lilyglyphs

Version 0.2.3

Urs Liska

September 29, 2020

Quick Start

lilyglyphs enables LaTeX to include arbitrary notational elements from the LilyPond[1] notation software like characters in continuous text. As it uses `fontspec` to access glyphs from an OpenType font *lilyglyphs* works with XeLaTeX or LuaLaTeX exclusively.

If you have obtained the package as part of a TeX distribution the following Minimal Working Example should get you going in a minute and show if everything works correctly. If you have downloaded the package please refer to section 1.1 on page 6 for a few more preliminary installation steps.

```
\documentclass{article} % Should work with any documentclass
\usepackage{fontspec}   % Is necessary to access OpenType fonts
\usepackage{lilyglyphs} % Currently there are no package options available
\begin{document}
\flatflat{} or \clefGInline. % Predefined commands,
                             % Use {} to allow trailing whitespace
\lilyGlyph[scale=1.2, raise=.5]{timesig.neomensural64}
                             % Access "Emmentaler" glyphs by their name
                             % Generic commands have one mandatory
                             % and one optional (placement) argument.
\twoBeamedQuavers % Commands that include images instead of font glyphs.
\end{document}
```

In the text you can use predefined commands like 𝄫 or 𝄞. Commands without the pair of curly braces swallow any trailing whitespace – use these before punctuations or to combine symbols. For the available predefined commands consult the reference in chapter 3 on page 14.

Glyphs that are part of LilyPond's Emmentaler font but not covered by a predefined command yet can be accessed through the generic command `\lilyGlyph{GLYPHNAME}`, the glyph names being listed in `http://www.lilypond.org/doc/v2.16/Documentation/notation/the-feta-font.html` or in the file `the-feta-font-2-16-2.html` located in the the `documentation` directory of the package. As an example take this neomensural time signature 𝄵. Other symbols can be included as image files ♫, but from the author's perspective this doesn't make a difference.

All commands accept an optional argument containing `key=value` pairs. Currently two keys are supported: `scale` multiplies the size of the glyph by the given factor while `raise` shifts it vertically by the given amount interpreted as *ex*. Glyphs accessed individually generally need these options, see the neomensural time signature in the example.

Apart from the Emmentaler glyphs *lilyglyphs* can include any notational construct that LilyPond can produce. Please refer to section 2.2 on page 9 to learn about the *generic access commands* and to section 5.2 on page 37 for detailed instructions how to create custom commands.

lilyglyphs elements automatically 𝄼 or manually 𝄼 scale with the 𝄼 text size commands.

---

[1] `http://www.lilypond.org`

Contents

1 Introduction

lilyglyphs came into existence when I looked for a way to include arbitrary notational elements in the continuous text of LaTeX documents. Unfortunately all packages I could find were quite restricted in the number of available symbols and/or their flexibility in scaling with the text size. Moreover no solution came near the beauty of the engraving of LilyPond[1]. Therefore I decided to “roll my own” package and make the notation font and notational elements of LilyPond available to LaTeX documents.

LilyPond is a promising competitor in producing the most beautiful musical engraving on the market, and one of the foundations of this beauty is its Emmentaler font. *lilyglyphs* accesses the Emmentaler font through the `fontspec` package and inserts its glyphs in the continuous text applying sophisticated control over scaling and spacing. Therefore it relies on a LaTeX engine supporting `fontspec`. It was written using XeLaTeX, but thanks to the contribution of Dave Bellows[2] it now also works with LuaLaTeX[3].

The Emmentaler font only provides a subset of LilyPond’s notational capabilities, as LilyPond draws many symbols itself. After some experiments with drawing such elements in LaTeX it became clear that this approach isn’t really maintainable – and especially wouldn’t provide a sufficient output quality. Therefore this kind of elements is included through small PDF image files that have explicitely been created with LilyPond.

Great care has been taken to make the notational elements scale well with the surrounding text font size. Each single element can be scaled and vertically shifted, but the settings can also be done per document to accomodate for unusual body fonts that might not match too well with *lilyglyphs*.

The coverage of glyphs is far from being comprehensive, but the package is already very usable for real-world documents. Any Emmentaler glyph that isn’t covered yet by a predefined command can be accessed by its name. Other – image based – commands (for example arbitrarily complex musical expressions) can be added through a process that has been made as simple as possible (although it involves using external software such as LilyPond and possibly Python). Details about extending *lilyglyphs* with additional commands can be found in chapter 5 on page 32.

As the number of possible commands seems endless we would be extremely happy about any contribution of new material. But there are also technical details with regard to LaTeX

---

[1] http://www.lilypond.org

[2] http://www.davebellows.com

[3] Principally it is possible to extend the functionality so it would also work with plain LaTeX by accessing non-OpenType variants of the font, but as the package maintainer has neither experience nor active interest in this area this will only be implemented if there are volunteers who join us.

programming where additional competence would be highly welcome. You may have a look at the issue tracker[4] to get an idea where you could help.

Originally *lilyglyphs* was a project on its own but by now it is part of a rather large family of projects around engraving musical scores with LilyPond and typesetting texts with LaTeX: openLilyLib[5]. If you have come here mainly as a LaTeX document author you should also have a look at the `musicexamples` package[6]. If you are also interested in LilyPond itself you might find the tutorials section, `lilylib` and `lilypond-doc` interesting too. A few more projects are still in the planning phase, and you can always see the current state of affairs at our Github starting page[7]. *lilyglyphs* is developed in the `lilyglyphs` directory of this project site, there you can inspect the source, clone or fork the repository, and submit issue reports. If you are interested in participating in the development of *lilyglyphs*, don't hesitate to contact us directly[8].

1.1 Installation

Installation Options There are several ways to get *lilyglyphs* up and running: As part of a TeX distribution, as a download from CTAN[9] or through its development repository on GitHub[10], which is the recommended way if you're familiar with Git. If it's contained in your distribution you should be able to simply *use* the package and start typing, e. g. with the example given in the "Quick Start" at the beginning of this manual. Otherwise there are a few steps of manual installation.

lilyglyphs is known to need fairly recent versions of `fontspec` and LuaLaTeX. So if you haven't got *lilyglyphs* through your LaTeX distribution you may have to take care that you use a sufficiently current distribution.

Note: If you have installed the package through a distribution you can still use one of the following installation methods because both will actually *hide* the distribution installation. One advantage of this approach is that you can access this manual conveniently using `texdoc lilylgyphs`. But if you want to use the included Python scripts you should be careful to run the correct versions of them.

CTAN Download If you've downloaded the package from CTAN you should extract the archive to a location in the `tex/latex/` directory of your TEXMFHOME (e.g. `~/texmf/` on Linux or `~/Library/texmf/` on Mac OS X).

---

[4] https://github.com/openlilylib/lilyglyphs/issues

[5] http://www.openlilylib.org

[6] http://www.openlilylib.org/musicexamples

[7] https://github.com/openlilylib

[8] mailto:ul@openlilylib.org

[9] http://www.ctan.org/pkg/lilyglyphs

[10] https://github.com/openlilylib/lilyglyphs

GitHub Download If you're familiar with Git the recommended way to use *lily𝄞ly**p**♭s* is to create a fork of the GitHub repository (assuming you already have a Github account). Inside the same `tex/latex/` dir run `git clone git@github.com:YOURUSERNAME/lilyglyphs.git` – please don't provide an alternative destination name because the package should actually reside in a directory called `lilyglyphs/`. Then `cd` into that directory and add the original repository with `git remote add upstream git@github.com:openlilylib/lilyglyphs.git`. From now on you will refer to your fork as `origin` and to the official repository as `upstream`.

The most immediate advantage of using *lily𝄞ly**p**♭s* through the Git repository is that it is the easiest way to keep the package up to date. Each time someone adds new commands to the package they will be instantly available through the repository while regular releases will happen much less frequently. And if updates should somehow break syntax in existing documents you can easily (and temporarily) check out the earlier version of the package you had used when originally writing your documents. Another advantage of the repository version is that you have "direct access" to extending the package with predefined commands in a way that makes it easy for us to incorporate your contribution in the official package.

Font Considerations If you install *lily𝄞ly**p**♭s* manually you will have to make the included Emmentaler OpenType font available for LaTeX. Copy the complete `fonts/` folder from the package directory to `TEXMFHOME/fonts/opentype` (creating these subdirectories if they aren't already present on your system) and rename it to a more explicit name such as 'emmentaler' or 'lilyglyphs'. An even better way would be not to copy the files but rather create a symlink to it with `cd TEXMFHOME/fonts/opentype` and `ln -s /path/to/lilyglyphs/fonts emmentaler` **[TODO: check on Windows]**. That way no manual adjustments have to be made when *lily𝄞ly**p**♭s* is updated and might bring new versions of the font files with it.
Note: the package probably won't work with XeLaTeX if you also have installed Emmentaler as a system font.
Note: If you create a link instead of a copy you have to make sure that there is at least one regular file or subdirectory in the directory beside the link. Otherwise LuaLaTeX won't find the font, which is a limitation in the `kpathsea` library. The easiest way is to simply add an empty `dummy/` subdir.

If you experience any issues with this or other topics during installation please give us feedback so we can improve documentation .

1.2 License

*lily𝄞ly**p**♭s* is distributed under the LaTeX Project Public License version 1.3 or (at your option) any later version of this license. You may find the latest version of this license at `http://www.latex-project.org/lppl.txt`, and version 1.3 or later is part of all distributions of LaTeX version 2005/12/01 or later. A full copy of the license is enclosed in the file `/license/COPYING.LPPL` of the package. The file `/license/MANIFEST` contains a list of all files affected by this licensing.

The package is currently maintained by its original author Urs Liska, but its LPPL status is simply ‘maintained’.

lilyglyphs contains the Emmentaler OpenType font developed provided by the LilyPond project (`http://www.lilypond.org`). It is redistributed unmodified under the SIL Open Font License, Version 1.1. For details see `/otf/LICENSE.OFL` and `/otf/FONTLOG`.

2 Usage

As mentioned in the Quick Start at the beginning of this document you have to activate *lilyglyphs* with `\usepackage{fontspec} \usepackage{lilyglyphs}`. This will give you access to the predefined commands and some generic access commands.

2.1 Usage of Predefined Commands

A number of predefined commands is available for immediate use. To use them you just have to enter the command to print the corresponding musical element, e. g. `\lilyTimeC` for a 𝄴. A complete reference of these commands is available in chapter 3 on page 14, along with any specific comments that might be necessary.

As you will see later there are some commands that take arguments and many that don't. Commands without mandatory arguments behave like e. g. the well-known `\LaTeX` command in that they swallow any whitespace entered after them. So in order to allow a space to be printed afterwards you have to supply a pair of curly braces, like `\flat{} and more` to print "♭ and more". Commands with arguments don't need this special treatment, so you can simply write a space character after them or not like `\lilyDynamics{mf} or \lilyDynamics{pp}.` to achieve ***mf*** or ***pp***.

2.2 Generic Access Commands

There is a wealth of conceivable musical symbols and only a limited number of predefined commands. Even when this package will become more and more comprehensive there will always be cases that haven't been covered yet. For these cases *lilyglyphs* provides four generic access commands:

- `\lilyGlyph`
- `\lilyGlyphByNumber`
- `\lilyText`
- `\lilyImage`.

They all accept one mandatory and one optional argument, the mandatory one being the content to be printed. This content has to be given in a form specific to the respective command.

`\lilyGlyph` expects the OpenType glyph name. You can look up the glyph names in the Appendix of LilyPond's *Notation Reference*[1] or in the somewhat reduced html page provided in

---

[1] http://www.lilypond.org/doc/v2.16/Documentation/notation/the-feta-font

the `/documentation` directory of the package download.
Please note that many Emmentaler glyphs, especially articulations, are aligned to their center because that's what they are used like in a score. So don't be surprise if you need considerable extra space before such glyphs. This isn't a bug but rather a characteristic of using the glyphs in a different context than they were designed for.

`\lilyGlyphByNumber` expects the Unicode code number of the glyph. You will generally not want to use this as the code positions aren't guaranteed to stay the same with new versions of the fonts. There may be some uses for numerical access however, e. g. if you want to iterate over a range of glyphs.

`\lilyText` expects ordinary text as its argument. In fact it just switches the font to Emmentaler and then writes the string given as the argument. This only works for Dynamics letters, numbers and the glyphs + - , . – as these glyphs are located at their ordinary ASCII character position in the font. But you can also enter any spacing commands (like `\hspace` or plain spaces) to control the spacing between glyphs. But keep in mind that this may result in line breaking inside your expression. If you need to prevent this you can surround your expression by an `\mbox`.
(There is a special command provided – `\lilyDynamics` – which is essentially a wrapper around `\lilyText` and presets the character size to a suitable default.)

`\lilyImage` expects the basename of an image file your TeX system can process. It then includes this file using the same optional argument mechanism as the other commands. What sets this command apart from simply including an image is that it automatically scales the image relative to the current text font size, with being printed at its original size at `\normalsize`. You have to take care yourself that LaTeX finds and can handle the image file. While this command has originally been created to print images generated by LilyPond you can actually print *any* image, e. g. scanned images from autographs, taking advantage of *lilyglyphs*' infrastructure, for example the optional arguments described in the following section.

2.3 The Optional Argument: Layout Adjustment

The generic access commands as well as the predefined commands allow an optional argument to be passed. This can contain a list of comma-separated options in `<key=value>` form that influence the appearance of the glyphs. Currently there are the `scale` and `raise` options.

`scale` changes the size of the glyph. As the Emmentaler glyphs are designed for a totally different purpose they often don't fit very well in the context of continuous text. `scale` is given as a factor by which the default size is multiplied. With Emmentaler glyphs this has to be a positive number, otherwise you will get an error. But glyphs printed by `\lilyImage` (or predefined commands based on it) can also be scaled negatively. This results in an image that is rotated around the center of the bottom line of the original. You will therefore have to add an appropriate `raise` value (try e. g. 2 as a starting point). You will have to take some care about the horizontal spacing, as such a flipped image seems to use it with inverted direction. But you can safely put extra horizontal space after the image, and it is a valid and practical way to create two symbols from one image file.

`raise` changes the vertical placement of the glyph. The majority of glyphs is placed too low by default, so they need a positive `raise` value. `raise` is given as a decimal value without units, which is interpreted as *ex*, or x-height. As there is no x in a musical font, this is somewhat arbitrary, but it is a natural unit to scale with the surrounding font size. Usually you may start trying `raise` values between 0 and 0.5.

These layout adjustments can be made at three different stages: at design time (of predefined commands), globally (per document), and at command invocation.

At design time a designer of a predefined command has already selected the optimal default values so the command will work out-of-the-box in most cases.

Globally the layout adjustments default to values that leave the glyphs unaltered (i. e. `scale=1` and `raise=0`). A document author can override these defaults at any time with `\lilyGlobalOptions{<options>}`. This may for example be necessary if you use a text font which doesn’t harmonize well with *lilyglyphs*’ default settings, for example because of its unusual x-height. You can use this command at the beginning of the document to modify the appearance for the whole document, or you can change its settings multiple times throughout the document. This is what you generally have to do when using `\lilyGlyph` for printing individual glyphs from Emmentaler.

At command invocation you can pass the layout adjustment options for the specific instance of the glyph.

The values that finally affect the layout of any given glyphs take all three stages into account. By passing an option at command invocation you don’t set absolute values, but modify the values already present. The effective `scale` value is `designtime * global * invocation`, the effective `raise` is `designtime + global + invocation`. So passing a `scale=1.1` will always slightly increase the glyph’s size, no matter what settings are already in effect.

Technically speaking lilyglyphs *applies a fourth layer of scaling with image files. It calculates a last scaling factor by multiplying the result of the above considerations with the ratio of the current font size versus the size of the* `\normalsize` *font.*

2.4 Example: Define a Custom Command

Now it’s time for an example that actually uses the generic access commands to print symbols provided by the Emmentaler font. It will walk you through the process up to defining a ‘local’ predefined command.

We want to print the *fermata* sign which isn't implemented yet as a predefined command. In the documentation we have looked up the name of the glyph: `scripts.ufermata`, so you can print it with `\lilyGlyph{scripts.ufermata}`:𝄐. While this gives us the right glyph its appearance isn't really what we're after yet and we want to adjust its size and placement. This is done with the optional argument described in the previous subsection.

First we increase the size of the glyph with the `scale` argument. We find that a scaling factor of 1.4 seems suitable: `\lilyGlyph[scale=1.4]{scripts.ufermata}` 𝄐

As you can see the glyph is – as most Emmentaler glyphs are – placed too low, so you have to add the `raise` argument. A value of 0.3 seems fine – remember, the raise argument is interpreted as *ex*, but you don't write down the unit.
`\lilyGlyph[scale=1.4,raise=0.3]{scripts.ufermta}` 𝄐

You can now further see that the glyph is placed too far to the left – which is a good example of the behaviour described earlier with the `\lilyGlyph` command. In fact it seems the point in the middle of the fermata is placed where we would expect the glyph to start. So you have to add some leading space, which might be practical to be entered in *ex*:
`\hspace{1ex}\lilyGlyph[scale=1.4,raise=0.3]{scripts.ufermta}` – 𝄐

If you want you can now simply enclose this definition in a `\newcommand` to be able to reuse it.
`\newcommand{\fermata}{hspace{1ex}\lilyGlyph[scale=1.4,raise=0.3]{scripts.ufermta}`.
After this you can enter your tweaked symbol by simply writing `\fermata`. However you are encouraged to create such a command the way we define predefined commands in *lilyglyphs* itself. This way it will gain the flexibility of the predefined commands, and it will be easier to be incorporated in the package itself. See our instructions on how to create predefined commands the *lilyglyphs* way in section 4.3 on page 27. If you manage to write a command that you find useful for others also please submit it to us – or even better: if you figured out how to create commands in general, please join us. As mentioned earlier the number of possible commands is huge, and the value of the package will increase with each step towards a comprehensive coverage.

2.5 Dotted symbols

There are commands that are accompanied by `\...Dotted` versions. While they can be used like any other commands there are some caveats because they are technically different from normal commands, actually appending a dot to normal commands.

If you use significant scaling factors for the commands you have to check carefully whether the gap and the position of the dot scale and move well. Unfortunately one can't influence the parameters of the dot independently. We have worked hard to enable the designer of a command to create rules how to scale the gap, but you still may run into problems here. In such a case you will have to either change the predefined command in the library or just create the dotted symbol from scratch.

Immediately after having used a dotted symbol you can arbitrarily add more dots with the `\lilyPrintMoreDots` command. This command uses the existing dot settings (scale and raise) and prints another dot. By default it has a gap of 0.25 ex, but you can override this by passing a number as an optional argument, which is interpreted as *ex*.

For example if you take the command `\halfNoteRestDotted`, which prints a dotted half note rest: 𝄼. you can easily add more dots through `\halfNoteRestDotted\lilyPrintMoreDots`: 𝄼..

Please note that you should only call `\lilyPrintMoreDots` immediately after calling a `\...Dotted` command. Otherwise you may get surprising results or even errors because the underlying key-value variables are inititialized wrongly or not at all.

2.6 Optical size

The Emmentaler fonts come in a set of eight “optical sizes”. These are variations of the font originally designed to be used at different point sizes. Generally you can assume that fonts for larger sizes offer more detail and give a somewhat lighter appearance, while fonts for smaller point sizes give more weight on the paper but less detail to be readable at small sizes.

*lily*glyphs gives you the option to access the available font versions, but it may make more sense to appreciate them as “weights” – although this is technically speaking or even conceptionally incorrect. The eight optical sizes of the Emmentaler font are: 11, 13, 14, 16, 18, 20, 23, 26. If you conceive these as weights you would somehow order them from black (11) to light (26). You can switch the used optical size at any time in a document using the command `\lilyOpticalSize`, giving the number as an option. You could for example use this feature to adapt the LilyPond glyphs to a darker or lighter default text font. Be sure to supply a number corresponding to a font actually available on your system. Maybe this will someday also be available as an option to select for a single glyph, but for now you have to switch twice: before and after the glyph.

The optical size used by *lily*glyphs defaults to 16.

Known issues and warnings: Optical sizes don’t work with glyphs printed as images. If you *have* to use these glyphs in different weights, you will have to take care for it yourself. The general plan would be to create different versions of the glyph by creating different glyphs in LilyPond (presumably by using different staff sizes).

3 Reference of Predefined Commands

The following sections document the predefined commands that already have been implemented. They generally contain explanations on the specific use of the commands (if necessary) and a table listing the implemented commands. Remember that any glyph of the Emmentaler font which is not covered by a predefined command yet can be accessed by its name through the `\lilyGlyph` command. A full list of available glyphs is available in the documentation folder of the *lilyglyphs* package or in LilyPond's original documentation at `http://www.lilypond.org/doc/v2.16/Documentation/notation/the-feta-font.html`.

The documentation explicitely mentions if the commands are based on image files.

3.1 Single Notes

Single notes may well be the most frequently used glyphs. Unfortunately they aren't present in EMMENTALER because LilyPond draws them by itself, so *lilyglyphs* realizes them using included pdf image files. The commands are available identically in British and American form. See table 3.1 for the available predefined commands.

Table 3.1: Single Notes

𝅝	`\semibreve` – `\wholeNote`
𝅝.	`\semibreveDotted` – `\wholeNoteDotted`
𝅗𝅥	`\minim` – `\halfNote`
𝅗𝅥	`\minimDown` – `\halfNoteDown`
𝅗𝅥.	`\minimDotted` – `\halfNoteDotted`
𝅗𝅥.	`\minimDottedDown` – `\halfNoteDottedDown`
𝅗𝅥..	`\minimDottedDouble` – `\halfNoteDottedDouble`
𝅗𝅥..	`\minimDottedDoubleDown` – `\halfNoteDottedDoubleDown`
♩	`\crotchet` – `\quarterNote`
♩	`\crotchetDown` – `\quarterNoteDown`
♩.	`\crotchetDotted` – `\quarterNoteDotted`
♩.	`\crotchetDottedDown` – `\quarterNoteDottedDown`
♩..	`\crotchetDottedDouble` – `\quarterNoteDottedDouble`
♩..	`\crotchetDottedDoubleDown` – `\quarterNoteDottedDoubleDown`

𝅘𝅥𝅮	`\quaver – \eighthNote`
𝅘𝅥𝅮	`\quaverDown – \eighthNoteDown`
𝅘𝅥𝅮.	`\quaverDotted – \eighthNoteDotted`
𝅘𝅥𝅮·	`\quaverDottedDown – \eighthNoteDottedDown`
𝅘𝅥𝅮..	`\quaverDottedDouble – \eighthNoteDottedDouble`
𝅘𝅥𝅮··	`\quaverDottedDoubleDown – \eighthNoteDottedDoubleDown`
𝅘𝅥𝅯	`\semiquaver – \sixteenthNote`
𝅘𝅥𝅯	`\semiquaverDown – \sixteenthNoteDown`
𝅘𝅥𝅯.	`\semiquaverDotted – \sixteenthNoteDotted`
𝅘𝅥𝅯·	`\semiquaverDottedDown – \sixteenthNoteDottedDown`
𝅘𝅥𝅯..	`\semiquaverDottedDouble – \sixteenthNoteDottedDouble`
𝅘𝅥𝅯··	`\semiquaverDottedDoubleDown – \sixteenthNoteDottedDoubleDown`
𝅘𝅥𝅰	`\demisemiquaver – \thirtysecondNote`
𝅘𝅥𝅰	`\demisemiquaverDown – \thirtysecondNoteDown`
𝅘𝅥𝅰.	`\demisemiquaverDotted – \thirtysecondNoteDotted`
𝅘𝅥𝅰·	`\demisemiquaverDottedDown – \thirtysecondNoteDottedDown`
𝅘𝅥𝅰..	`\demisemiquaverDottedDouble – \thirtysecondNoteDottedDouble`
𝅘𝅥𝅰··	`\demisemiquaverDottedDoubleDown – \thirtysecondNoteDottedDoubleDown`

3.2 Beamed notes

We will only provide a few complex symbols like beamed notes for default use. Of course one could have the wish for indefinite variations like notes with variable beam slope. But as long as it isn't possible to make this parametrical[1] it is probably a good idea to stick with a few basic commands. For now see table 3.2 for the implemented commands.

Beamed notes are implemented using PDF files.

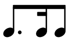
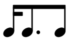
Table 3.2: Two Beamed Notes

♫	`\twoBeamedQuavers`

---

[1] See (GitHub-Issue #64)

Table 3.3: Three Beamed Notes

	`\threeBeamedQuavers`	Three beamed quavers
	`\threeBeamedQuaversI`	Second dotted
	`\threeBeamedQuaversII`	First dotted
	`\threeBeamedQuaversIII`	Second dotted, first short

3.3 Clefs

Some of the clef glyphs are among the few that are too large by default. You couldn't use a G clef at default size within continuous text without severely 𝄞 damaging line spacing. But if you scale them to a size that doesn't disturb line spacing, they look quite disproportionate, especially when combined with other elements: 𝄡♮. To ease the handling of that issue we provide the clefs in two forms, at ordinary size which can be problematic in continuous text, and as an -Inline version which looks somewhat funny but can be used within the line.

See table 3.4 for the available predefined commands.

Table 3.4: Clefs

𝄞	`\clefG`, `\clefGInline`	clefs.G
𝄢	`\clefF`, `\clefFInline`	clefs.F
𝄡	`\clefC`, `\clefCInline`	clefs.C

3.4 Time Signatures

The Emmentaler font provides two "real" glyphs for time signatures, the 𝄴 and the 𝄵.

The numerical (single and compound) time signatures can be printed using `\lilyTimeSignature{numerator}{deno` $\frac{4}{4}$. `numerator` and `denominator` are treated as `\lilyText`, so you can enter anything this command can use (see section 2.2 on page 9). This way you can easily write compound time signatures like `\lilyTimeSignature{4 + 7}{8}`: $\frac{4+7}{8}$. But be aware that the command does *not* have a notion of columns, so you have to take care about the horizontal alignment yourself if there are more than one item in both rows, for example by adding explicit space.

`\lilyTimeSignature` respects any whitespace after the closing bracket so you don't have to supply the pair of curly braces.

Table 3.5: Time Signatures

𝄴	`\lilyTimeC`	timesig.C44
𝄵	`\lilyTimeCHalf`	timesig.C22
$\frac{7}{8}$	`\lilyTimeSignature{7}{8}`	
$\frac{3+4}{4+8}$	`\lilyTimeSignature{3 + 4}{4 + 8}`	

Known issues and warnings: `\lilyTimeSignature` also expects the optional argument as the other commands, but it doesn't understand the `raise` option correctly. The box with the time signature is vertically centered so it should generally be OK, but if you for some reason have to change its vertical position you should manually surround the whole command by a `\raisebox`.

3.5 Numbers

Numbers can be entered with the already known `\lilyText` command. Access through the glyph names is possible but not necessary. Therefore we don't provide predefined commands for them. With the default scaling of 1.0 they generally fit as lowercase letters like **0123456789** `\lilyText{0 1 2 3 4 5 6 7 8 9}`. For Uppercase letters you can start trying a scaling of 1.3. A future version of the package may provide convenience commands with default scalings for upper/lowercase letters, fingerings, figured bass numbers, time signature numbers etc.

A special case are four glyphs that are related to numbers: **+-.,** (plus, hyphen, fullstop and comma). These are also accessible through `\lilyText` and their respective characters, the example in the previous sentence being written as `\lilyText[scale=1.5]{+ - . ,}`.

3.6 Accidentals

The `\natural` ♮, the `\flat` ♭ and the `\sharp` ♯ replace the respective commands from standard LaTeX. Please note that all the accidentals are designed at the same scaling in order to allow a uniform appearance. You will however have to check if they don't affect an even line spacing.

See table 3.6 on the next page for the list of implemented commands.

Table 3.6: Accidentals

♮	`\natural`	accidentals.natural
♯	`\sharp`	accidentals.sharp
♯↑	`\sharpArrowup`	accidentals.sharp.arrowup
♯↓	`\sharpArrowdown`	accidentals.sharp.arrowdown
♯↕	`\sharpArrowboth`	accidentals.sharp.arrowboth
𝄲	`\sharpSlashslashStem`	accidentals.sharp.slashslash.stem
♯	`\sharpSlashslashslashStemstem`	accidentals.sharp.slashslashslash.stemstem
𝄲	`\sharpSlashslashslashStem`	accidentals.sharp.slashslashslash.stem
♯	`\sharpSlashslashStemstemstem`	accidentals.sharp.slashslash.stemstemstem
𝄪	`\doublesharp`	accidentals.doublesharp
♭	`\flat`	accidentals.flat
𝄫	`\flatflat`	accidentals.flatflat

3.7 Rests

See table 3.7 for the implemented rest commands.

For more information on how to use `\lilyPrintMoreDots` to produce multiply dotted rests please see section 2.5 on page 12.

Table 3.7: Rests

𝄻	`\wholeNoteRest`	Whole Note Rest
𝄻.	`\wholeNoteRestDotted`	DottedWhole Note Rest
𝄼	`\halfNoteRest`	Half Note Rest
𝄼.	`\halfNoteRestDotted`	Dotted Half Note Rest
𝄼..	`\halfNoteRestDotted\lilyPrintMoreDots`	Example of Double Dotted Rest
𝄽	`\crotchetRest`	Crotchet Rest
𝄽.	`\crotchetRestDotted`	Dotted Crotchet Rest
𝄾	`\quaverRest`	Quaver Rest

𝄾.	`\quaverRestDotted`	Dotted Quaver Rest
𝄿	`\semiquaverRest`	Semiquaver Rest
𝄿.	`\semiquaverRestDotted`	Dotted Semiquaver Rest

3.8 Dynamic Text

As explained earlier the Dynamic Letters can be accessed through `\lilyText` without providing glyph names or numbers as argument. For the available letters see 3.8. As a convenience there is a predefined command `\lilyDynamics`, which is just a wrapper around `\lilyText` that sets the `Scale` argument to a default value of 1.5.

Table 3.8: Single Dynamics Letters

f	`\lilyDynamics{f}`	forte
p	`\lilyDynamics{p}`	piano
m	`\lilyDynamics{m}`	mezzo-
r	`\lilyDynamics{r}`	rin-
s	`\lilyDynamics{s}`	s-
z	`\lilyDynamics{z}`	-z

These Letters can be combined to make complex Dynamics. *lilyglyphs* doesn't provide predefined commands as they can easily be entered as single strings to `\lilyDynamics`, like `\lilyDynamics{sffzrmp}`, resulting in ***sffzrmp***. In this specific situation you could enter a small horizontal space between the ***z*** and the ***r*** – but as this combination wouldn't occur in real life we don't need to demonstrate it here. There are a few predefined commands (see table 3.9) handling the "kerning" of some special combination of letters. Internally these commands internally use `\lilyDynamics` with its default scaling.

Table 3.9: Combined Dynamics Expressions

rf	`\lilyRF`	rinforzando
rfz	`\lilyRFZ`	rinforzando (alternative)

3.9 Graphical Dynamic Symbols

Graphical dynamic symbols like hairpins are realized by including image files. See table 3.10 for the implemented commands.

Table 3.10: Dynamics Signs

<	`\crescHairpin`
>	`\decrescHairpin`

3.10 Articulations

Table 3.11: Articulations

>	`\lilyAccent`	
<>	`\lilyEspressivo`	
·	`\lilyStaccato`	
ǫ	`\lilyThumb`	Thumb pizzicato
^	`\marcato`	
v	`\marcatoDown`	
∸	`\portato`	
⨪	`\portatoDown`	
ˈ	`\staccatissimo`	
–	`\tenuto`	

3.11 Scripts

Script implementation has just begun. For the implemented glyphs see table 3.12.

If you manually enter scripts through `\lilyGlyph` you will notice that they usually seem to print too far to the left, clashing with the preceding text. This is due to the fact that in musical engraving scripts are centered relative to the note they belong to. Therefore you often have to add extra space before the glyph if you access them directly. The predefined commands should of course have this already built in.

Table 3.12: Scripts

𝄐	`\fermata`	Fermata

3.12 Accordion Notation

Table 3.13: Accordion notation

⌸	`\accordionBayanBass`	Bayan bass register
⊖	`\accordionDiscant`	Discant register
⊖	`\accordionFreeBass`	Free bass register
❃	`\accordionOldEE`	Unknown accordion notation
˥	`\accordionPull`	Directon of bellows
〉	`\accordionPush`	Directon of bellows
⊜	`\accordionStdBass`	Standard bass register

These are the symbols present in the Emmentaler font. More symbols that are created through combination of the discant symbol with one or more accordion dot(s) have yet to be created.

3.13 Fancy (Example) Commands

This is just an example of a fancy notation generated with the assistance of our scripts (see section 5.2 on page 37).

Table 3.14: Fancy (Example) Commands

	`\lilyFancyExample`	a fancy command using a LilyPond file

4 Internals

4.1 Documentation of the generic access commands

This section is essential for readers who want to understand how this package works internally, for example if they want to actively participate in its development. It is structured from the perspective of the package's behaviour instead of from a user's POV. If you simply want to add your own predefined commands "the *lilyglyphs* way" it is a good idea to read this section too, but you may also skip it and directly go to 4.2.

In order to make the package's `.sty` file easier to understand, its content is split into multiple input files which are located in the `/commands` and `/core` subfolders. The most fundamental definitions are in the `core/keyval.inp` and `core/genericAccess.inp` files.

4.1.1 Accessing Emmentaler Glyphs

The command that actually prints glyphs from the Emmentaler font is `\lilyPrint`, defined in `core/genericAccess.inp`. It isn't intended to be called directly within a document, but only from the predefined commands. It takes two arguments, the first – optional – being the comma-separated list of `<key=value>` pairs, the second the actual content to be printed.

```
\newcommand*{\lilyPrint}[2][]{%
    \interpretLilyOptions{#1}%
    \raisebox{{\lilyEffectiveRaise}ex}{%
        {\fontspec[Scale=\lilyEffectiveScale]
        {emmentaler-\lilyOpticalSuffix.otf}#2}%
    }%
}
```

At first the command `\interpretLilyOptions` is called, where the options of the different levels are evaluated and calculated to their effective values. Then the content of #2 is printed, within a `\raisebox` and with the currently selected opticals version of the Emmentaler font.

```
\newcommand*{\interpretLilyOptions}[1]{%
    \setkeys{lilyCmdOptions}{scale=1,raise=0}%
    \setkeys{lilyCmdOptions}{#1}%
    \pgfmathsetmacro{\lilyEffectiveScale}{%
        \lilyGlobalOptions@scale *
        \lilyCmdOptions@scale * \lilyDesignOptions@scale}%
    \pgfmathsetmacro{\lilyEffectiveRaise}{%
        \lilyGlobalOptions@raise +
        \lilyCmdOptions@raise + \lilyDesignOptions@raise}%
}
```

`\interpretLilyOptions` is defined in `core/keyval.inp`.
The `<key=value>` mechanism is achieved using the `keyval` package as the most basic solution available. If this can be implemented in a more elegant, extensible and/or powerful way using other packages, e. g. `pgfkeys`, we'd appreciate any input. It uses three families of keys, corresponding to the three levels of options: `lilyGlobalOptions`, `lilyDesignOptions` and `lilyCmdOptions`.

In a first step the keys for the actual command options are initialized to a neutral state. This is necessary because otherwise options not present in the command invocation would be in an uninitialized or unknown state. After this the options provided by the command invocation (i. e. the ones in the end user's document) are applied. Finally the effective values of the options are calculated from the global, the design and the command invocation options. The scaling values are multiplied, the raise values added. While the command options have just been determined, the global options are valid globally (and can be changed globally) and the design options have been set by the command that actually called `\lilyPrint`. This is the reason why `\lilyPrint` should never be invoked directly – the design options would be in the unknown state of the previous invocation of`\lilyPrint`.

At the next higher level there are the three generic access functions `\lilyGlyph`, `\lilyGlyphByNumber` and `\lilyText`, defined in `core/genericAccess.inp`. They are very similar and differ only in the way they determine the actual content to be printed. As stated in the end user part of this documentation they expect two arguments, the optional `<key=value>` pair list and the contents. As a first step the commands initialize the design options to a neutral state, because the "design" of the generic glyphs has to be neutral by design. In the second step they invoke `\lilyPrint`, passing the optional argument along and determine the printed content individually: `\lilyGlyph` calls the helper function `\lilyGetGlyph`, `\lilyGlyphByNumber` calls `\lilyGetGlyphByNumber`, while `\lilyText` just passes its contents argument unchanged to `\lilyPrint`.

These helper functions are important because most predefined commands call one of them to select glyphs from the Emmentaler fonts.
`\lilyGetGlyph` takes the glyph name as found in the LilyPond documentation.
`\lilyGetGlyphByNumber` takes the Unicode character index of the intended glyph. But be aware that the Unicode index may change at any time with new versions of the Emmentaler font, so it usually isn't a good idea to access glyphs through their index. There may be some uses for numerical access, however, e. g. to iterate over a range of glyphs.

4.1.2 Printing image files

`\lilyPrintImage`, defined in `core/genericAccess.inp`, is the command that prints glyphs from a supplied image file. It actually is quite similar to `\lilyPrint`, with only the extra consideration of scaling the image to the text font size.

```
\newcommand*{\lilyPrintImage}[2][]{%
    % interpret optional argument
    \interpretLilyOptions{#1}%
    % determine scaling factor to accomodate the current font size
```

```
    % (as images don't scale automatically with the font)
    \lilyScaleImage%
    % Print the image in a raisebox
    \raisebox{{\lilyEffectiveRaise}ex}{%
        \includegraphics[scale=\lilyImageEffectiveScale]{#2}%
    }%
}
```

First the command calls `\interpretLilyOptions`, which is the same as with `\lilyPrint`. But as an additional step `\lilyScaleImage` is called, and finally it uses `\lilyImageEffectiveScale` as the scaling factor instead of `\lilyEffectiveScale`. I won't explain this in detail, but in effect it multiplies the `\lilyEffectiveScale` calculated before with the ratio of the current font size to the `\normalsize` size.

What is finally given as the content to be printed is the basename of an image file. This can be any file format understood by the used TeXengine but we highly recommend using PDF files for sake of printing quality.

As with printing Emmentaler glyphs there is no handling of design time options here, and for that reason you should never call this command directly from a document. Please use `\lilyImage` instead which does the same as `\lilyPrintImage` but additionally defaults the design time options to neutral values.

4.2 The Package's Directory Structure

This section describes the directory structure as found in the development repository on GitHub[1]. If you have obtained *lilyglyphs* from there and want to write new commands on a regular basis, especially if you want to contribute your results back to the package, you will need to have a fair understanding of it. The same is true if you want to use the Python scripts as described in chapter 5 on page 32. If you intend to contribute or just start doing so it is highly recommended to go that way.

If you have downloaded *lilyglyphs* from CTAN there will be significant differences in the directory structure which are mentioned below, so you should read the section anyway.

If you are using *lilyglyphs* from a TeX distribution you can't really consider the contents of this section because the files will be scattered over a number of system and distribution specific places which are out of our control. There is a way around this problem which is described in chapter 5 on page 32, but we still recommend using the GitHub version for serious extending work. Getting an idea about the package content is a good idea nevertheless.

The root directory contains the usual number of files such as README, installation hints and licensing information as well as the main package file `lilyglyphs.sty`.

---

[1] https://github.com/openlilylib/lilyglyphs

`/core`[2] consists of include files for `lilyglyphs.sty` that contain the fundamental program logic of the package.

The `/commands` directory contains files defining the predefined commands available to the end user of *lilyglyphs*. If you are going to write new commands you will include them in these files or add new files here.

`/documentation` contains this manual and an example document, boxth as `.tex` and PDF files as well as an HTML document listing all glyphs of the EMMENTALER font.

The subdirectory `resources` contains included files for the LaTeX document as well as helper packages for the LaTeX documents. The subdirectory `lilyglyphs_logo` contains the logo of the package in PDF and PNG format along with its `.tex` source.

`/fonts` contains the EMMENTALER font files, copied from LilyPond 2.16.2 as the latest stable version. This is the directory you will have copied or linked to during a manual installation of the package.

`/glyphimages` contains all material for the image based commands: original input files, generated LilyPond source files and the resulting PDF images. If you are going to create new commands using LilyPond generated images you will work in this directory. This is explained in more detail in section 5.2 on page 37.

`/license` contains the full LPPL license for the package and the OFL license for the included Emmentaler fonts.

`/scripts` contains Python scripts that simplify the generation and management of new commands. In the CTAN archive the contents of this directory is distributed over two directories, `/bin` and `/lib`.

`source` contains the METAFONT sources for the Emmentaler fonts. They are checked out from the LilyPond development repository at exactly the same state as the binary font files used.

4.2.1 "Private" Directory Structure

If you have obtained *lilyglyphs* any other way than through its Git repository you probably can't or don't want to modify the package files directly. Instead we recommend you set up a private "shadow" search path where you can safely place your own additions. Apart from potential issues with write access to files in the TeX installation this procedure will avoid your changes being overwritten by package updates.

---

[2] For this section a leading "/" in directory names refer to the package root directory.

We have provided a skeleton directory structure in the file `lilyglyphs_private.zip` which is located in the `documentation/` directory of the package[3]. Extract this archive somehwere in your LaTeX search path, e. g. to `TEXMFHOME/tex/latex/lilyglyphs_private`.

In the first level of this directory you'll find two nearly empty stubs: `lilyglyphsPrivate.sty` and `lilyglyphsPrivate.tex`. The first file is a LaTeX package where you can store and organize your own predefined commands[4], the second a file where you can (and are strongly advised to) document your additions. If you make use of these files in a structured way (following the hints in the comments in these files) it will be very easy for us to incorporate any additions you send to us by email (if you don't want to go the Git way).

The rest is a copy of the directory structure below `/glyphimages`. There you can store sources and results of commands generated with LilyPond. Please adhere to the model of the directories in the package for this. And please also see the chapter chapter 6 on page 42 on contributing.

4.3 How to write predefined commands

Writing your own predefined commands is actually quite straightforward – and identical if you want to write a command for your document or for inclusion in the package. So if you find yourself creating predefined commands that you think are useful for general use, don't hesitate to submit them to us.

Commands that print single glyphs

Let's review an example of a predefined command, the `\doublesharp`.

```
% "accidentals.doublesharp"
\newcommand*{\doublesharp}[1][]{%
    \setkeys{lilyDesignOptions}{scale=1.5,raise=0.35}%
    \lilyPrint[#1]{\lilyGetGlyph{accidentals.doublesharp}}%
}
```

We use the starred version of `\newcommand`, because a glyph command naturally doesn't span paragraphs. We declare to accept one optional argument, which defaults to empty. This argument can take the list of `<key=value>` options. When writing the commands, please take care not to omit the `%` characters at the line endings, as they prevent unwanted whitespace to be introduced in the output.

In the second line we define the design options for the command. In the example the designer has decided that a doublesharp glyph should be scaled to 1.5 and raised 0.35 ex compared to its default appearance.

The third line calls the internal `\lilyPrint` command. It passes the optional argument, with which the end user can override (i. e. modify) the designed values. As the 𝄪 is a glyph that has

---

[3] Please consult the documentation of your TeX distribution where to find these. A good chance would be to run `kpsewhich lilyglyphs.sty` which should find the location of the main package file

[4] Of course you will have to "use" this package to make the commands available to other documents

to be selected by its glyph name, we call `\lilyGetGlyph`, supplying the glyph name found in the documentation. The result of this command is passed as the `#2` to `\lilyPrint`.

To summarize: Writing a predefined command for printing an Emmentaler glyph involves just two steps, setting the design time options and calling `\lilyPrint` with the appropriate `#2` argument.

If you know the Unicode number of the desired glyph you can call `\lilyGetGlyphByNumber` instead of `\lilyGetGlyph`, but you can't be sure this number will stay the same forever.

Creating commands using image files is practically the same and even simpler. If you look at the definition of a `\crotchet`,

```
\newcommand*{\crotchet}[1][]{%
    \setkeys{lilyDesignOptions}{scale=0.9,raise=-0.2}%
    \lilyPrintImage[#1]{crotchet}%
}
```

you will notice that the only differences are that the actual printing is done with `\lilyPrintImage` instead of `\lilyPrint` and that therefore the basename of the image file can be passed directly. In section 5.2 on page 37 you will see a tool that allows to create numerous image commands quite easily.

As a last example we will look at the definition of `\lilyRFZ` ***rfz***.

```
\newcommand{\lilyRFZ}[1][]{%
    \mbox{%
        \lilyDynamics[#1]{r\hspace{0.035ex}fz}%
    }%
}
```

You may notice that we use `\lilyDynamics` here, one of the Generic Access Commands (see section 2.2 on page 9) instead of the low-level printing command. We can do this and also use the other ones: `\lilyText`, `\lilyGlyph`, `\lilyGlyphByNumber` or `\lilyImage`. This is actually simpler because we don't have to set the design time options – but that's also the main disadvantage: this way we *can't* set them and have to use the given default parameters. As mentioned in section 3.8 on page 19, `\lilyDynamics` is just a wrapper around `\lilyText`, setting the `scale` factor to 1.5. While the other generic commands only print single glyphs, `\lilyText` can print 'plain text', so usually there is no need to write predefined commands only to combine letters to a single command. In some cases this may however be necessary. In the given example of `\lilyRFZ` we need to apply a little bit of extra space between the ***r*** and the ***f***. We see that we can insert a `\hspace` command between the letters without any problems. But as it turns out LaTeX may now decide to insert a line break at its discretion, so we have to additionally enclose this call to `\lilyDynamics` in a `\mbox`. The command just passes the optional argument to `\lilyDynamics`, so you can use these arguments in your document as usual.

This example is meant to encourage you to experiment with the definition of new commands. All you have to deal with is setting the design time options and the optional argument, and chosing the appropriate input method for the second argument. Apart from this you can design commands as you aanre used to.

Once you have defined your new command you of course have to make it available to LaTeX. If it is a “local” command that you just use for a specific document you can simply put it in the document’s preamble or a dedicated helper `.sty` file. If it is an image driven command you will have created the image file before and will have to place this in a location where LaTeX can find it. But the best place you can permanently store your new commands is the `lilyglyphsPrivate` package described in subsection 4.2.1 on page 26. Please don’t forget to document your command in `lilyglyphsPrivate.tex`. For more information on how to contribute your new commands to the package see chapter 6 on page 42.

Create Dotted Symbols

It is not exactly trivial to create dotted symbols as predefined commands. Of course you can always use `\lilyDot` to print LilyPond’s dot glyph, but if you want to create commands that combine a glyph and one ore more dots you encounter two difficulties: You can’t apply the optional arguments independently on the two items, and there may be issues with the scaling and the gap between the two glyphs.

There is some infrastructure in *lilyglyphs* (defined in the file dotted.inp) to faciliate dealing with dotted symbols, but this implementation isn’t completely satisfactory so far. Great care has been taken to hide as much complexity as possible in the mentioned file, in order to make the definition of actual commands as clean and concise as possible.

Let’s analyze the implementation of a dotted half note rest:

```
% Dotted half note rest
\newcommand*{\halfNoteRestDotted}[1][]{%
    % define the optional arguments for the dot
    \setkeys{lilyDesignOptions}{scale=0.8,raise=0.2}%
    % Calculate effective scale/raise and the hspace for the dot
    \lilySetDotOptions[#1]{0.05}{0.5}{0}%
    % Print the rest and then the dot
    \halfNoteRest[#1]\lilyDotSpace\lilyPrintDot
}
```

The command takes the usual optional argument that can contain `<key=value>` pairs. This applies to the dotted symbol as a whole. Please note that in the last line the predefined command `\halfNoteRest[#1]` is called and passed the optional argument. You can only use this technique of creating dotted symbols on top of correctly implemented predefined commands.

The first thing you have to do is to define the DesignTimeOptions for the dot. They are relative to the original design of `\lilyDot`, and you have to adjust them so the dot suits the main glyph in its size and vertical position (you can ignore the horizontal spacing for now):

```
    % define the optional arguments for the dot
    \setkeys{lilyDesignOptions}{scale=0.8,raise=0.2}%
```

You should always set these options because otherwise you might get strange results or error messages.

The next step is to call a quite complex command `\lilySetDotOptions` that sets several options for the dot:

```
% Calculate effective scale/raise and the hspace for the dot
\lilySetDotOptions[#1]{0}{0.5}{0.4}%
```

This command takes one optional and three mandatory arguments. The optional argument is just the one that is written in the LaTeX document and that is passed into the function. The remaining three arguments control the horizontal spacing between the main glyph and the dot. As we now have two individual elements we have to control the gap between them explicitely as it doesn't scale relative to the `scale` argument by itself. The relation between the `scale` factor and the horizontal gap can be understood as a curve (mathematically spoken: a 2nd order function).

The first (mandatory) argument sets the intensity of the curve. A value of 0 will result in a linear relation (no curve at all), that is when doubling `scale` the gap will be exactly twice as wide. Positive values will result in larger gaps for larger **scale**s. As this is a quadratic function you will want to start with very small values or 0.

The second argument sets the general (linear) steepness of the curve. A value of 1 means that by increasing `scale` by 1 the gap will be wider by 1*ex*. 0.5 seems a good starting point for this argument.

The last argument is an offset in *ex* for the whole curve which is independent from the scaling. You can use it to accomodate specifically wide or narrow glyphs.

`\lilySetDotOptions` takes all these informations, calculates some settings for the dot and stores them in internal variables that can be used by subsequent commands. Please understand that they may be partially or totally overwritten by the next use of *any* predefined command. So you have to call this command immediately before actually printing the dot, otherwise it may or may not provide satisfying results.

The final line actually calls three commands:
`\halfNoteRest[#1]` prints the already defined main glyph. `\lilyDotSpace` prints a horizontal space that is determined by the previous call to `\lilySetDotOptions`, and `\lilyPrintDot` finally prints the dot with the settings just defined.

```
% Print the rest and then the dot
\halfNoteRest*[#1]\lilyDotSpace\lilyPrintDot
```

This was an explanation from the perspective of designing new predefined commands. If you want to know how this is implemented internally, please look at the generously commented file `core/dotted.inp`.

Known issues and warning: One issue that hasn't been addressed yet is the vertical placement of the dot when scaled. The dot is positioned relatively to the baseline of the text, and the main glyph may have a different center point. So when scaling the main glyph may seem to behave differently from the dot.

If you want to create a dotted version of a glyph that is printed from an image file it will generally be easier and more reliable to create a command using a new image file.

Create Multiply Dotted Symbols

There is no need to define additional versions of glyphs with more than one dot. For this purpose we have implemented the command `\lilyPrintMoreDots`. This prints a dot with the same characteristics as the preceding one from a dotted command (but remember that there shouldn't be any calls to other predefined commands in betwen). The horizontal gap between the dots scales linearly with a default of 0.25*ex* per unit of `scale`. But if you pass an number as an optional argument this is interpreted as a different gap in *ex*.

5 Generating Commands with Python

As we just have seen in subsection 4.3 on page 27 it is quite easy to create predefined commands. At least with regard to the Emmentaler glyphs it is really straightforward to simply create a command based on existing models and the documentation. Nothing prevents you from simply adding new commands to your document or helper package. Image driven commands are slightly more complex. The LaTeX part is equally straightforward, but you also have to provide the PDF image by creating a score with LilyPond and produce the right output file, which involves a few steps.

Therefore we have developed a set of tools and templates to streamline the process of creating new commands even further. These tools are Python 2 scripts, so in order to use them you will need a working Python installation. They are located in the `/scripts` (GitHub version) or `/bin` (CTAN version) directory of the package. If you want to create image driven commands you will of course have to have LilyPond installed too. In both cases the process basically consists of writing a definitions file (something like a template), calling a script and then putting the results to a useful place.

To summarize your options for extending the symbols coverage:

- If you only have the plain *lilyglyphs* package you can create new commands using Emmentaler glyphs.
  You can also create image driven commands using any preexisting images (e. g. scans or output from any software).
- If you also have LilyPond installed you can additionally create image driven commands that match the ones already present in the package.
- If you have Python working you can make use of our tools to generate Emmentaler and LilyPond commands (LilyPond presence provided).

The Python scripts expect to work inside either the package directory itself or in the private “shadow” directory described in subsection 4.2.1 on page 26. They perform the check based on the “lilygpyhs” part of the directory name, so for the scripts to work correctly it is crucial that you didn’t change their names during a manual installation.

All Python scripts handle `-h`/`--help` and `-v`/`--version` arguments, printing a short usage help or the current *lilyglyphs* version.

Licensing Note All files generated by the Python scripts contain the *lilyglyphs* license preamble stating them to be licensed with the GPL. But this is only relevant if you contribute them back to the package itself. Initially these generated files are the result of *your* work and you can do with them whatever you want. So if you use the Python scripts to create commands for your own needs you can simply remove the license preamble from them.

5.1 Generating Commands for EMMENTALER Glyphs

Generating commands that print Emmentaler glyphs with our tools is a straightforward process. The steps involved are few: 1) create an input definitions file, 2) run a Python script on it, 3) fine-tune the commands in the generated LaTeX file, and 4) move the resulting code to an appropriate place. We'll go through these steps in the following subsections. Be assured that although these instructions span several pages the actual process of creating new commands is very fast once you have got used to it.

In addition to the following documentation you can also refer to a more casual post on the *Scores of Beauty* blog[1] which shows an example of creating a complete group of glyphs.

5.1.1 Preparing the Input File

The first step is to create a text file with entries for any number of new commands to be created. You can save it to any convenient location, and it is up to you to keep this file for reference or drop it after use.

The input file consists of one ore more command definitions. Each command definition is composed of a set of lines with `key=value` pairs (please note that you shouldn't use whitespace around the equals sign). The end of the definition is indicated by an empty line, therefore it is important that your file ends with an empty line, otherwise the last entry will be discarded. Lines beginning with Python and LaTeX style comments (% and #) are ignored, so you can use them to document your file if you want.

A command entry consists of several mandatory or optional lines. The order doesn't matter, but it is considered good practice to stick to one style. The following items are possible/necessary:

- **cmd** (mandatory): Specifies the command name. You have to make sure that it is a valid LaTeX name and that it isn't in use already. *lilyglyphs* prefixes commands that seem prone to ambiguity with "lily" followed by an uppercase letter: instead of `\dynamics` we used `\lilyDynamics`. You are encouraged to adhere to that convention.

- **comment** (optional): You can pass a single line comment that will be used before the command definition. (If you want to have a multiline comment instead you can insert line breaks with `\n`.)

- **element** (mandatory): The actual element to be passed to the internal printing functions. The possible type of its value depends on the `type` of the command, as described below.

- **type** (mandatory, but defaulted): The type determines the internal printing command to be used with the command. Please refer to section 2.2 on page 9 for more information. The

---

[1] `http://lilypondblog.org/2013/09/extending-lilyglyphs-part-1/`

option is mandatory but defaults to “glyphname”, so you can skip it for the most common case that the glyph is called by name.

  - **`glyphname:`** The glyph is selected by its glyphname, which is what you have to specify as “element” (e. g. “accidentals.sharp“ (without the quotes)). You can look up the glyph names in LilyPond’s documentation or in the glyph list contained in the package.
  - **`number:`** The Unicode number is used to determine the glyph.
  - **`text:`** The content is passed as plain text (works only for Dynamic letters, numbers and `+ - , .`)
  - **`dynamics:`** The content is also passed as plain text, but the `\lilyDynamics` function is used to print it (applying the suitable default scaling).
  - **`image:`** The command prints an image file.
    If you already have an image file created with LilyPond or obtained from any other source than LilyPond (e. g. a scanned image from a printed edition or an autograph) this is the recommended way to create image driven commands. []You can even use this to create non-musical commands that print images and profit from *lilyglyphs*’ infrastructure (like the automatic scaling with text size).] For this purpose you can use .pdf files (preferred) or any image files your LaTeX installation can process.

    The “element” for an image driven command is the plain file name without extension or path. The file has to be stored in a location that is visible to LaTeX, and you are responsible for avoiding name clashes. If you have set up a private directory structure as recommended in subsection 4.2.1 on page 26, its `custom_images` subdirectory is a good choice.

- **scale / raise** (optional): If a line contains one of the keys `scale=` or `raise=` the value after the equals sign is used for the design time options of the new command. These values are also kept for subsequent command entries until the file is finished or the script finds a new entry – which would replace its value. The idea behind this option is to simplify the (likely) process of defining a set of related commands within an input file with its high probability of sharing default values.

Here you can see two examples of command entries:

```
cmd=fermataDown
element=scripts.dfermata
comment=downward fermata
# type glyphname is implicitely used

cmd=rinforzando
type=dynamics
element=rfz
comment=Rinforzando (kerned)
```

In the following subsection you can see what these commands are processed to.

5.1.2 Generating the LaTeX code

The Python script that processes your input file is `lily-glyph-commands.py`. It expects the (absolute or relative) filename of the input file as its first and single parameter (apart from the mentioned standard arguments). The program parses the input file and creates a new file with the same basename but a `.tex` extension, in the same folder where the input file is. Please note that the script in its current implementation will silently overwrite any earlier file with that name. So if you need to keep such a generated file for reference you'll have to copy/move it to a safe location.

The resulting file is a working LaTeX document that uses the *lilyglyphs* package. It contains LaTeX `\newcommand` definitions for each command definition in the input file. The visible part of the document contains a reference table (as used throughout this manual) with all generated commands and additionally example text for each generated command.

The examples above would be processed to the following two LaTeX commands:

```
% downward fermata
\newcommand*{\fermataDown}[1][0]{%
    \setkeys{lilyDesignOptions}{scale=1,raise=0}%
    \lilyPrint[#1]{\lilyGetGlyphs{scripts.dfermata}}%
}

% Rinforzando (kerned)
\newcommand*{\rinforzando}[1][]{%
    \setkeys{lilyDesignOptions}{scale=1,raise=0}%
    \lilyDynamics{rfz}}%
}
```

5.1.3 Fine-tuning the LaTeX Commands

So what are we going to do with the contents of this file? Well, it depends on how/where you want to eventually use it, but the first step will be to fine-tune the commands.

The first thing the generated example text does is showing that the command actually works. But its more important purpose is to print the new glyph in different contexts: in continuous text, before punctuations, at the beginning of a line etc. You should use these blocks of example text to adjust the arguments in the `\setkeys{lilyDesignOptions}` clause. The script generates default values for the design time values of the optional argument, and it would of course be purely random if they would be perfect right away. Even if you supplied `scale` and/or `raise` values in the input file you're likely to have to tune the results. You should specifically keep an eye on spacing issues: Does the glyph affect line spacing, is the "kerning" of the glyph correct? On the other hand you should probably try to keep corresponding glyphs at an equal scaling. Besides tweaking the optional argument values you can add space before or after the glyph – keep in mind that you may as well use negative `\hspace`. And of course you can adjust the commands any way you like, but if you are going to make experimental commands you will probably rather write them manually as described in section 4.3 on page 27.

5.1.4 Finishing Off (and Contributing)

If you are satisfied with the new command(s) you will have to move them to a useful place. If you are just creating the commands for your personal or one-time use you can either copy the command definitions to the preamble of your current LaTeX document or to any style file you might maintain. A good choice would be the `lilyglyphsPrivate.sty` as described in subsection 4.2.1 on page 26. In this case it is recommended to supply documentation in the `lilyglyphsPrivate.tex` file on the same directory for your own reference.

But of course we would be happy if you decided to contribute your commands as additions to the package – we consider increasing coverage of glyphs through user contribution a natural way of evolution for *lilyglyphs*. Be prepared for compilation errors when your contribution returns to you as package updates. Once this happens your own command definitions will try to re-define the commands, and LaTeX will throw out error messages. In that case you will have to remove your definitions from your private packages (wherever you have put them) – they are obsolete now anyway.

Probably the simplest way to contribute is sending the processed `.tex` file as an email to the package maintainer (currently `ul@openlilylib.org`). He would then incorporate them into the package. *Please* keep the generated reference table in the file as we will need it to update the command reference in the manual. And please also add some information about your additions in the file: where it belongs, where it should be documented etc. If your commands need any special considerations (e. g. specific arguments) please also add some material suitable for the manual.

If you are working on a fork of the GitHub repository (as is recommended for potential contributors) you can incoporate the new commands directly and send us a pull request (preferably as soon as possible to minimize risk of conflicts with others' changes). This works by completing the following steps:

- Create a new branch with a suitable name.

- Move/copy the command definitions (with all comments) to appropriate .inp files in the package's `/commands` subdirectory. If you find you should create a new .inp file because your commands belong to a new category please take the existing files as a model and add an appropriate `\input` statement to `lilyglyphs.sty`. For any newly created files copy & paste the usual copyright comment at the beginning from another file.

- *Please* add documentation for your commands in the manual (`/documentation/lilyglyphs.tex`). Find the appropriate subsection in the "Predefined commands" section (or create a new one) and add information to it. At least we need the entries in the reference table, but if there is anything special to note about the commands please explain this too. If you add to an existing subsection you may copy the rows from the table in the file generated by the script, otherwise you should copy the whole table (but update the caption and label fields appropriately).

- If you are happy with the result you can push to your fork of the repository and send a pull request through the GitHub web site.

If you have any questions on these procedures it is certainly a good idea to get in contact with us *before* starting any substantial work.

5.2 Generating Commands with Glyph Images

Creating commands with images generated by LilyPond is a significantly more complex task than simply inserting Emmentaler glyphs, and therefore the Python script `lily-image-commands.py` is a more complex one. But from the end user's perspective the process is surprisingly similar: 1) create an input definitions file, 2) run a Python script on it, 3) fine-tune the commands in the generated LaTeX file, and 4) move the resulting code to an appropriate place. The main differences to consider are the different structure of the input files and some considerations about file locations.

But let us start with looking at the form of the input files.

5.2.1 Preparing the Input Files

`lily-image-commands.py` doesn't expect regular LilyPond source files as its input, but rather a file with one or multiple 'snippets' in it, similar to those for `lily-glyph-commands.py`. We will later see that it is possible and makes sense to provide compilable Lilypond files, though.

Depending on the type of your *lilyglyphs* installation you may work either in the `/glyphimages` directory of the package or in the `lilyglyphs_private` directory you set up according to subsection 4.2.1 on page 26. So in the context of the following instructions `/` or `ROOT` refer to the appropriate one of these two directories.

Input definition files are to be stored in the `/definitions` subdirectory . Other than the files used for the generation of Emmentaler commands these files are considered persistent and should remain in place to be able to recreate the images at a later time. Therefore it is recommended to define related commands in one file and give it an appropriate name. A filename extension isn't mandatory but you may use `.ly` as you will see later. The file name will also be reflected in the resulting file with the generated LaTeX commands.

As with the Emmentaler commands the definitions file can contain any number of command entries, although they are structured differently.

A command entry starts with the special key `%%lilyglyphs` on a single line.

After this there may be any number of lines starting with a single % as a LilyPond comment. These comment lines are used for commenting the command in the resulting LaTeX file.

If a line consists of the special key `%%protected` the script skips the entire command. You should use this key whenever you consider a command finished because it will prevent LilyPond from recompiling that command[2] and the script from re-generating the LaTeX command.

---

[2] This is especially important when working inside a Git repository. With any new or different LilyPond installation the generated `pdf` files are likely to be different and would therefore be considered as 'modified' by Git

If a line contains one of the keys `scale=` or `raise=` the value after the equals sign is used for the design time options of the new command. These values are also kept for subsequent command entries until the file is finished or the script finds a new entry – which would replace its value. The idea behind this option is to simplify the (likely) process of defining a set of related commands within an input file with its high probability of sharing default values. The setting of option defaults also works if a command is marked as "protected". Generally this is an advantage since you don't have to take care to correctly update the file when you mark something as protected. But there may be cases where you will want to remove the defaults setting from a `%%protected` clause.

Afterwards you provide the actual LilyPond command in the form of a variable holding a musical expression. The first line should contain exactly the name, the '=' and the opening curly brace. The name of the variable is very important because it will also be used as the file name of the image file and the LaTeX command name. The following lines are interpreted as LilyPond code until the parser finds a line starting with the closing curly brace. This will also end the command entry, but for readability's sake it is a good idea to enter at least one empty line after the entry.

One important thing to know is that by default the generated image will *not* print staff lines, a time signature and the clef.

This is a minimal working example of a *lilyglyphs* entry section:

```
%%lilyglyphs
% crotchet with upward stem
crotchet = {
  g'4
}
```

Warnings: So far the parser isn't very smart. It just *assumes* that the input file it parses is correct. If you provide code with deviations from this explanation, the script will probably produce erroneous results or just stop working. Probably it won't do any harm, but we can't make any promises on that. Please consider the script as being in a **very experimental** state.

In the current implementation you can't create commands that *do* print staff lines, time signatures or clefs. To achieve this you'd have to write the LilyPond input file(s) on your own. This issue is on the TODO list for one of the next releases of *lilyglyphs*.

You can define as many entries in one file as you like – they will all be processed at once. Of course it is recommended to combine a coherent set of commands.

Everything that is outside of the lilyglyph entries is ignored by the script, so you can make use of these places to make a usable LilyPond file out of it. You can start off with the file `/definitions/_template.ly` to see how it works.

Basically you have to copy your definition to a new variable named `symbol` and then include a special include file `score.ily` (which is provided in that `/definitions` directory). This will print your command definition as the only element in a new score block that is prepared not to print staff, time signature and clef. You can repeat this several times, inserting top-level

`\markup`s and have LilyPond produce a sheet with all defined symbols. This way you can finish your design in “pure LilyPond” before actually going on to the LaTeX part – and at the same time prepare a reference sheet of your work.

```
\version "2.16.2"

%%lilyglyphs
% crotchet with upward stem
crotchet = {
  g'4
}
\markup "Crotchet with upward stem"
symbol = \crotchet
\include "score.ily"

% %lilyglyphs
% crotchet with downward stem
crotchetDown = {
  \stemDown
  c'4
}
\markup "Crotchet with downward stem"
symbol = \crotchetDown
\include "score.ily"

% and so on ...
```

5.2.2 Running the Script

If you have prepared your input file you can run the `lily-image-commands.py` script, passing it the complete file name (as a relative or absolute path) of your definitions file. The script tries to check if the input file is in the correct location – which is considered true if it is in a `defintions` folder and has a `lilyglyphs*` directory in its path. This is valid both within the package directory and the private directory structure.

Now the script will do several things:

Generate LilyPond source files: In a first step the program parses the input file and extracts information about the command definitions that haven’t been marked as protected. For each command definition it generates one compilable LilyPond source file in the `/generated_src` directory, named with the command name. If there is such a file already (i. e. the command had already been processed before) you will be asked if you want to overwrite this file. If you are currently working on a command you will probably want the file to be overwritten, but if you *accidentally* used a name that already exists you should of course keep the existing files and rename your new command.

Compile images with LilyPond: Now these generated source files are compiled using LilyPond with the command line option `-dpreview`. The directory is then cleaned up, and the resulting small PDF files are moved to the `/pdfs` directory.

Generate LaTeX commands: Finally the script generates a LaTeX file that is comparable to the one generated by the Emmentaler command generation script. It will be stored in the `/generated_cmd` directory as a `.tex` file with the same basename as your original input file. Please note that these generated files aren't considered persistent and will be overwritten if you repeatedly process the same input file.

5.2.3 Utilizing the results

If you have successfully run the script you have the following results:

- LilyPond source files in the `/generated_src` folder (one for each command),
- corresponding PDF files in the `/pdfs` folder and
- the file `/generated_cmd/INPUTFILENAME.tex`.

You shouldn't touch the first two items but go on and open the `.tex` file. This file is structured identically as the files generated by `lily-glyph-commands.py`, and also the way to finish off and possibly contribute your commands is essentially the same as described in subsection 5.1.3 on page 35 and subsection 5.1.4 on page 36.

There are a few differences however:

- You should always keep the original definitions file so you can later re-generate the image from it. This can happen when you need to modify it – or when you want to upgrade the package to a new version of EMMENTALER or LilyPond.
- Don't forget to mark finished command definitions as 'protected'.
- Keep in mind that the generated `.tex` file will be overwritten next time the same input file is processed by the script, so if you need to make any changes to the LaTeX command definitions you should immediately copy them to a safe place.
- Sometimes you will find that a rather tall glyph is difficult to accomodate because the necessary scaling will make it look unnaturally small. In such a case you might consider going back to the LilyPond side to review the design of your glyph (for example the glyphs for the single note commands have shortened stems). Please make sure that you understand the implications of the following section about partial processing.
- If you want to submit new commands by email we need 1) the original definition, 2) the generated LilyPond source, and 3) the final LaTeX command.

5.2.4 Partial processing of the input file

You are encouraged to put a coherent set of multiple command definitions in one definitions file and keep this file, as it is the source from that everything can be rebuilt at any time. But there will be occasions when you don't want the Python script to do all the work over and over again. If you add a new command to the input file (and you *should* add it to an existing file if it belongs to the same category) you only want the new command to be processed. Or if you are working on the fine-tuning of the commands (as described in the previous section) and decide that you have to change the LilyPond definition of a single glyph you also only want to reprocess this one. For this purpose you can mark entries in the input definitions file as "protected", in order to prevent them to be newly processed by the script. To do this you enter a line containing `%%protected` in your entry definition (note the double percent sign and the absence of a `space` after them). You can see an example for this in the `_template.ly` file – because the example entry shouldn't be processed at all. From now on the marked command won't be processed anymore. This means that the LilyPond source file won't be generated again, LilyPond isn't run again for the script (which would be the most annoying thing), and the LaTeX commands won't be generated again. The output file will of course be overwritten (you have renamed it before re-running the script, didn't you?) but it won't be cluttered with commands that you have already dealt with earlier.

5.2.5 Recreating Image Files

There also are occasions when you might have to (re)create the PDF files that serve as the glyph images without having to regenerate the LaTeX commands or the LilyPond source files. One example would be the upgrade to a new an EMMENTALER version. In that case you can run the script `rebuild-pdfs.py`. In order to work the script expects the current working directory to be either the `glyphimages` subdirectory of the package or the root of the `lilyglyphs_private` directory. This script essentially compares the directories with the generated LilyPond sources and with the created pdf files, and if it finds a source without a corresponding pdf it will call LilyPond to recreate it. If you don't want to regenerate *missing* PDFs but rather replace *existing* files you first have to delete the *pdf* files on disk so the script can detect them as missing. Of course this script will only work with a working LilyPond installation.

Maybe in the future there will be functionality to detect changes in the glyphs' definitions, or remove PDF files that don't have a corresponding source file anymore.

6 Contributing to *lilyglyphs*

The original motivation to create *lilyglyphs* was the need to include musical symbols in a critical report of a scholarly edition. Looking around on the internet and CTAN I didn't find a solution that was even near the flexibility and output quality I wanted for this edition (and my work in general), so I came up with the idea of accessing the glyphs of LilyPond's Emmentaler font. Fortunately I realized very early that I should make all this more generic and create a public LaTeX package. When I found the solution to incorporate non-Emmentaler glyphs through LilyPond-generated PDF images I saw that it worked out quite well. By now I have a solution that completely works for my personal needs and presumably is a useful package for any LaTeX authors interested in incorporating musical symbols in their documents.

But *lilyglyphs* definitely is a package that lives on community contribution. Why? Because one of the main goals for continuing development is a growing coverage of glyphs. This task is very "scalable" and prone to "crowd-development" because any number of contributors can add commands from the fields they are working on and need glyphs from. I personally will probably only add new commands as I need them for my work, maybe sometimes getting the hang and complete a group. So anybody who uses the package to a more than casual degree is highly encouraged to share his additions with me and the community.

There are also existing issues with the LaTeX implementation that can be seen in the project's issue tracker on GitHub (https://github.com/openlilylib/lilyglyphs). Furthermore I'm sure there are quite a few things that could be implemented more elegantly, more efficiently, or with less package dependencies – this package is actually my first serious work with LaTeX …

And finally I have some ideas how *lilyglyphs*' functionality could be enhanced, and others could come up with more ideas. For example I would be interested in having parametrized commands that can print a number of related glyphs, there could be more layout options, I would like to be able to use global layout options for specific glyph groups (e.g. raise all accidentals by a certain amount) etc. And one day I would like to make the package compliant to the currently emerging SMuFL standard[1] in order to allow using glyphs from different music fonts.

So if you think this package is interesting, please contact me so it can evolve over time! If you're just using it, please submit any additions you make for your own use (or start submitting arbitrary commands to enhance the package's coverage). And if you're a LaTeX expert please don't hesitate to go right to the core and help me with the evolution of its code base …

---

[1] http://www.smufl.org