>
	<code>\setmonofont{IBMPlexMono-Regular.otf}[Fractions=On]</code>
$\frac{1}{2}$ $\frac{47}{11}$ $\frac{1}{4000}$	<code>\sffamily 1/2 47/11 1/1000 \par</code>
$\frac{1}{2}$ 47/11	<code>\ttfamily 1/2 47/11</code>

3.1.6 Kerning — kern

Specifies how inter-glyph spacing should behave. Well-made fonts include information for how differing amounts of space should be inserted between separate character pairs. This kerning space is inserted automatically but in rare circumstances you may wish to turn it off.

As briefly mentioned previously at the end of 3.1.7, the `Uppercase` option will add a small amount of tracking between uppercase letters, seen in Example 22, which uses the *Romande* fonts⁶ (thanks to Clea F. Rees for the suggestion). The `Uppercase` option acts separately to the regular kerning controlled by the `On/Off` options.

3.1.7 Letters

The `Letters` feature specifies how the letters in the current font will look. OpenType fonts may contain the following options: `SmallCaps`, `PetiteCaps`, `UppercaseSmallCaps`, `UppercasePetiteCaps`, and `Unicase`. Additionally `Uppercase` and `Lowercase` are supported for all fonts in Lua_{TEX}. In contrast to earlier version, the `Uppercase` and `Lowercase` options turn the text into uppercase or lowercase and do not require the text to already have the right casing. The old behavior of `Uppercase` is available with `Style=Uppercase`. When the `Uppercase` option is selected, `Style=Uppercase` and `Kerning=Uppercase` are automatically applied if supported by the font.

`PetiteCaps` are smaller than small caps. `SmallCaps` and `PetiteCaps` turn lowercase letters into the smaller caps letters, whereas the `Uppercase...` options turn the *capital* letters into the smaller caps (good, *e.g.*, for applying to already uppercase acronyms like ‘NASA’). This difference is shown in Example 23. ‘`Unicase`’ is a weird hybrid of upper and lower case letters.

3.1.8 Ligatures

`Ligatures` refer to the replacement of two separate characters with a specially drawn glyph for functional or aesthetic reasons. The list of options, of which multiple may be selected at one time, is shown in Table 11. A demonstration with the Linux Libertine fonts⁷ is shown in Example 24.

⁶<http://arkandis.tuxfamily.org/adffonts.html>

⁷<http://www.linuxlibertine.org/>

Table 9: Options for the OpenType font feature ‘Kerning’.

Feature	Option	Tag
Kerning =	On	+kern
	Off	-kern
	Reset	
	Uppercase csp	†
	ResetAll	

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 22: Adding extra kerning for uppercase letters. (The difference is usually very small.)

UPPERCASE EXAMPLE
UPPERCASE EXAMPLE

```
\fontspec{RomandeADFStd-DemiBold.otf}
UPPERCASE EXAMPLE \\
\addfontfeature{Kerning=Uppercase}
UPPERCASE EXAMPLE
```

Table 10: Options for the OpenType font feature ‘Letters’.

Feature	Option	Tag
Letters =	SmallCaps	smcp †
	PetiteCaps	pcap †
	UppercaseSmallCaps	c2sc †
	UppercasePetiteCaps	c2pc †
	Unicase	unic †
	Uppercase	†
	Lowercase	†
	ResetAll	

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 23: Small caps from lowercase or uppercase letters.

	<code>\fontspec{Coelac.otf}[Letters=SmallCaps]</code>
	THIS SENTENCE no verb \\
THIS SENTENCE NO VERB	<code>\fontspec{Coelac.otf}[Letters=UppercaseSmallCaps]</code>
THIS SENTENCE no verb	THIS SENTENCE no verb \\
THIS SENTENCE NO VERB	<code>\fontspec{Coelac.otf}[Letters=PetiteCaps]</code>
	THIS SENTENCE no verb

Note the additional features accessed with `Ligatures=TeX`. These are not actually real OpenType features, but additions provided by `luaotfload` (i.e., Lua $\TeX$ only) to emulate $\TeX$'s behaviour for ASCII input of curly quotes and punctuation. In $\text{X}\mathfrak{L}\mathfrak{A}\mathfrak{T}\mathfrak{E}\mathfrak{X}$ this is achieved with the Mapping feature (see [Section 1.1 on page 61](#)) but for consistency `Ligatures=TeX` will perform the same function as `Mapping=tex-text`.

3.1.9 Localised Forms — `locl`

This feature enables and disables glyph substitutions, etc., that are specific to the Language selected in the font. This feature is automatically activated by default when present, so it should not be generally necessary to use `LocalForms = On`. In certain scenarios it may be important to turn it `Off` (although nothing specifically springs to mind).

3.1.10 Numbers

The Numbers feature defines how numbers will look in the selected font, accepting options shown in [Table 13](#).

The synonyms `Uppercase` and `Lowercase` are equivalent to `Lining` and `OldStyle`, respectively. The differences have been shown previously in [Section 2 on page 22](#). The `Monospaced` option is useful for tabular material when digits need to be vertically aligned.

The `SlashedZero` option replaces the default zero with a slashed version to prevent confusion with an uppercase 'O', shown in [Example 25](#).

The `Arabic` option (with tag `anum`) maps regular numerals to their Arabic script or Persian equivalents based on the current Language setting (see [Section 2 on page 38](#)). This option is based on a Lua $\TeX$ feature of the `luaotfload` package, not an OpenType feature. (Thus, this feature is unavailable in $\text{X}\mathfrak{L}\mathfrak{A}\mathfrak{T}\mathfrak{E}\mathfrak{X}$.) This feature should be considered deprecated; while there are no plans to remove it from this package, if its support is dropped from the font loader it could disappear from `fontspec` with little notice.

3.1.11 Ornament — `ornm`

Ornaments are selected with the `Ornament` feature (OpenType feature `ornm`), selected numerically such as for the `Annotation` feature.

Table 11: Options for the OpenType font feature 'Ligatures'.

Feature	Option	Tag
Ligatures =	Required	<code>rlig</code> †
	Common	<code>liga</code> †
	Contextual	<code>clig</code> †
	Rare/Discretionary	<code>dlig</code> †
	Historic	<code>hlig</code> †
	TeX	<code>tlig</code> †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 24: An example of the Ligatures feature.

strict	→	strict	<pre> \def\test#1#2{% #2 \$\to\$ {\addfontfeature{#1} #2}\} \fontspec{LinLibertine_R.otf} \test{Ligatures=Historic}{strict} \test{Ligatures=Rare}{wurtzite} \test{Ligatures=CommonOff}{firefly} </pre>
wurtzite	→	wurtzite	
firefly	→	firefly	

Table 12: Options for the OpenType font feature ‘LocalForms’.

Feature	Option	Tag
LocalForms =	On	+locl
	Off	-locl
	Reset	

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Table 13: Options for the OpenType font feature ‘Numbers’.

Feature	Option	Tag
Numbers =	Uppercase	lnum †
	Lowercase	onum †
	Lining	lnum †
	OldStyle	onum †
	Proportional	pnum †
	Monospaced	tnum †
	SlashedZero	zero †
	Arabic	anum †
	ResetAll	

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 25: The effect of the SlashedZero option.

	<pre> \fontspec[Numbers=Lining]{texgyrebonum-regular.otf} 0123456789 \fontspec[Numbers=SlashedZero]{texgyrebonum-regular.otf} 0123456789 </pre>
--	---

3.1.12 Style

‘Ruby’ refers to a small optical size, used in Japanese typography for annotations. For fonts with multiple `slat` OpenType features, use the `fontspec Alternate` feature instead.

Example 26 shows an example of a font feature that involves glyph substitution for particular letters within an alphabet. Other options in these categories operate in similar ways, with the choice of how particular substitutions are organised with which feature largely up to the font designer.

The `Uppercase` option is designed to select various uppercase forms for glyphs such as accents and dashes, such as shown in Example 27; note the raised position of the hyphen to better match the surrounding letters. It will (probably) not actually map letters to uppercase.⁸ This option used to be selected under the `Letters` feature, but moved here as it generally does not actually affect the letters themselves. The `Kerning` feature also contains an `Uppercase` option, which adds a small amount of spacing in between letters (see 3.1.6 on page 45).

In other features, larger breadths of changes can be seen, covering the style of an entire alphabet. For instance, in some Japanese fonts features such as `Style=Italic` or `Style=Ruby` respectively change the style of all Latin characters to italic or all Hiragana characters to a darker optical shape:

```
\fontspec{Hiragino Mincho Pro}
Latin \kana      \
\addfontfeature{Style={Italic, Ruby}}
Latin \kana
```

3.1.13 Stylistic Set variations — `ssNN`

This feature selects a ‘Stylistic Set’ variation, which usually corresponds to an alternate glyph style for a range of characters (usually an alphabet or subset thereof). This feature is specified numerically. These correspond to OpenType features `ss01`, `ss02`, etc.

Two demonstrations from the `Junicode` font⁹ are shown in Example 28 and Example 29; thanks to Adam Buchbinder for the suggestion.

Multiple stylistic sets may be selected simultaneously by writing, e.g., `StylisticSet={1,2,3}`.

The `StylisticSet` feature is a synonym of the `Variant` feature for `AAT` fonts. See Section 1 on page 66 for a way to assign names to stylistic sets, which should be done on a per-font basis.

⁸If you want automatic uppercase letters, look to L^AT_EX’s `\MakeUppercase` command or, when using Lua_TE_X, to the `Letters` feature.

⁹<http://junicode.sf.net>

Example 26: Example of the `Alternate` option of the `Style` feature.

M Q W
M Q W

```
\fontspec{Quattrocento-Regular.ttf}
M Q W      \
\addfontfeature{Style=Alternate}
M Q W
```

Table 14: Options for the OpenType font feature ‘Style’.

Feature Option	Tag
Style = Alternate	salt †
Cursive	curs †
Historic	hist †
Italic	ital †
Ruby	ruby †
Swash	swsh †
Titling	titl †
Uppercase	case †
HorizontalKana	hkna †
VerticalKana	vkna †
ResetAll	

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 27: An example of the Uppercase option of the Style feature.

UPPER-CASE example	<code>\fontspec{LinLibertine_R.otf}</code> <code>UPPER-CASE example \\</code>
UPPER-CASE example	<code>\addfontfeature{Style=Uppercase}</code> <code>UPPER-CASE example</code>

Example 28: Insular letterforms, as used in medieval Northern Europe, for the Junicode font accessed with the StylisticSet feature.

Insular forms.	<code>\fontspec{Junicode}</code> <code>Insular forms. \\</code>
Insular forms.	<code>\addfontfeature{StylisticSet=2}</code> <code>Insular forms. \\</code>

Example 29: Enlarged minuscules (capital letters remain unchanged) for the Junicode font, accessed with the StylisticSet feature.

ENLARGED Minuscules.	<code>\fontspec{Junicode}</code> <code>ENLARGED Minuscules. \\</code>
ENLARGED Minuscules.	<code>\addfontfeature{StylisticSet=6}</code> <code>ENLARGED Minuscules. \\</code>

3.1.14 Vertical Position

The `VerticalPosition` feature is used to access things like subscript (`Inferior`) and superscript (`Superior`) numbers and letters (and a small amount of punctuation, sometimes). The `Ordinal` option will only raise characters that are used in some languages directly after a number. The `ScientificInferior` feature will move glyphs further below the baseline than the `Inferior` feature. These are shown in Example 30

Numerator and Denominator should only be used for creating arbitrary fractions (see next section).

The `realscripts` package (which is also loaded by `xltxtra` for $\text{\LaTeX}$) redefines the `\textsubscript` and `\textsuperscript` commands to use the above font features automatically, including for use in footnote labels. If this is the only feature of `xltxtra` you wish to use, consider loading `realscripts` on its own instead.

3.2 CJK features

This section summarises the features which are largely intending for Chinese, Korean, and Japanese typesetting.

3.2.1 Annotation — `nalt`

Some fonts are equipped with an extensive range of numbers and numerals in different forms. These are accessed with the `Annotation` feature (OpenType feature `nalt`), selected numerically. Note that the indexing starts from zero.

```
\fontspec{Hiragino Maru Gothic Pro}
 1 2 3 4 5 6 7 8 9
\def\x#1{\{\addfontfeature{Annotation=#1}
 1 2 3 4 5 6 7 8 9 \}}
\x0\x1\x2\x3\x4\x5\x6\x7\x8\x9
```

Table 15: Options for the OpenType font feature ‘VerticalPosition’.

Feature	Option	Tag
VerticalPosition =	Superior	sups †
	Inferior	subs †
	Numerator	numr †
	Denominator	dnom †
	ScientificInferior	sinf †
	Ordinal	ordn †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 30: The VerticalPosition feature.

	<code>\fontspec{LibreCaslonText-Regular.otf}[VerticalPosition=Superior]</code>
Superior: 1234567890	Superior: 1234567890
	<code>\fontspec{LibreCaslonText-Regular.otf}[VerticalPosition=Numerator]</code>
Numerator: 12345	Numerator: 12345
	<code>\fontspec{LibreCaslonText-Regular.otf}[VerticalPosition=Denominator]</code>
Denominator: 12345	Denominator: 12345
	<code>\fontspec{LibreCaslonText-Regular.otf}[VerticalPosition=ScientificInferior]</code>
Scientific Inferior: 12345	Scientific Inferior: 12345

3.2.2 Character width

Many Asian fonts are equipped with variously spaced characters for shoe-horning into their generally monospaced text. These are accessed through the `CharacterWidth` feature.

Japanese alphabetic glyphs (in Hiragana or Katakana) may be typeset proportionally, to better fit horizontal measures, or monospaced, to fit into the rigid grid imposed by ideographic typesetting. In this latter case, there are also half-width forms for squeezing more kana glyphs (which are less complex than the kanji they are amongst) into a given block of space. The same features are given to roman letters in Japanese fonts, for typesetting foreign words in the same style as the surrounding text. Example omitted until I find an open source font which supports these features.

3.2.3 CJK shape

There have been many standards for how CJK ideographic glyphs are ‘supposed’ to look. Some fonts will contain many alternate glyphs available in order to be able to display these glyphs correctly in whichever form is appropriate. Both AAT and OpenType fonts support the following `CJKShape` options: `Traditional`, `Simplified`, `JIS1978`, `JIS1983`, `JIS1990`, and `Expert`. OpenType also supports the `NLC` option.

Table 16: Options for the OpenType font feature ‘`CharacterWidth`’.

Feature	Option	Tag
CharacterWidth =	Proportional	pwid †
	Full	fwid †
	Half	hwid †
	Third	twid †
	Quarter	qwid †
	AlternateProportional	palt †
	AlternateHalf	halt †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Table 17: Options for the OpenType font feature ‘CJKShape’.

Feature	Option	Tag
CJKShape =	Traditional	trad
	Simplified	smp1
	JIS1978	jp78
	JIS1983	jp83
	JIS1990	jp90
	Expert	expt
	NLC	nlck

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Example 31: Different standards for CJK ideograph presentation.

啞嚙軀 妍并訝
啞嚙軀 妍并訝

```
\fontspec{NotoSansJP-Regular.ttf}
{\addfontfeature{CJKShape=Traditional}
\text }
{\addfontfeature{CJKShape=NLC}
\text }
```

3.2.4 Vertical typesetting

OpenType provides a plethora of features for accommodating the varieties of possibilities needed for vertical typesetting (CJK and others). No capabilities for achieving such vertical typesetting are provided by `fontspec`, however; please get in touch if there are improvements that could be made.

Table 18: Options for the OpenType font feature ‘Vertical’.

Feature	Option	Tag
Vertical	= RotatedGlyphs	vrt2 †
	AlternatesForRotation	vrtr †
	Alternates	vert †
	KanaAlternates	vkna †
	Kerning	vkrr †
	AlternateMetrics	valt †
	HalfMetrics	vhal †
	ProportionalMetrics	vpal †
ResetAll		

† These feature options can be disabled with `..Off` variants, and reset to default state (neither explicitly on nor off) with `..Reset`.

Part V

Commands for accents and symbols (‘encodings’)

The functionality described in this section is experimental.

In the pre-Unicode era, significant work was required by $\text{\LaTeX}$ to ensure that input characters in the source could be interpreted correctly depending on file encoding, and that glyphs in the output were selected correctly depending on the font encoding. With Unicode, we have the luxury of a single file and font encoding that is used for both input and output.

While this may provide some illusion that we could get away simply with typing Unicode text and receive correct output, this is not always the case. For a start, hyphenation in particular is language-specific, so tags should be used when switch between languages in a document. The `babel` and `polyglossia` packages both provide features for this.

Multilingual documents will often use different fonts for different languages, not just for style, but for the more pragmatic reason that fonts do not all contain the same glyphs. (In fact, only test fonts such as Code2000 provide anywhere near the full Unicode coverage.) Indeed, certain fonts may be perfect for a certain application but miss a handful of necessary diacritics or accented letters. In these cases, `fontspec` can leverage the font encoding technology built into $\text{\LaTeX}$ 2 ϵ to provide on a per-font basis either provide fallback options or error messages when a desired accent or symbol is not available. However, at present these features can only be provided for input using $\text{\LaTeX}$ commands rather than Unicode input; for example, typing `\`e` instead of `\textcopyright` instead of `\textcopyright` in the source file.

The most widely-used encoding in $\text{\LaTeX}$ 2 ϵ was T1 with companion ‘TS1’ symbols provided by the `textcomp` package. These encodings provided glyphs to typeset text in a variety of western European languages. As with most legacy $\text{\LaTeX}$ 2 ϵ input methods, accents and symbols were input using encoding-dependent commands such as `\`e` as described above. As of 2017, in $\text{\LaTeX}$ 2 ϵ on $\text{\XeTeX}$ and $\text{\LuaTeX}$, the default encoding is TU, which uses Unicode for input and output. The TU encoding provides appropriate encoding-dependent definitions for input commands to match the coverage of the T1+TS1 encodings. Wider coverage is not provided by default since (a) each font will provide different glyph coverage, and (b) it is expected that most users will be writing with direct Unicode input.

For those users who do need finer-grained control, `fontspec` provides an interface for a more extensible system.

1 A new Unicode-based encoding from scratch

Let’s say you need to provide support for a document originally written with fonts in the OT2 encoding, which contains encoding-dependent commands for Cyrillic letters. An example from the OT2 encoding definition file (`ot2enc.def`) reads:

```

57 \DeclareTextSymbol{\CYRIE}{OT2}{5}
58 \DeclareTextSymbol{\CYRDJE}{OT2}{6}
59 \DeclareTextSymbol{\CYRTSHE}{OT2}{7}
60 \DeclareTextSymbol{\cyrnje}{OT2}{8}
61 \DeclareTextSymbol{\cyrlje}{OT2}{9}
62 \DeclareTextSymbol{\cyrdzhe}{OT2}{10}

```

To recreate this encoding in a form suitable for `fontspec`, create a new file named, say, `fontrange-cyr.def` and populate it with

```

...
\DeclareTextSymbol{\CYRIE} {\LastDeclaredEncoding}{\0404}
\DeclareTextSymbol{\CYRDJE} {\LastDeclaredEncoding}{\0402}
\DeclareTextSymbol{\CYRTSHE}{\LastDeclaredEncoding}{\040B}
\DeclareTextSymbol{\cyrnje} {\LastDeclaredEncoding}{\045A}
\DeclareTextSymbol{\cyrlje} {\LastDeclaredEncoding}{\0459}
\DeclareTextSymbol{\cyrdzhe}{\LastDeclaredEncoding}{\045F}
...

```

The numbers `\0404`, `\0402`, ..., are the Unicode slots (in hexadecimal) of each glyph respectively. The `fontspec` package provides a number of shorthands to simplify this style of input; in this case, you could also write

```

\EncodingSymbol{\CYRIE}{\0404}
...

```

To use this encoding in a `fontspec` font, you would first add this to your preamble:

```

\DeclareUnicodeEncoding{unicyr}{
  \input{fontrange-cyr.def}
}

```

Then follow it up with a font loading call such as

```

\setmainfont{...}[NFSSEncoding=unicyr]

```

The first argument `unicyr` is the name of the ‘encoding’ to use in the font family. (There’s nothing special about the name chosen but it must be unique.) The second argument to `\DeclareUnicodeEncoding` also allows adjustments to be made for per-font changes. We’ll cover this use case in the next section.

2 Adjusting a pre-existing encoding

There are three reasons to adjust a pre-existing encoding: to add, to remove, and to redefine some symbols, letters, and/or accents.

When adding symbols, etc., simply write

```

\DeclareUnicodeEncoding{unicyr}{
  \input{tuenc.def}
  \input{fontrange-cyr.def}
  \EncodingSymbol{\textruble}{\20BD}
}

```

Of course if you consistently add a number of symbols to an encoding it would be a good idea to create a new `fontrange-XX.def` file to suit your needs.

When removing symbols, use the `\UndeclareSymbol{<cmd>}` command. For example, if you are loading a font that you know is missing, say, the interrobang (not that unusual a situation), you might write:

```
\DeclareUnicodeEncoding{nobang}{
  \input{tuenc.def}
  \UndeclareSymbol\textinterrobang
}
```

Provided that you use the command `\textinterrobang` to typeset this symbol, it will appear in fonts with the default encoding, while in any font loaded with the `nobang` encoding an attempt to access the symbol will either use the default fallback definition or return an error, depending on the symbol being undeclared.

The third use case is to redefine a symbol or accent. The most common use case in this scenario is to adjust a specific accent command to either fine-tune its placement or to ‘fake’ it entirely. For example, the underdot diacritic is used in typeset Sanskrit, but it is not necessarily included as an accent symbol in all fonts. By default the underdot is defined in TU as:

```
\EncodingAccent{\d}{"0323}
```

For fonts with a missing (or poorly-spaced) "0323 accent glyph, the ‘traditional’ $\TeX$ fake accent construction could be used instead:

```
\DeclareUnicodeEncoding{fakeacc}{
  \input{tuenc.def}
  \EncodingCommand{\d}[1]{%
    \hmode\bgroup
      \o@lign{\relax#1\crrc\hidewidth\ltx@sh@ft{-1ex}.\hidewidth}%
    \egroup
  }
}
```

This would be set up in a document as such:

```
\newfontfamily\sanskritfont{CharisSIL}
\newfontfamily\titlefont{Posterama}[NFSSEncoding=fakeacc]
```

Then later in the document, no additional work is needed:

```
...{\titlefont kalita\d m}... % <- uses fake accent
...{\sanskritfont kalita\d m}... % <- uses real accent
```

To reiterate from above, typing this input with Unicode text (‘kalitaṃ’) will *bypass* this encoding mechanism and you will receive only what is contained literally within the font.

3 Summary of commands

The L^AT_EX 2_ε kernel provides the following font encoding commands suitable for Unicode encodings:

```
\DeclareTextCommand{<command>}{<encoding>}[<num>][<default>]{<code>}
\DeclareUnicodeAccent{<command>}{<encoding>}{<slot>}
\DeclareTextSymbol{<command>}{<encoding>}{<slot>}
\DeclareTextComposite{<command>}{<encoding>}{<letter>}{<slot>}
\DeclareTextCompositeCommand{<command>}{<encoding>}{<letter>}{<code>}
\UndeclareTextCommand{<command>}{<encoding>}
```

See `fntguide.pdf` for full documentation of these. As shown above, the following short-hands are provided by `fontspec` to simplify the process of defining Unicode font range encodings:

```
\EncodingCommand{<command>}[<num>][<default>]{<code>}
\EncodingAccent{<command>}{<code>}
\EncodingSymbol{<command>}{<code>}
\EncodingComposite{<command>}{<letter>}{<slot>}
\EncodingCompositeCommand{<command>}{<letter>}{<code>}
\UndeclareSymbol{<command>}
\UndeclareAccent{<command>}
\UndeclareCommand{<command>}
\UndeclareComposite{<command>}{<letter>}
```

Part VI

LuaTeX-only font features

1 Different font technologies and shapers

LuaTeX does not directly support any font rendering technologies out of the box, it requires additional functionality to be added to properly support and control technologies such as OpenType.

Using the `Renderer` feature, there are a number of options that `fontspec` can pass to the engine to control which font technology is being used. Pre-2019, there were two options provided by `luaotfload` that generally did not require user intervention.

- `Renderer = Node` : the default ‘mode’ for typesetting OpenType fonts.
- `Renderer = Base` : a simplified mode useful only in a limited number of situations such as mathematics typesetting.

From 2019 the possibility of using the Harfbuzz text shaping engine within LuaTeX has been developed by Khaled Hosny. When running a suitable LuaTeX engine with Harfbuzz support, `fontspec` provides the following options:

- `Renderer = HarfBuzz` : use the Harfbuzz engine without an explicit ‘shaper’ (the old Harfbuzz name is kept for compatibility).
- `Renderer = OpenType` : use the Harfbuzz engine with the OpenType shaper.
- `Renderer = AAT` : use the Harfbuzz engine with the AAT shaper.
- `Renderer = Graphite` : use the Harfbuzz engine with the Graphite shaper.
- `Renderer = <foo>` : use the Harfbuzz engine with the `<foo>` shaper.

Support for the Harfbuzz renderer is preliminary and may be improved over time. Please treat the interface for Harfbuzz fonts as subject to change.

2 Custom font features

LuaTeX, via the `luaotfload` package, allows the definition and re-definition of custom OpenType features for a selected font. This facility is particularly useful to implement custom substitutions or to disable unwanted but not all ligatures.

Figure 1 shows an minimal example of this type of functionality. This example creates a new OpenType feature, `oneb`, which substitutes the glyph when typesetting ‘1’ for the named glyph `one.ss01`. The glyph names are font specific and can be interrogated with third-party software such as *FontForge*.

A third-party collection of additional examples are maintained in the repository ‘`fonts-in-luatex`’¹⁰. These examples are intended to correct or adjust font features in a range of commercial fonts and provide a good introduction to some of the possibilities that LuaTeX affords.

Please refer to the LuaTeX/luaotfload documentation for more details.

¹⁰<https://github.com/mewtant/fonts-in-luatex>

Figure 1: An example of custom font features.

```
\documentclass{article}
\usepackage{fontspec}
\directlua{
  fonts.handlers.otf.addfeature {
    name = "oneb",
    type = "substitution",
    data = {
      ["1"] = "one.ss01",
    }
  }
}
\setmainfont{Vollkorn-Regular.otf}[RawFeature=+oneb]
\begin{document}
1234567890
\end{document}
```

Part VII

Fonts and features with X_YTeX

1 X_YTeX-only font features

The features described here are available for any font selected by `fontspec`.

1.1 Mapping

The `Mapping` feature enables a X_YTeX text-mapping scheme, with an example shown in Example 32.

Only one mapping can be active at a time and a second call to `Mapping` will override the first. Using the `tex-text` mapping is also equivalent to writing `Ligatures=TeX`. The use of the latter syntax is recommended for better compatibility with LuaTeX documents.

1.2 Different font technologies: AAT, OpenType, and Graphite

Note that from 2020 it appears that X_YTeX can no longer support AAT fonts in macOS.

X_YTeX supports three rendering technologies for typesetting, selected with the `Renderer` font feature. The first, AAT, is that provided only by macOS. The second, OpenType, is an open source OpenType interpreter. It provides greater support for OpenType features, notably contextual arrangement, over AAT. The third is Graphite, which is an alternative to OpenType with particular features for less-common languages and the capability for more powerful font options. Features for OpenType have already been discussed in [IV on page 36](#); Graphite and AAT features are discussed later in [Section 2 on the following page](#) and [Section 3 on the next page](#).

Unless you have a particular need, the `Renderer` feature is rarely explicitly required: for OpenType fonts, the OpenType renderer is used automatically, and for AAT fonts, AAT is chosen by default. Some fonts, however, will contain font tables for multiple rendering technologies, such as the Hiragino Japanese fonts distributed with macOS, and in these cases one over the other may be preferred.

Among some other font features only available through a specific renderer, OpenType provides for the `Script` and `Language` features, which allow different font behaviour for different alphabets and languages; see [Section 2 on page 38](#) for the description of these features. *Because these font features can change which features are able to be selected for the font instance, they are selected by `fontspec` before all others and will automatically and without warning select the OpenType renderer.*

Example 32: X_YTeX's Mapping feature.

```
"¡A small amount of—text!" \fontspec{texgyrepagella-regular.otf}[Mapping=tex-text]
                             ``!`A small amount of---text!'"
```

1.3 Vertical typesetting

X_YTeX provides for vertical typesetting simply with the ability to rotate the individual glyphs as a font is used for typesetting:

```
\def\verttext{  }
\fontspec{Hiragino Mincho Pro}
\verttext

\fontspec{Hiragino Mincho Pro}[Renderer=AAT,Vertical=RotatedGlyphs]
\rotatebox{-90}{\verttext}% requires the graphicx package
```

No actual provision is made for typesetting top-to-bottom languages; for an example of how to do this, see the vertical Chinese example provided in the X_YTeX documentation.

2 The Graphite renderer

Since the Graphite renderer is designed for less common scripts and languages, usually with specific or unique requirements, Graphite features are not standard across fonts.

Currently `fontspec` does not support a convenient interface to select Graphite font features and all selection must be done via ‘raw’ font feature selection.

Here’s an example:

```
\fontspec{Charis SIL}[
  Renderer=Graphite,
  RawFeature={Uppercase Eng alternates=Large eng on baseline}]
D
```

Here’s another:

```
\fontspec{AwamiNastaliq-Regular.ttf}[Renderer=Graphite] ~~~~06b5
\addfontfeature{RawFeature={Lam with V=V over bowl}} ~~~~06b5
```

3 macOS’s AAT fonts

***Warning!** X_YTeX’s implementation on macOS is currently in a state of flux and the information contained below may well be wrong from 2013 onwards. There is a good chance that the features described in this section will not be available any more as X_YTeX’s completes its transition to a cross-platform-only application. All examples in this section have now been removed.*

macOS’s font technology began life before the ubiquitous-OpenType era and revolved around the Apple-invented ‘AAT’ font format. This format had some advantages (and other disadvantages) but it never became widely popular in the font world.

Nonetheless, this is the font format that was first supported by X_YTeX (due to its pedigree on macOS in the first place) and was the first font format supported by `fontspec`. A number of fonts distributed with macOS are still in the AAT format, such as ‘Skia’.

3.1 Ligatures

Ligatures refer to the replacement of two separate characters with a specially drawn glyph for functional or æsthetic reasons. For AAT fonts, you may choose from any combination of Required, Common, Rare (or Discretionary), Logos, Rebus, Diphthong, Squared, AbbrevSquared, and Icelandic.

Some other Apple AAT fonts have those ‘Rare’ ligatures contained in the Icelandic feature. Notice also that the old T_EX trick of splitting up a ligature with an empty brace pair does not work in X_YT_EX; you must use a `\opt kern` or `\hbox` (e.g., `\null`) to split the characters up if you do not want a ligature to be performed (the usual examples for when this might be desired are words like ‘shelffull’).

3.2 Letters

The Letters feature specifies how the letters in the current font will look. For AAT fonts, you may choose from Normal, Uppercase, Lowercase, SmallCaps, and InitialCaps.

3.3 Numbers

The Numbers feature defines how numbers will look in the selected font. For AAT fonts, they may be a combination of Lining or OldStyle and Proportional or Monospaced (the latter is good for tabular material). The synonyms Uppercase and Lowercase are equivalent to Lining and OldStyle, respectively. The differences have been shown previously in [Section 2 on page 22](#).

3.4 Contextuals

This feature refers to glyph substitution that vary by their position; things like contextual swashes are implemented here. The options for AAT fonts are WordInitial, WordFinal, LineInitial, LineFinal, and Inner (sometimes called ‘non-final’). As non-exclusive selectors, like the ligatures, you can turn them off by prefixing their name with No.

3.5 Vertical position

The VerticalPosition feature is used to access things like subscript (Inferior) and superscript (Superior) numbers and letters (and a small amount of punctuation, sometimes). The Ordinal option is (supposed to be) contextually sensitive to only raise characters that appear directly after a number.

The `realscripts` package redefines the `\textsubscript` and `\textsuperscript` commands to use the above font features, including for use in footnote labels.

3.6 Fractions

Many fonts come with the capability to typeset various forms of fractional material. This is accessed in `fontspec` with the Fractions feature, which may be turned On or Off in both AAT and OpenType fonts.

In AAT fonts, the ‘fraction slash’ or solidus character, is to be used to create fractions. When Fractions are turned On, then only pre-drawn fractions will be used.

Using the `Diagonal` option (AAT only), the font will attempt to create the fraction from superscript and subscript characters.

Some (Asian fonts predominantly) also provide for the `Alternate` feature.

3.7 Variants

The `Variant` feature takes a single numerical input for choosing different alphabetic shapes. See [Section 1 on page 66](#) for a way to assign names to variants, which should be done on a per-font basis.

3.8 Alternates

Selection of Alternates again must be done numerically. See [Section 1 on page 66](#) for a way to assign names to alternates, which should be done on a per-font basis.

3.9 Style

The options of the `Style` feature are defined in AAT as one of the following: `Display`, `Engraved`, `IlluminatedCaps`, `Italic`, `Ruby`,¹¹ `TallCaps`, or `Titling`.

Typical examples for these features are shown in [3.1.12](#).

3.10 CJK shape

There have been many standards for how CJK ideographic glyphs are ‘supposed’ to look. Some fonts will contain many alternate glyphs in order to be able to display these glyphs correctly in whichever form is appropriate. Both AAT and OpenType fonts support the following `CJKShape` options: `Traditional`, `Simplified`, `JIS1978`, `JIS1983`, `JIS1990`, and `Expert`. OpenType also supports the `NLC` option.

3.11 Character width

See [3.2.2 on page 52](#) for relevant examples; the features are the same between OpenType and AAT fonts. AAT also allows `CharacterWidth=Default` to return to the original font settings.

3.12 Diacritics

Diacritics are marks, such as the acute accent or the tilde, applied to letters; they usually indicate a change in pronunciation. In Arabic scripts, diacritics are used to indicate vowels. You may either choose to `Show`, `Hide` or `Decompose` them in AAT fonts. The `Hide` option is for scripts such as Arabic which may be displayed either with or without vowel markings. E.g., `\fontspec[Diacritics=Hide]{...}`

Some older fonts distributed with macOS included ‘Ø’ *etc.* as shorthand for writing ‘Ø’ under the label of the `Diacritics` feature. If you come across such fonts, you’ll want to turn this feature off (imagine typing `hello/goodbye` and getting ‘helløgoodbye’ instead!) by decomposing the two characters in the diacritic into the ones you actually

¹¹‘Ruby’ refers to a small optical size, used in Japanese typography for annotations.

want. I recommend using the proper $\text{\LaTeX}$ input conventions for obtaining such characters instead.

3.13 Annotation

Various Asian fonts are equipped with a more extensive range of numbers and numerals in different forms. These are accessed through the `Annotation` feature with the following options: `Off`, `Box`, `RoundedBox`, `Circle`, `BlackCircle`, `Parenthesis`, `Period`, `RomanNumerals`, `Diamond`, `BlackSquare`, `BlackRoundSquare`, and `DoubleCircle`.

Part VIII

Customisation and programming interface

This chapter describes the current interfaces and hooks that use `fontspec` for various macro programming purposes.

1 Defining new features

This package cannot hope to contain every possible font feature. Three commands are provided for selecting font features that are not provided for out of the box. If you are using them a lot, chances are I've left something out, so please let me know.

`\newAATfeature` New AAT features may be created with this command:
 `\newAATfeature{<feature>}{<option>}{<feature code>}{<selector code>}`
Use the XeTeX file `AAT-info.tex` to obtain the code numbers.

```
\newAATfeature{Alternate}{HoeflerSwash}{17}{1}
\fontspec{Hoefler Text Italic}[Alternate=HoeflerSwash]
This is XeTeX by Jonathan Kew.
```

`\newopentypefeature` New OpenType features may be created with this command:
 `\newopentypefeature{<feature>}{<option>}{<feature tag>}`
The synonym `\newICUfeature` is deprecated.
Here's what it would look like in practise:

```
\newopentypefeature{Style}{NoLocalForms}{-loc1}
```

`\newfontfeature` In case the above commands do not accommodate the desired font feature (perhaps a new XeTeX feature that `fontspec` hasn't been updated to support), a command is provided to pass arbitrary input into the font selection string:
 `\newfontfeature{<name>}{<input string>}`

For example, Zapfino used to contain an AAT feature 'Avoid d-collisions'. To access it with this package, you could do some like the following:

```
\newfontfeature{AvoidD} {Special= Avoid d-collisions}
\newfontfeature{NoAvoidD}{Special=!Avoid d-collisions}
\fontspec{Zapfino}[AvoidD,Variant=1]
sockdolager rubdown                    \
\fontspec{Zapfino}[NoAvoidD,Variant=1]
sockdolager rubdown
```

The advantage to using the `\newAATfeature` and `\newopentypefeature` commands instead of `\newfontfeature` is that they check if the selected font actually contains the desired font feature at load time. By contrast, `\newfontfeature` will not give a warning for improper input.

2 Defining new scripts and languages

`\newfontscript` While the scripts and languages listed in [Table 2](#) and [Table 3](#) are intended to be comprehensive, there may be some missing; alternatively, you might wish to use different names to access scripts/languages that are already listed. Adding scripts and languages can be performed with the `\newfontscript` and `\newfontlanguage` commands. For example,

```
\newfontscript{Arabic}{arab}
\newfontlanguage{Zulu}{ZUL}
```

The first argument is the `fontspec` name, the second the OpenType tag. The advantage to using these commands rather than `\newfontfeature` (see [Section 1 on the previous page](#)) is the error-checking that is performed when the script or language is requested.

Both commands accept a comma-separated list of OpenType tags in order of preference. This permits, for example, supporting both new and old versions of a language tag with a common user interface:

```
\newfontlanguage{Turkish}{TRK,TUR}
```

Here, a font that is requested with `Script=Turkish` will first be checked for the OpenType language tag `TRK`, which will be selected if available. If not available, the `TUR` tag will be queried and used if possible as a fallback.

3 Going behind `fontspec`'s back

Expert users may wish not to use `fontspec`'s feature handling at all, while still taking advantage of its $\text{\LaTeX}$ font selection conveniences. The `RawFeature` font feature allows font feature selection using a literal feature selection string if you happen to have the OpenType feature tag memorised. More importantly, this can be used to enable features for which `fontspec` does not yet have a user interface to.

Multiple features can either be included in a single declaration:

```
[RawFeature=+smcp;+onum]
```

or with multiple declarations:

```
[RawFeature=+smcp, RawFeature=+onum]
```

Note that there is no error-checking when using `RawFeature`. Where a `fontspec` interface exists to a feature it is generally better to use it. If the font lacks the feature or if it would clash with another feature, `fontspec` will attempt to warn and/or resolve the issues.

Example 33: Using raw font features directly.

PAGELLA SMALL CAPS	<code>\fontspec{texgyrepagella-regular.otf}[RawFeature=+smcp]</code> Pagella small caps
--------------------	--

4 Renaming existing features & options

`\aliasfontfeature` If you don't like the name of a particular font feature, it may be aliased to another with the `\aliasfontfeature{⟨existing name⟩}{⟨new name⟩}` command, such as shown in Example 34.

Spaces in feature (and option names, see below) *are* allowed. (You may have noticed this already in the lists of OpenType scripts and languages).

`\aliasfontfeatureoption` If you wish to change the name of a font feature option, it can be aliased to another with the command `\aliasfontfeatureoption{⟨font feature⟩}{⟨existing name⟩}{⟨new name⟩}`, such as shown in Example 35.

This example demonstrates an important point: when aliasing the feature options, the *original* feature name must be used when declaring to which feature the option belongs.

Only feature options that exist as sets of fixed strings may be altered in this way. That is, Proportional can be aliased to Prop in the Letters feature, but 550099BB cannot be substituted for Purple in a Color specification. For this type of thing, the `\newfontfeature` command should be used to declare a new, *e.g.*, PurpleColor feature:

```
\newfontfeature{PurpleColor}{color=550099BB}
```

Except that this example was written before support for named colours was implemented. But you get the idea.

5 Programming interface

5.1 Variables

`\l_fontspec_family_tl` In some cases, it is useful to know what the L^AT_EX font family of a specific fontspec font is. After a `\fontspec`-like command, this is stored inside the `\l_fontspec_family_tl` macro. Otherwise, L^AT_EX's own `\f@family` macro can be useful here, too. The raw T_EX font that is defined from the 'base' font in the family is stored in `\l_fontspec_font`.

`\g_fontspec_encoding_tl` Package authors who need to load fonts with legacy L^AT_EX NFSS commands may also need to know what the default font encoding is. Since this has changed from EU1/EU2 to TU, it is best to use the variable `\g_fontspec_encoding_tl` instead.

Example 34: Renaming font features.

<pre> \aliasfontfeature{ItalicFont}{IF} \aliasfontfeature{ItalicFeatures}{IFF} \setmainfont{ EBGaramond-Regular.otf }[IF = EBGaramond-Italic.otf , IFF = {Style=Swash} ,] Roman Letters <i>And Swash</i> </pre>	<pre> \aliasfontfeature{ItalicFont}{IF} \aliasfontfeature{ItalicFeatures}{IFF} \setmainfont{ EBGaramond-Regular.otf }[IF = EBGaramond-Italic.otf , IFF = {Style=Swash} ,] Roman Letters \itshape And Swash </pre>
---	---

Example 35: Renaming font feature options.

```

\aliasfontfeature{VerticalPosition}{Vert Pos}
\aliasfontfeatureoption{VerticalPosition}{ScientificInferior}{Sci Inf}
\fontspec{LinLibertine_R.otf}[Vert Pos=Sci Inf]
Scientific Inferior: 12345    Scientific Inferior: 12345

```

5.2 Functions for loading new fonts and families

```

\fontspec_gset_family:Nnn #1 : LATEX family
\fontspec_set_family:Nnn #2 : fontspec features
                          #3 : font name

```

Defines a new NFSS family from given $\langle features \rangle$ and $\langle font \rangle$, and stores the family name in the variable $\langle family \rangle$. This font family can then be selected with standard L^AT_EX commands $\fontfamily{\langle family \rangle}\selectfont$. See the standard fontspec user commands for applications of this function.

```

\fontspec_gset_fontface:NNnn
\fontspec_set_fontface:NNnn

```

```

#4 : primitive font
#5 : LATEX family
#6 : fontspec features
#7 : font name

```

Variant of the above in which the primitive T_EX font command is stored in the variable $\langle primitive font \rangle$. If a family is loaded (with bold and italic shapes) the primitive font command will only select the regular face. This feature is designed for L^AT_EX programmers who need to perform subsequent font-related tests on the $\langle primitive font \rangle$.

5.3 Conditionals

The following functions in expl3 syntax may be used for writing code that interfaces with fontspec-loaded fonts. The following conditionals are all provided in TF, T, and F forms.

5.3.1 Querying font families

`\fontspec_font_if_exist:nTF` `\fontspec_font_if_exist:nTF {<font name/file>} {<true code>} {<false code>}`

Test whether the ‘font name’ (#1) exists or is loadable. `<true code>` or `<false code>` is executed accordingly. The syntax of #1 is a restricted/simplified version of `fontspec`’s usual font loading syntax; fonts to be loaded by filename are detected by the presence of an appropriate extension (`.otf`, etc.), and paths should be included inline. E.g.:

```
\fontspec_font_if_exist:nTF {cmr10}{T}{F}
\fontspec_font_if_exist:nTF {Times~ New~ Roman}{T}{F}
\fontspec_font_if_exist:nTF {texgyrepagella-regular.otf}{T}{F}
\fontspec_font_if_exist:nTF {/Users/will/Library/Fonts/CODE2000.TTF}{T}{F}
```

The synonym `\IfFontExistsTF` is provided for ‘document authors’.

`\fontspec_if_fontspec_font:TF`

Test whether the currently selected font has been loaded by `fontspec`.

`\fontspec_if_opentype:TF` Test whether the currently selected font is an OpenType font. Always true for Lua_{TEX} fonts.

`\fontspec_if_small_caps:TF` Test whether the currently selected font has a ‘small caps’ face to be selected with `\scshape` or similar. Note that testing whether the font has the `Letters=SmallCaps` font feature is sufficient but not necessary for this command to return true, since small caps can also be loaded from separate font files. The logic of this command is complicated by the fact that `fontspec` will merge shapes together (for italic small caps, etc.).

5.3.2 Availability of features

`\fontspec_if_aat_feature:nnTF`

Test whether the currently selected font contains the AAT feature (#1,#2).

`\fontspec_if_feature:nTF` Test whether the currently selected font contains the raw OpenType feature #1. E.g.: `\fontspec_if_feature:nTF {pnum} {True} {False}`. Returns false if the font is not loaded by `fontspec` or is not an OpenType font.

`\fontspec_if_feature:nnnTF` Test whether the currently selected font with raw OpenType script tag #1 and raw OpenType language tag #2 contains the raw OpenType feature tag #3. E.g.: `\fontspec_if_feature:nnnTF {la}`. Returns false if the font is not loaded by `fontspec` or is not an OpenType font.

<code>\fontspec_if_script:nTF</code>	Test whether the currently selected font contains the raw OpenType script #1. E.g.: <code>\fontspec_if_script:nTF {latn} {True} {False}</code> . Returns false if the font is not loaded by fontspec or is not an OpenType font.
--------------------------------------	--

<code>\fontspec_if_language:nTF</code>	Test whether the currently selected font contains the raw OpenType language tag #1. E.g.: <code>\fontspec_if_language:nTF {ROM} {True} {False}</code> . Returns false if the font is not loaded by fontspec or is not an OpenType font.
--	---

<code>\fontspec_if_language:nnTF</code>	Test whether the currently selected font contains the raw OpenType language tag #2 in script #1. E.g.: <code>\fontspec_if_language:nnTF {cyr1} {SRB} {True} {False}</code> . Returns false if the font is not loaded by fontspec or is not an OpenType font.
---	--

5.3.3 Currently selected features

<code>\fontspec_if_current_feature:nTF</code>

Test whether the currently loaded font is using the specified raw OpenType feature tag #1. The tag string #1 should be prefixed with + to query an active feature, and with a - (hyphen) to query a disabled feature.

<code>\fontspec_if_current_script:nTF</code>
--

Test whether the currently loaded font is using the specified raw OpenType script tag #1.

<code>\fontspec_if_current_language:nTF</code>
--

Test whether the currently loaded font is using the specified raw OpenType language tag #1.
