LilyPond

The music typesetter

The LilyPond development team

(For LilyPond version 2.10.29)

Table of Contents

Preface

It must have been during a rehearsal of the EJE (Eindhoven Youth Orchestra), somewhere in 1995 that Jan, one of the cranked violists, told Han-Wen, one of the distorted French horn players, about the grand new project he was working on. It was an automated system for printing music (to be precise, it was MPP, a preprocessor for MusiXTeX). As it happened, Han-Wen accidentally wanted to print out some parts from a score, so he started looking at the software, and he quickly got hooked. It was decided that MPP was a dead end. After lots of philosophizing and heated email exchanges, Han-Wen started LilyPond in 1996. This time, Jan got sucked into Han-Wen's new project.

In some ways, developing a computer program is like learning to play an instrument. In the beginning, discovering how it works is fun, and the things you cannot do are challenging. After the initial excitement, you have to practice and practice. Scales and studies can be dull, and if you are not motivated by others – teachers, conductors or audience – it is very tempting to give up. You continue, and gradually playing becomes a part of your life. Some days it comes naturally, and it is wonderful, and on some days it just does not work, but you keep playing, day after day.

Like making music, working on LilyPond can be dull work, and on some days it feels like plodding through a morass of bugs. Nevertheless, it has become a part of our life, and we keep doing it. Probably the most important motivation is that our program actually does something useful for people. When we browse around the net we find many people who use LilyPond, and produce impressive pieces of sheet music. Seeing that feels unreal, but in a very pleasant way.

Our users not only give us good vibes by using our program, many of them also help us by giving suggestions and sending bug reports, so we would like to thank all users that sent us bug reports, gave suggestions or contributed in any other way to LilyPond.

Playing and printing music is more than a nice analogy. Programming together is a lot of fun, and helping people is deeply satisfying, but ultimately, working on LilyPond is a way to express our deep love for music. May it help you create lots of beautiful music!

Han-Wen and Jan

Utrecht/Eindhoven, The Netherlands, July 2002.

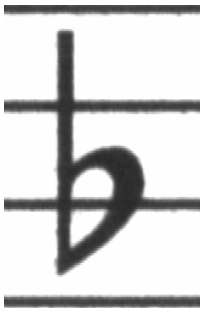
1 Introduction

1.1 Engraving

The art of music typography is called *(plate) engraving*. The term derives from the traditional process of music printing. Just a few decades ago, sheet music was made by cutting and stamping the music into a zinc or pewter plate in mirror image. The plate would be inked, the depressions caused by the cutting and stamping would hold ink. An image was formed by pressing paper to the plate. The stamping and cutting was completely done by hand. Making a correction was cumbersome, if possible at all, so the engraving had to be perfect in one go. Engraving was a highly specialized skill; a craftsman had to complete around five years of training before earning the title of master engraver, and another five years of experience were necessary to become truly skilled.

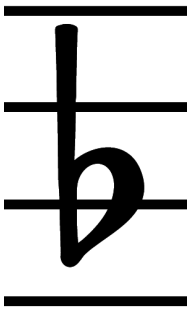
Nowadays, all newly printed music is produced with computers. This has obvious advantages; prints are cheaper to make, and editorial work can be delivered by email. Unfortunately, the pervasive use of computers has also decreased the graphical quality of scores. Computer printouts have a bland, mechanical look, which makes them unpleasant to play from.

The images below illustrate the difference between traditional engraving and typical computer output, and the third picture shows how LilyPond mimics the traditional look. The left picture shows a scan of a flat symbol from an edition published in 2000. The center depicts a symbol from a hand-engraved Bärenreiter edition of the same music. The left scan illustrates typical flaws of computer print: the staff lines are thin, the weight of the flat symbol matches the light lines and it has a straight layout with sharp corners. By contrast, the Bärenreiter flat has a bold, almost voluptuous rounded look. Our flat symbol is designed after, among others, this one. It is rounded, and its weight harmonizes with the thickness of our staff lines, which are also much thicker than lines in the computer edition.



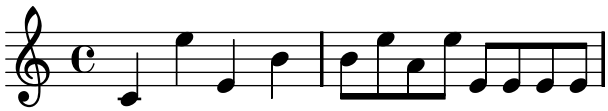
Henle (2000)

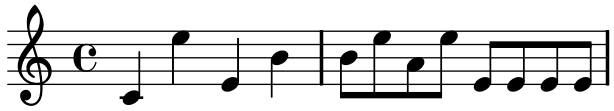
Bärenreiter (1950)



LilyPond Feta font (2003)

In spacing, the distribution of space should reflect the durations between notes. However, many modern scores adhere to the durations with mathematical precision, which leads to poor results. In the next example a motive is printed twice: once using exact mathematical spacing, and once with corrections. Can you spot which fragment is which?





Each bar in the fragment only uses notes that are played in a constant rhythm. The spacing should reflect that. Unfortunately, the eye deceives us a little; not only does it notice the distance between note heads, it also takes into account the distance between consecutive stems. As a result, the notes of an up-stem/down-stem combination should be put farther apart, and the notes of a down-stem/up-stem combination should be put closer together, all depending on the combined vertical positions of the notes. The upper two measures are printed with this correction, the lower two measures without, forming down-stem/up-stem clumps of notes.

Musicians are usually more absorbed with performing than with studying the looks of a piece of music, so nitpicking about typographical details may seem academical. But it is not. In larger pieces with monotonous rhythms, spacing corrections lead to subtle variations in the layout of every line, giving each one a distinct visual signature. Without this signature all lines would look the same, and they become like a labyrinth. If a musician looks away once or has a lapse in concentration, the lines might lose their place on the page.

Similarly, the strong visual look of bold symbols on heavy staff lines stands out better when the music is far away from the reader, for example, if it is on a music stand. A careful distribution of white space allows music to be set very tightly without cluttering symbols together. The result minimizes the number of page turns, which is a great advantage.

This is a common characteristic of typography. Layout should be pretty, not only for its own sake, but especially because it helps the reader in her task. For performance material like sheet music, this is of double importance: musicians have a limited amount of attention. The less attention they need for reading, the more they can focus on playing the music. In other words, better typography translates to better performances.

These examples demonstrate that music typography is an art that is subtle and complex, and that producing it requires considerable expertise, which musicians usually do not have. LilyPond is our effort to bring the graphical excellence of hand-engraved music to the computer age, and make it available to normal musicians. We have tuned our algorithms, font-designs, and program settings to produce prints that match the quality of the old editions we love to see and love to play from.

1.2 Automated engraving

How do we go about implementing typography? If craftsmen need over ten years to become true masters, how could we simple hackers ever write a program to take over their jobs?

The answer is: we cannot. Typography relies on human judgment of appearance, so people cannot be replaced completely. However, much of the dull work can be automated. If LilyPond solves most of the common situations correctly, this will be a huge improvement over existing software. The remaining cases can be tuned by hand. Over the course of years, the software can be refined to do more and more things automatically, so manual overrides are less and less necessary.

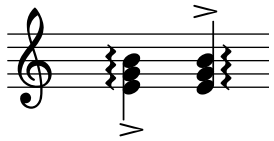
When we started, we wrote the LilyPond program entirely in the C++ programming language; the program's functionality was set in stone by the developers. That proved to be unsatisfactory for a number of reasons:

- When LilyPond makes mistakes, users need to override formatting decisions. Therefore, the user must have access to the formatting engine. Hence, rules and settings cannot be fixed by us at compile-time but must be accessible for users at run-time.
- Engraving is a matter of visual judgment, and therefore a matter of taste. As knowledgeable as we are, users can disagree with our personal decisions. Therefore, the definitions of typographical style must also be accessible to the user.

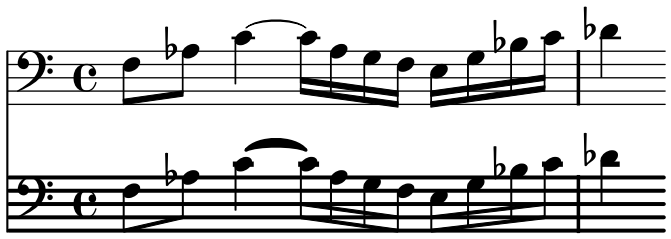
- Finally, we continually refine the formatting algorithms, so we need a flexible approach to rules. The C++ language forces a certain method of grouping rules that do not match well with how music notation works.

These problems have been addressed by integrating an interpreter for the Scheme programming language and rewriting parts of LilyPond in Scheme. The current formatting architecture is built around the notion of graphical objects, described by Scheme variables and functions. This architecture encompasses formatting rules, typographical style and individual formatting decisions. The user has direct access to most of these controls.

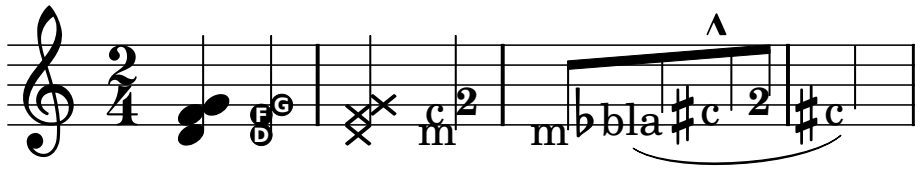
Scheme variables control layout decisions. For example, many graphical objects have a direction variable that encodes the choice between up and down (or left and right). Here you see two chords, with accents and arpeggios. In the first chord, the graphical objects have all directions down (or left). The second chord has all directions up (right).



The process of formatting a score consists of reading and writing the variables of graphical objects. Some variables have a preset value. For example, the thickness of many lines – a characteristic of typographical style – is a variable with a preset value. You are free to alter this value, giving your score a different typographical impression.



Formatting rules are also preset variables: each object has variables containing procedures. These procedures perform the actual formatting, and by substituting different ones, we can change the appearance of objects. In the following example, the rule which note head objects are used to produce their symbol is changed during the music fragment.



1.3 What symbols to engrave?

The formatting process decides where to place symbols. However, this can only be done once it is decided *what* symbols should be printed, in other words what notation to use.

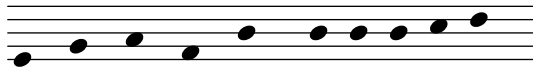
Common music notation is a system of recording music that has evolved over the past 1000 years. The form that is now in common use dates from the early renaissance. Although the basic form (i.e., note heads on a 5-line staff) has not changed, the details still evolve to express

the innovations of contemporary notation. Hence, it encompasses some 500 years of music. Its applications range from monophonic melodies to monstrous counterpoints for large orchestras.

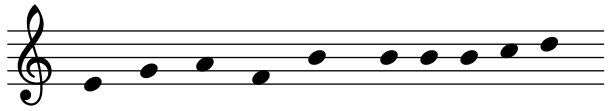
How can we get a grip on such a many-headed beast, and force it into the confines of a computer program? Our solution is to break up the problem of notation (as opposed to engraving, i.e., typography) into digestible and programmable chunks: every type of symbol is handled by a separate module, a so-called plug-in. Each plug-in is completely modular and independent, so each can be developed and improved separately. Such plug-ins are called `engraver`s, by analogy with craftsmen who translate musical ideas to graphic symbols.

In the following example, we see how we start out with a plug-in for note heads, the `Note_heads_engraver`.

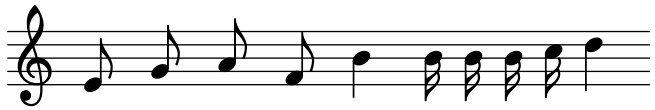
Then a `Staff_symbol_engraver` adds the staff



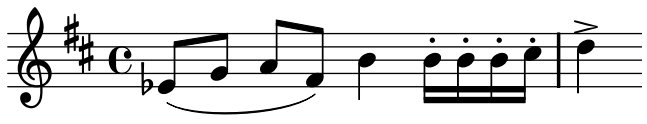
the `Clef_engraver` defines a reference point for the staff



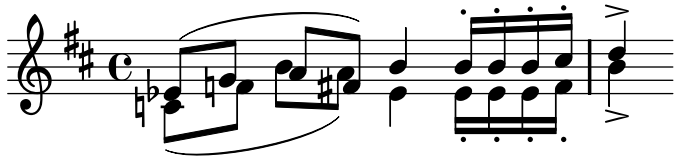
and the `Stem_engraver` adds stems.



The `Stem_engraver` is notified of any note head coming along. Every time one (or more, for a chord) note head is seen, a stem object is created and connected to the note head. By adding engravers for beams, slurs, accents, accidentals, bar lines, time signature, and key signature, we get a complete piece of notation.



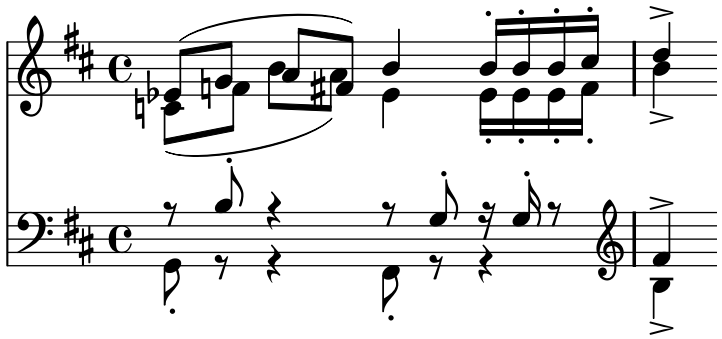
This system works well for monophonic music, but what about polyphony? In polyphonic notation, many voices can share a staff.



In this situation, the accidentals and staff are shared, but the stems, slurs, beams, etc., are private to each voice. Hence, engravers should be grouped. The engravers for note heads, stems, slurs, etc., go into a group called 'Voice context', while the engravers for key, accidental, bar, etc., go into a group called 'Staff context'. In the case of polyphony, a single Staff context contains more than one Voice context. Similarly, multiple Staff contexts can be put into a single Score context. The Score context is the top level notation context.

See also

Program reference: `Contexts`.



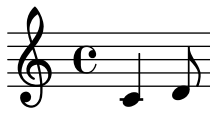
1.4 Music representation

Ideally, the input format for any high-level formatting system is an abstract description of the content. In this case, that would be the music itself. This poses a formidable problem: how can we define what music really is? Instead of trying to find an answer, we have reversed the question. We write a program capable of producing sheet music, and adjust the format to be as lean as possible. When the format can no longer be trimmed down, by definition we are left with content itself. Our program serves as a formal definition of a music document.

The syntax is also the user-interface for LilyPond, hence it is easy to type

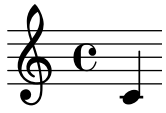
```
c'4 d'8
```

a quarter note C1 (middle C) and an eighth note D1 (D above middle C)



On a microscopic scale, such syntax is easy to use. On a larger scale, syntax also needs structure. How else can you enter complex pieces like symphonies and operas? The structure is formed by the concept of music expressions: by combining small fragments of music into larger ones, more complex music can be expressed. For example

```
c4
```

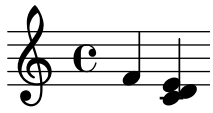


Chords can be constructed with `<<` and `>>` enclosing the notes

```
<<c4 d4 e4>>
```

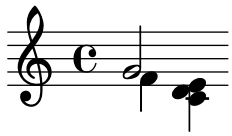
This expression is put in sequence by enclosing it in curly braces `{ ... }`

```
{ f4 <<c4 d4 e4>> }
```



The above is also an expression, and so it may be combined again with another simultaneous expression (a half note) using `<<`, `\\`, and `>>`

```
<< g2 \\ { f4 <<c4 d4 e4>> } >>
```

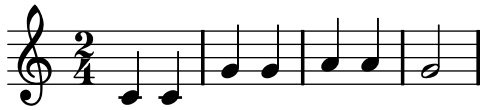


Such recursive structures can be specified neatly and formally in a context-free grammar. The parsing code is also generated from this grammar. In other words, the syntax of LilyPond is clearly and unambiguously defined.

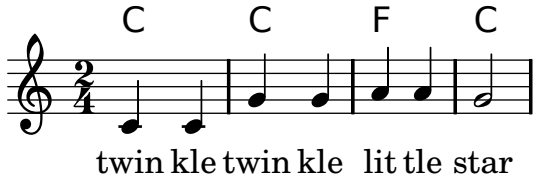
User-interfaces and syntax are what people see and deal with most. They are partly a matter of taste, and also subject of much discussion. Although discussions on taste do have their merit, they are not very productive. In the larger picture of LilyPond, the importance of input syntax is small: inventing neat syntax is easy, while writing decent formatting code is much harder. This is also illustrated by the line-counts for the respective components: parsing and representation take up less than 10% of the source code.

1.5 Example applications

We have written LilyPond as an experiment of how to condense the art of music engraving into a computer program. Thanks to all that hard work, the program can now be used to perform useful tasks. The simplest application is printing notes.



By adding chord names and lyrics we obtain a lead sheet.

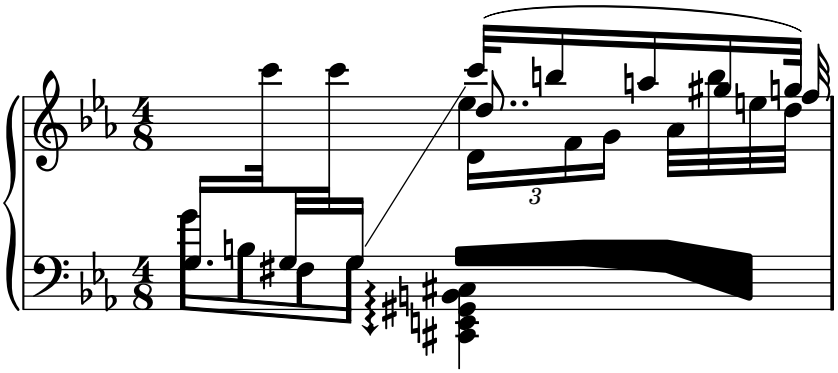


Polyphonic notation and piano music can also be printed. The following example combines some more exotic constructs.

Screech and boink

Random complex notation

Han-Wen Nienhuys



The fragments shown above have all been written by hand, but that is not a requirement. Since the formatting engine is mostly automatic, it can serve as an output means for other programs that manipulate music. For example, it can also be used to convert databases of musical fragments to images for use on websites and multimedia presentations.

This manual also shows an application: the input format is text, and can therefore be easily embedded in other text-based formats such as LaTeX, HTML, or in the case of this manual, Texinfo. By means of a special program, the input fragments can be replaced by music images in the resulting PDF or HTML output files. This makes it easy to mix music and text in documents.

1.6 About this manual

The manual is divided into the following chapters:

- *Chapter 2 [Tutorial], page 11* gives a gentle introduction to typesetting music. First time users should start here.
- *Chapter 3 [Putting it all together], page 34* explains some general concepts about the lilypond file format. If you are not certain where to place a command, read this chapter!
- *Chapter 4 [Working on LilyPond projects], page 42* discusses practical uses of LilyPond and how to avoid some common problems.
- *Chapter 5 [Tweaking output], page 51* shows how to change the default engraving that LilyPond produces.
- *Chapter 6 [Basic notation], page 59* discusses topics grouped by notation construct. This section gives details about basic notation that will be useful in almost any notation project.
- *Chapter 7 [Instrument-specific notation], page 109* discusses topics grouped by notation construct. This section gives details about special notation that will only be useful for particular instrument (or vocal) groups.
- *Chapter 8 [Advanced notation], page 167* discusses topics grouped by notation construct. This section gives details about complicated or unusual notation.
- *Chapter 9 [Changing defaults], page 213* explains how to fine tune layout.

- *Chapter 10 [Non-musical notation], page 234* discusses non-musical output such as titles, multiple movements, and how to select which MIDI instruments to use.
- *Chapter 11 [Spacing issues], page 246* discusses issues which affect the global output, such as selecting paper size or specifying page breaks.
- *Chapter 12 [Interfaces for programmers], page 271* explains how to create music functions.
- *Chapter 13 [Running LilyPond], page 286* shows how to run LilyPond and its helper programs. In addition, this section explains how to upgrade input files from previous versions of LilyPond.
- *Chapter 14 [LilyPond-book], page 295* explains the details behind creating documents with in-line music examples, like this manual.
- *Chapter 15 [Converting from other formats], page 306* explains how to run the conversion programs. These programs are supplied with the LilyPond package, and convert a variety of music formats to the `.ly` format.
- *Appendix A [Literature list], page 309* contains a set of useful reference books for those who wish to know more on notation and engraving.
- The *Appendix B [Scheme tutorial], page 310* presents a short introduction to Scheme, the programming language that music functions use.
- *Appendix C [Notation manual tables], page 312* are a set of tables showing the chord names, MIDI instruments, a list of color names, and the Feta font.
- *Appendix D [Templates], page 318* of LilyPond pieces. Just cut and paste a template into a file, add notes, and you're done!
- The *Appendix E [Cheat sheet], page 349* is a handy reference of the most common LilyPond commands.
- The *Appendix G [LilyPond command index], page 359* is an index of all LilyPond `\commands`.
- The *Appendix H [LilyPond index], page 362* is a complete index.

Once you are an experienced user, you can use the manual as reference: there is an extensive index[1], but the document is also available in a big HTML page, which can be searched easily using the search facility of a web browser.

If you are not familiar with music notation or music terminology (especially if you are a non-native English speaker), it is advisable to consult the glossary as well. The music glossary explains musical terms, and includes translations to various languages. It is a separate document, available in HTML and PDF.

This manual is not complete without a number of other documents. They are not available in print, but should be included with the documentation package for your platform

- Program reference

  The program reference is a set of heavily cross linked HTML pages, which document the nitty-gritty details of each and every LilyPond class, object, and function. It is produced directly from the formatting definitions used.

  Almost all formatting functionality that is used internally, is available directly to the user. For example, all variables that control thickness values, distances, etc., can be changed in input files. There are a huge number of formatting options, and all of them are described in this document. Each section of the notation manual has a **See also** subsection, which refers to the generated documentation. In the HTML document, these subsections have clickable links.

[1] If you are looking for something, and you cannot find it in the manual, that is considered a bug. In that case, please file a bug report.

- Various input examples.

  This collection of files shows various tips and tricks, and is available as a big HTML document, with pictures and explanatory texts included.

- The regression tests.

  This collection of files tests each notation and engraving feature of LilyPond in one file. The collection is primarily there to help us debug problems, but it can be instructive to see how we exercise the program. The format is similar to the tips and tricks document.

In all HTML documents that have music fragments embedded, the LilyPond input that was used to produce that image can be viewed by clicking the image.

The location of the documentation files that are mentioned here can vary from system to system. On occasion, this manual refers to initialization and example files. Throughout this manual, we refer to input files relative to the top-directory of the source archive. For example, '`input/test/bla.ly`' may refer to the file '`lilypond2.x.y/input/test/bla.ly`'. On binary packages for the Unix platform, the documentation and examples can typically be found somewhere below '`/usr/share/doc/lilypond/`'. Initialization files, for example '`scm/lily.scm`', or '`ly/engraver-init.ly`', are usually found in the directory '`/usr/share/lilypond/`'.

Finally, this and all other manuals, are available online both as PDF files and HTML from the web site, which can be found at http://www.lilypond.org/.

2 Tutorial

This tutorial starts with an introduction to the LilyPond music language and how to produce printed music. After this first contact we will explain how to create common musical notation.

2.1 First steps

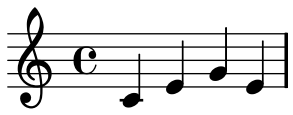
This section gives a basic introduction to working with LilyPond.

2.1.1 Compiling a file

The first example demonstrates how to start working with LilyPond. To create sheet music, we write a text file that specifies the notation. For example, if we write

```
{
  c' e' g' e'
}
```

the result looks like this



Warning: Every piece of LilyPond input needs to have `{` **curly braces** `}` placed around the input. The braces should also be surrounded by a space unless they are at the beginning or end of a line to avoid ambiguities. These may be omitted in some examples in this manual, but don't forget them in your own music!

In addition, LilyPond input is **case sensitive**. `{ c d e }` is valid input; `{ C D E }` will produce an error message.

Entering music and viewing output

In this section we will explain what commands to run and how to view or print the output.

MacOS X

If you double click LilyPond.app, it will open with an example file. Save it, for example, to '`test.ly`' on your Desktop, and then process it with the menu command '`Compile > Typeset File`'. The resulting PDF file will be displayed on your screen.

Be warned that the first time you ever run LilyPond, it will take a minute or two because all of the system fonts have to be analyzed first.

For future use of LilyPond, you should begin by selecting "New" or "Open". You must save your file before typesetting it. If any errors occur in processing, please see the log window.

Windows

On Windows, start up a text-editor[1] and enter

```
{
  c' e' g' e'
}
```

[1] Any simple or programmer-oriented editor with UTF-8 support will do, for example Notepad. Do not use a word processor, since these insert formatting codes that will confuse LilyPond.

Save it on the desktop as '`test.ly`' and make sure that it is not called '`test.ly.TXT`'. Double clicking '`test.ly`' will process the file and show the resulting PDF file. To edit an existing '`.ly`' file, right-click on it and select "Edit source".

If you double-click in the LilyPond icon on the Desktop, it will open a simple text editor with an example file. Save it, for example, to '`test.ly`' on your Desktop, and then double-click on the file to process it. After some seconds, you will get a file '`test.pdf`' on your desktop. Double-click on this PDF file to view the typeset score. An alternative method to process the '`test.ly`' file is to drag and drop it onto the LilyPond icon using your mouse pointer.

Double-clicking the file does not only result in a PDF file, but also produces a '`.log`' file that contains some information on what LilyPond has done to the file. If any errors occur, please examine this file.

Unix

Begin by opening a terminal window and starting a text editor. For example, you could open an xterm and execute `joe`[2]. In your text editor, enter the following input and save the file as '`test.ly`'

```
{
  c' e' g' e'
}
```

To process '`test.ly`', proceed as follows

```
lilypond test.ly
```

You will see something resembling

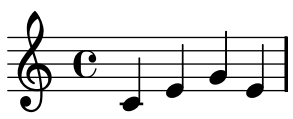
```
lilypond test.ly
GNU LilyPond 2.10.0
Processing `test.ly'
Parsing...
Interpreting music... [1]
Preprocessing graphical objects...
Calculating line breaks... [2]
Layout output to `test.ps'...
Converting to `test.pdf'...
```

The result is the file '`test.pdf`' which you can print or view with the standard facilities of your operating system.[3]

2.1.2 Simple notation

LilyPond will add some notation elements automatically. In the next example, we have only specified four pitches, but LilyPond has added a clef, time signature, and rhythms.

```
{
  c' e' g' e'
}
```



[2] There are macro files for VIM addicts, and there is a `LilyPond-mode` for Emacs addicts. If they have not been installed already, refer to the file '`INSTALL.txt`'. The easiest editing environment is '`LilyPondTool`'. See Section 13.6 [Editor support], page 293 for more information.

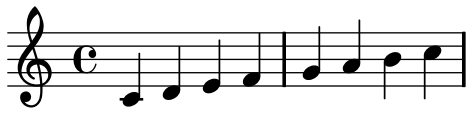
[3] If your system does not have any such tools installed, you can try Ghostscript, a freely available package for viewing and printing PDF and PostScript files.

This behavior may be altered, but in most cases these automatic values are useful.

Pitches

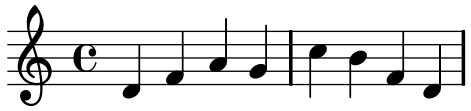
The easiest way to enter notes is by using `\relative` mode. In this mode, the `interval` between the previous note and the current note is assumed to be within a `fourth`. We begin by entering the most elementary piece of music, a `scale`.

```
\relative c' {
  c d e f
  g a b c
}
```



The initial note is `middle C`. Each successive note is within a fourth of the previous note – in other words, the first '`c`' is the closest C to middle C. This is followed by the closest D to the previous note. We can create melodies which have larger intervals:

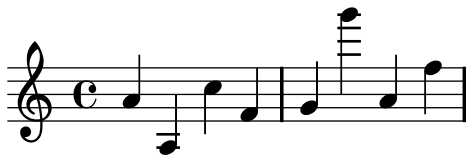
```
\relative c' {
  d f a g
  c b f d
}
```



As you may notice, this example does not start on middle C. The first note – the '`d`' – is the closest D to middle C.

To add intervals that are larger than a fourth, we can raise the octave by adding a single quote `'` (or apostrophe) to the note name. We can lower the octave by adding a comma `,` to the note name.

```
\relative c'' {
  a a, c' f,
  g g'' a,, f'
}
```

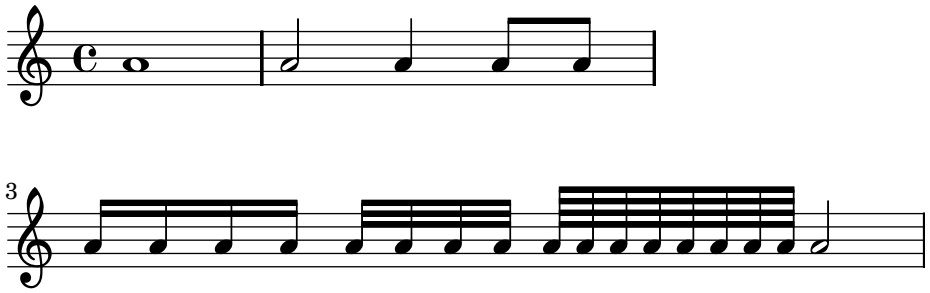


To change a note by two (or more!) octaves, we use multiple `''` or `,,` – but be careful that you use two single quotes `''` and not one double quote `"` ! The initial value in `\relative c'` may also be modified like this.

Durations (rhythms)

The `duration` of a note is specified by a number after the note name. '`1`' for a `whole note`, '`2`' for a `half note`, '`4`' for a `quarter note` and so on. Beams are added automatically.

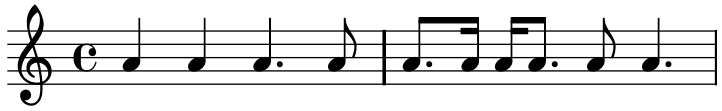
```
\relative c'' {
  a1
  a2 a4 a8 a
  a16 a a a a32 a a a a64 a a a a a a a a2
}
```



If you do not specify a duration, the previous duration is used for the next note. The duration of the first note defaults to a quarter.

To create `dotted notes`, add a dot '`.`' to the duration number.

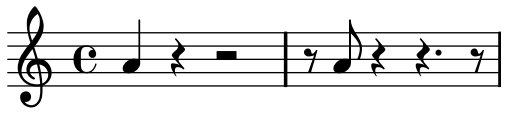
```
\relative c'' {
  a a a4. a8
  a8. a16 a a8. a8 a4.
}
```



Rests

A `rest` is entered just like a note with the name '`r`':

```
\relative c'' {
  a r r2
  r8 a r4 r4. r8
}
```



Time signature

The `time signature`) can be set with the `\time` command:

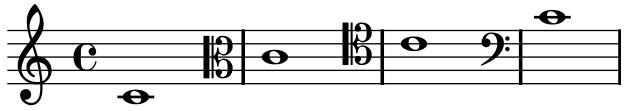
```
\relative c'' {
  \time 3/4
  a4 a a
  \time 6/8
  a4. a
```

```
  \time 4/4
  a4 a a a
}
```

Clef

The `clef` can be set using the `\clef` command:

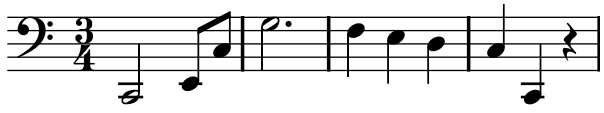
```
\relative c' {
  \clef treble
  c1
  \clef alto
  c1
  \clef tenor
  c1
  \clef bass
  c1
}
```



All together

Here is a small example showing all these elements together:

```
\relative c, {
  \time 3/4
  \clef bass
  c2 e8 c' g'2.
  f4 e d c4 c, r4
}
```



More information

Entering pitches and durations
: see Section 6.1 [Pitches], page 59 and Section 6.2.1 [Durations], page 66.

Rests
: see Section 6.1.9 [Rests], page 64.

Time signatures and other timing commands
: see Section 6.4.3 [Time signature], page 78.

Clefs
: see Section 6.4.1 [Clef], page 76.

2.1.3 Working on text files

LilyPond input files are treated like files in most programming languages: they are case sensitive, white-space insensitive, expressions are formed with curly braces `{ }`, and comments are denoted with `%` or `%{ .. %}`.

If the previous sentence sounds like nonsense, don't worry! We'll explain what all these terms mean:

- **Case sensitive**: it matters whether you enter a letter in lower case (i.e. `a, b, s, t`) or upper case (i.e. `A, B, S, T`). Notes are lower case: `{ c d e }` is valid input; `{ C D E }` will produce an error message.
- **Whitespace insensitive**: it does not matter how many spaces (or new lines) you add. `{ c d e }` means the same thing as `{ c d e }` and

```
{
c                       d
  e }
```

  Of course, the previous example is hard to read. A good rule of thumb is to indent code blocks with either a tab or two spaces:

```
{
  c d e
}
```

- **Expressions:** Every piece of LilyPond input needs to have `{` **curly braces** `}` placed around the input. These braces tell LilyPond that the input is a single music expression, just like parenthesis '`()`' in mathematics. The braces should be surrounded by a space unless they are at the beginning or end of a line to avoid ambiguities.

  A function (such as `\relative { }`) also counts as a single music expression.

- **Comments**: A comment is a remark for the human reader of the music input; it is ignored while parsing, so it has no effect on the printed output. There are two types of comments. The percent symbol '`%`' introduces a line comment; anything after '`%`' on that line is ignored. A block comment marks a whole section of music input as a comment. Anything that is enclosed in `%{` and `%}` is ignored. The following fragment shows possible uses for comments

```
% notes for twinkle twinkle follow
  c4 c g' g a a g2

%{
    This line, and the notes below
    are ignored, since they are in a
    block comment.

    g g f f e e d d c2
%}
```

There are more tips for constructing input files in Section 4.1 [Suggestions for writing LilyPond files], page 42.

2.1.4 How to read the tutorial

As we saw in Section 2.1.3 [Working on text files], page 16, LilyPond input must be surrounded by `{ }` marks or a `\relative c'' { ... }`. For the rest of this manual, most examples will omit this.

If you are reading the HTML documentation and wish to see the exact exact LilyPond code that was used to create the example, simply click on the picture. If you are not reading the

HTML version, you could copy and paste the displayed input, but you **must** add the `\relative c'' { }` like this:

```
\relative c'' {
  ... example goes here...
}
```

Why omit the braces? Most examples in this manual can be inserted into the middle of a longer piece of music. For these examples, it does not make sense to add `\relative c'' { }` – you should not place a `\relative` inside another `\relative`, so you would not be able to copy a small documentation example and paste it inside a longer piece of your own.

2.2 Single staff notation

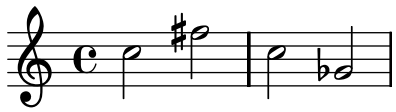
This section introduces common notation that is used for one voice on one staff.

2.2.1 Relative note names

As we saw in Section 2.1.2 [Simple notation], page 12, LilyPond calculates the pitch of each note relative to the previous one[4]. If no extra octave marks (`'` and `,`) are added, it assumes that each pitch is within a fourth of the previous note.

LilyPond examines pitches based on the note names – in other words, an augmented fourth is *not* the same as a diminished fifth. If we begin at a C, then an F-sharp will be placed a higher than the C, while a G-flat will be placed lower than the C.

```
c2 fis
c2 ges
```



More information

Relative octaves
: see Section 6.1.6 [Relative octaves], page 62.

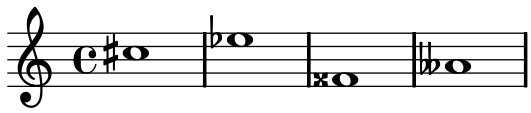
Octave check
: see Section 6.1.7 [Octave check], page 63.

2.2.2 Accidentals and key signatures

Accidentals

A `sharp` pitch is made by adding '`is`' to the name, and a `flat` pitch by adding '`es`'. As you might expect, a `double sharp` or `double flat` is made by adding '`isis`' or '`eses`'[5]

```
cis1 ees fisis, aeses
```



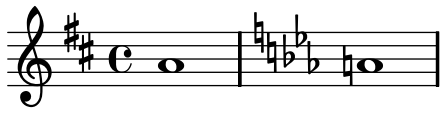
[4] There is another mode of entering pitches, Section 2.5.3 [Absolute note names], page 30, but in practice relative mode is much easier and safer to use.

[5] This syntax derived from note naming conventions in Nordic and Germanic languages, like German and Dutch. To use other names for accidentals, see Section 6.1.5 [Note names in other languages], page 61.

Key signatures

The key signature is set with the command `\key` followed by a pitch and `\major` or `\minor`.

```
\key d \major
a1
\key c \minor
a
```



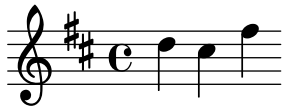
Warning: key signatures and pitches

To determine whether to print an accidental, LilyPond examines the pitches and the key signature. The key signature only effects the *printed* accidentals, not the actual pitches! This is a feature that often causes confusion to newcomers, so let us explain it in more detail.

LilyPond makes a sharp distinction between musical content and layout. The alteration (flat, natural or sharp) of a note is part of the pitch, and is therefore musical content. Whether an accidental (a *printed* flat, natural or sharp sign) is printed in front of the corresponding note is a question of layout. Layout is something that follows rules, so accidentals are printed automatically according to those rules. The pitches in your music are works of art, so they will not be added automatically, and you must enter what you want to hear.

In this example

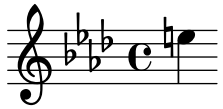
```
\key d \major
d cis fis
```



No note has a printed accidental, but you must still add the '`is`' to `cis` and `fis`.

The code '`e`' does not mean "print a black dot just below the first line of the staff." Rather, it means: "there is a note with pitch E-natural." In the key of A-flat major, it *does* get an accidental:

```
\key aes \major
e
```



Adding all alterations explicitly might require a little more effort when typing, but the advantage is that transposing is easier, and accidentals can be printed according to different conventions. See Section 9.1.1 [Automatic accidentals], page 213 for some examples how accidentals can be printed according to different rules.

More information

Accidentals

see Section 6.1.2 [Accidentals], page 60 and Section 9.1.1 [Automatic accidentals], page 213.

Key signature

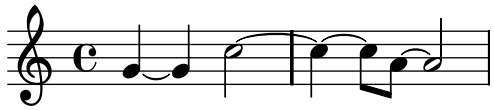
see Section 6.4.2 [Key signature], page 77.

2.2.3 Ties and slurs

Ties

A `tie` is created by appending a tilde '~' to the first note being tied

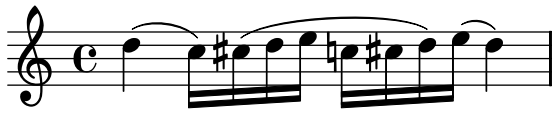
```
g4~ g c2~
c4 ~ c8 a8 ~ a2
```



Slurs

A `slur` is a curve drawn across many notes. The starting note and ending note are marked with '(' and ')' respectively.

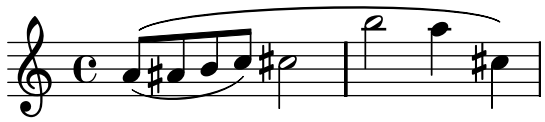
```
d4( c16) cis( d e c cis d) e( d4)
```



Phrasing slurs

Slurs to indicate longer phrasing can be entered with `\(` and `\)`. You can have both legato slurs and phrasing slurs at the same time, but you cannot have simultaneous slurs or simultaneous phrasing slurs.

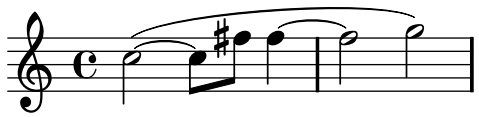
```
a8(\( ais b c) cis2 b'2 a4 cis,\)
```



Warnings: slurs vs. ties

A slur looks like a `tie`, but it has a different meaning. A tie simply makes the first note longer, and can only be used on pairs of notes with the same pitch. Slurs indicate the articulations of notes, and can be used on larger groups of notes. Slurs and ties can be nested.

```
c2~( c8 fis fis4 ~ fis2 g2)
```



More information

Ties see Section 6.5.1 [Ties], page 86.

Slurs see Section 6.5.2 [Slurs], page 88.

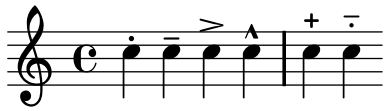
Phrasing slurs
see Section 6.5.3 [Phrasing slurs], page 89.

2.2.4 Articulation and dynamics

Articulations

Common `articulations` can be added to a note using a dash '`-`' and a single character:

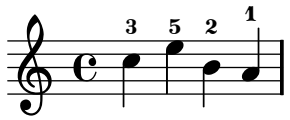
```
c-. c-- c-> c-^ c-+ c-_
```



Fingerings

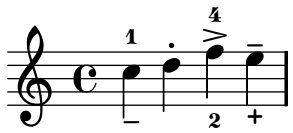
Similarly, fingering indications can be added to a note using a dash ('`-`') and the digit to be printed:

```
c-3 e-5 b-2 a-1
```



Articulations and fingerings are usually placed automatically, but you can specify a direction using '`^`' (up) or '`_`' (down). You can also use multiple articulations on the same note. However, in most cases it is best to let LilyPond determine the articulation directions.

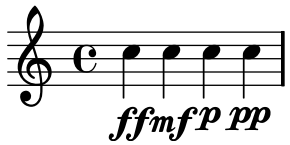
```
c_-^1 d^. f^4_2-> e^-_+
```



Dynamics

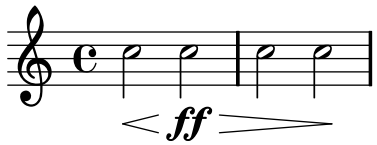
Dynamic signs are made by adding the markings (with a backslash) to the note

```
c\ff c\mf c\p c\pp
```



Crescendi and decrescendi are started with the commands `\<` and `\>`. An ending dynamic, for example `\f`, will finish the (de)crescendo, or the command `\!` can be used

```
c2\< c2\ff\> c2 c2\!
```



More information

Articulations
: see Section 6.6.1 [Articulations], page 94.

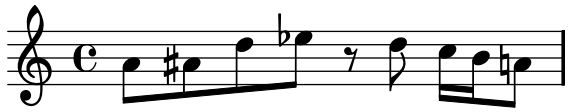
Fingering see Section 6.6.2 [Fingering instructions], page 96.

Dynamics see Section 6.6.3 [Dynamics], page 98.

2.2.5 Automatic and manual beams

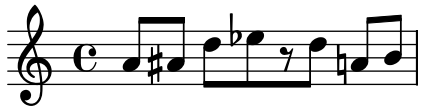
All `beams` are drawn automatically:

```
a8 ais d ees r d c16 b a8
```



If you do not like the automatic beams, they may be overridden manually. Mark the first note to be beamed with '`[`' and the last one with '`]`'.

```
a8[ ais] d[ ees r d] a b
```



More information

Automatic beams
: see Section 6.5.5 [Automatic beams], page 90.

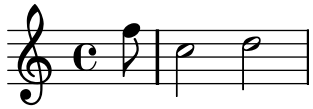
Manual beams
: see Section 6.5.6 [Manual beams], page 90.

2.2.6 Advanced rhythmic commands

Partial measure

A pickup (or `anacrusis`) is entered with the keyword `\partial`. It is followed by a duration: `\partial 4` is a quarter note pickup and `\partial 8` an eighth note.

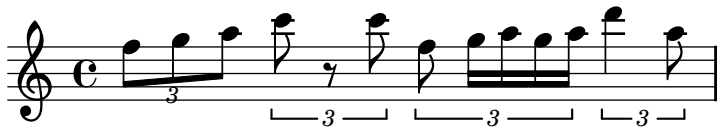
```
\partial 8
f8 c2 d
```



Tuplets

Tuplets are made with the `\times` keyword. It takes two arguments: a fraction and a piece of music. The duration of the piece of music is multiplied by the fraction. Triplets make notes occupy 2/3 of their notated duration, so a triplet has 2/3 as its fraction

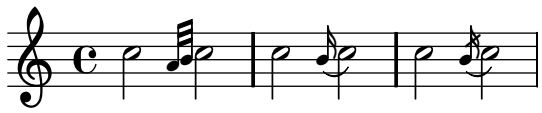
```
\times 2/3 { f8 g a }
\times 2/3 { c r c }
\times 2/3 { f,8 g16[ a g a] }
\times 2/3 { d4 a8 }
```



Grace notes

Grace notes are created with the `\grace` command, although they can also be created by prefixing a music expression with the keyword `\appoggiatura` or `\acciaccatura`

```
c2 \grace { a32[ b] } c2
c2 \appoggiatura b16 c2
c2 \acciaccatura b16 c2
```



More information

Grace notes
: see Section 6.5.7 [Grace notes], page 91,

Tuplets
: see Section 6.2.3 [Tuplets], page 67,

Pickups
: see Section 6.4.4 [Partial measures], page 79.

2.3 Multiple notes at once

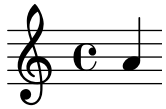
This section introduces having more than one note at the same time: multiple instruments, multiple staves for a single instrument (i.e. piano), and chords.

Polyphony in music refers to having more than one voice occurring in a piece of music. Polyphony in LilyPond refers to having more than one voice on the same staff.

2.3.1 Music expressions explained

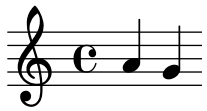
In LilyPond input files, music is represented by *music expressions*. A single note is a music expression, although it is not valid input all on its own.

```
a4
```



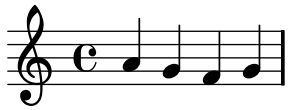
Enclosing a group of notes in braces creates a new music expression:

```
{ a4 g4 }
```



Putting a group of music expressions (e.g. notes) in braces means that they are in sequence (i.e. each one follows the previous one). The result is another music expression:

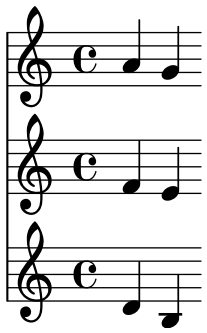
```
{ { a4 g } f g }
```



Simultaneous music expressions: multiple staves

This technique is useful for polyphonic music. To enter music with more voices or more staves, we combine expressions in parallel. To indicate that two voices should play at the same time, simply enter a simultaneous combination of music expressions. A 'simultaneous' music expression is formed by enclosing expressions inside << and >>. In the following example, three sequences (all containing two separate notes) are combined simultaneously:

```
\relative c'' {
  <<
    { a4 g }
    { f e }
    { d b }
  >>
}
```



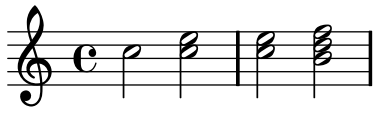
Note that we have indented each level of the input with a different amount of space. LilyPond does not care how much (or little) space there is at the beginning of a line, but indenting LilyPond code like this makes it much easier for humans to read.

Warning: each note is relative to the previous note in the input, not relative to the `c''` in the initial `\relative` command.

Simultaneous music expressions: single staff

To determine the number of staves in a piece, LilyPond looks at the first expression. If it is a single note, there is one staff; if there is a simultaneous expression, there is more than one staff.

```
\relative c'' {
  c2 <<c e>>
  << { e f } { c <<b d>> } >>
}
```



Analogy: mathematical expressions

This mechanism is similar to mathematical formulas: a big formula is created by composing small formulas. Such formulas are called expressions, and their definition is recursive so you can make arbitrarily complex and large expressions. For example,

```
1

1 + 2

(1 + 2) * 3

((1 + 2) * 3) / (4 * 5)
```

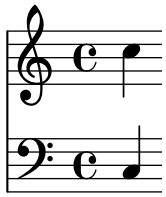
This is a sequence of expressions, where each expression is contained in the next (larger) one. The simplest expressions are numbers, and larger ones are made by combining expressions with operators (like '+', '*' and '/') and parentheses. Like mathematical expressions, music expressions can be nested arbitrarily deep, which is necessary for complex music like polyphonic scores.

2.3.2 Multiple staves

As we saw in Section 2.3.1 [Music expressions explained], page 22, LilyPond input files are constructed out of music expressions. If the score begins with simultaneous music expressions, LilyPond creates multiples staves. However, it is easier to see what happens if we create each staff explicitly.

To print more than one staff, each piece of music that makes up a staff is marked by adding `\new Staff` before it. These `Staff` elements are then combined in parallel with `<<` and `>>`:

```
\relative c'' {
  <<
    \new Staff { \clef treble c }
    \new Staff { \clef bass c,, }
  >>
}
```



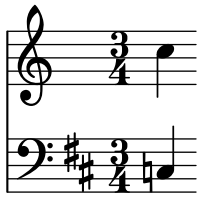
The command `\new` introduces a 'notation context.' A notation context is an environment in which musical events (like notes or `\clef` commands) are interpreted. For simple pieces, such notation contexts are created automatically. For more complex pieces, it is best to mark contexts explicitly.

There are several types of contexts. `Score`, `Staff`, and `Voice` handle melodic notation, while `Lyrics` sets lyric texts and `ChordNames` prints chord names.

In terms of syntax, prepending `\new` to a music expression creates a bigger music expression. In this way it resembles the minus sign in mathematics. The formula $(4 + 5)$ is an expression, so $-(4 + 5)$ is a bigger expression.

Time signatures entered in one staff affects all other staves, but the key signature of one staff does *not* affect other staves[6].

```
\relative c'' {
  <<
    \new Staff { \clef treble \time 3/4 c }
    \new Staff { \clef bass \key d \major c,, }
  >>
}
```



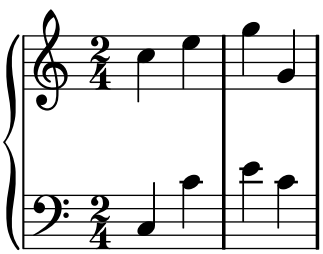
2.3.3 Piano staves

Piano music is typeset in two staves connected by a brace. Printing such a staff is similar to the polyphonic example in Section 2.3.2 [Multiple staves], page 24, but now this entire expression is inserted inside a `PianoStaff`:

```
\new PianoStaff <<
  \new Staff ...
  \new Staff ...
>>
```

Here is a small example

```
\relative c'' {
  \new PianoStaff <<
    \new Staff { \time 2/4 c4 e g g, }
    \new Staff { \clef bass c,, c' e c }
  >>
}
```



[6] This behavior may be changed if desired; see Chapter 9 [Changing defaults], page 213 for details.

More information

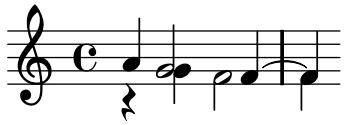
See Section 7.1 [Piano music], page 109.

2.3.4 Single staff polyphony

When different melodic lines are combined on a single staff they are printed as polyphonic voices; each voice has its own stems, slurs and beams, and the top voice has the stems up, while the bottom voice has them down.

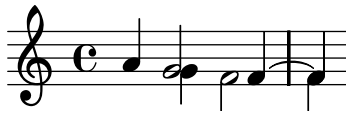
Entering such parts is done by entering each voice as a sequence (with `{...}`) and combining these simultaneously, separating the voices with `\\`

```
<<
  { a4 g2 f4~ f4 } \\
  { r4 g4 f2 f4 }
>>
```



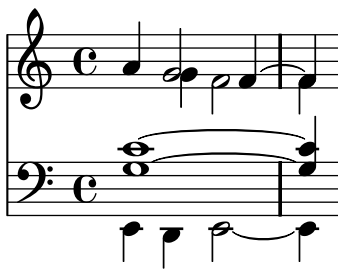
For polyphonic music typesetting, spacer rests can also be convenient; these are rests that do not print. They are useful for filling up voices that temporarily do not play. Here is the same example with a spacer rest ('`s`') instead of a normal rest ('`r`'),

```
<<
  { a4 g2 f4~ f4 } \\
  { s4 g4 f2 f4 }
>>
```



Again, these expressions can be nested arbitrarily.

```
<<
  \new Staff <<
    { a4 g2 f4~ f4 } \\
    { s4 g4 f2 f4 }
  >>
  \new Staff <<
    \clef bass
    { <c g>1 ~ <c g>4 } \\
    { e,,4 d e2 ~ e4}
  >>
>>
```



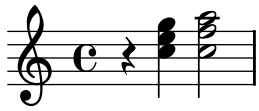
More information

See Section 6.3.3 [Basic polyphony], page 70.

2.3.5 Combining notes into chords

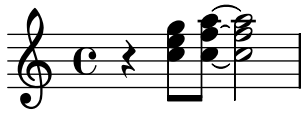
Chords can be made by surrounding pitches with single angle brackets. Angle brackets are the symbols '<' and '>'.

```
r4 <c e g>4 <c f a>2
```

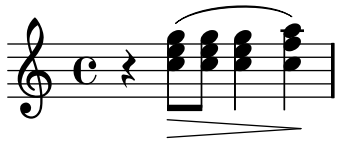


You can combine markings like beams and ties with chords. They must be placed outside the angle brackets

```
r4 <c e g>8[ <c f a>]~ <c f a>2
```



```
r4 <c e g>8\>( <c e g> <c e g>4 <c f a>\!)
```



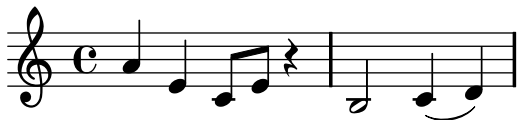
2.4 Songs

This section introduces vocal music and simple song sheets.

2.4.1 Printing lyrics

Consider a simple melody:

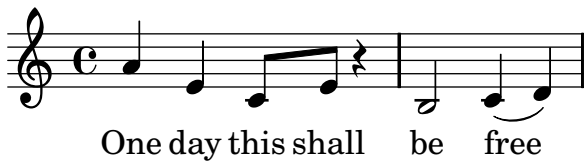
```
\relative c'' {
  a4 e c8 e r4
  b2 c4( d)
}
```



The lyrics can be set to these notes, combining both with the `\addlyrics` keyword. Lyrics are entered by separating each syllable with a space.

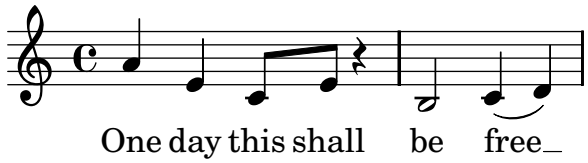
```
<<
  \relative c'' {
    a4 e c8 e r4
    b2 c4( d)
```

```
  }
  \addlyrics { One day this shall be free }
>>
```



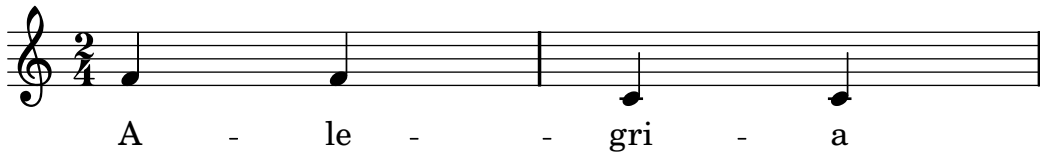
This melody ends on a `melisma`, a single syllable ('free') sung to more than one note. This is indicated with an *extender line*. It is entered as two underscores `__`:

```
<<
  \relative c'' {
    a4 e c8 e r4
    b2 c4( d)
  }
  \addlyrics { One day this shall be free __ }
>>
```



Similarly, hyphens between words can be entered as two dashes, resulting in a centered hyphen between two syllables

```
<<
  \relative c' {
    \time 2/4
    f4 f c c
  }
  \addlyrics { A -- le -- gri -- a }
>>
```



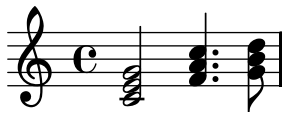
More information

More options, such as putting multiple stanzas below a melody, are discussed in Section 7.3 [Vocal music], page 118.

2.4.2 A lead sheet

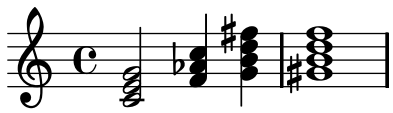
In popular music it is common to denote accompaniment with chord names. Such chords can be entered like notes,

```
\chordmode { c2 f4. g8 }
```



Now each pitch is read as the root of a chord instead of a note. This mode is switched on with `\chordmode`. Other chords can be created by adding modifiers after a colon. The following example shows a few common modifiers:

```
\chordmode { c2 f4:m g4:maj7 gis1:dim7 }
```

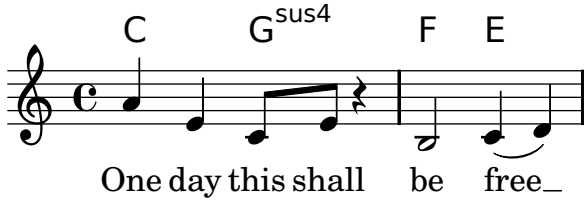


For lead sheets, chords are not printed on staves, but as names on a line for themselves. This is achieved by using `\chords` instead of `\chordmode`. This uses the same syntax as `\chordmode`, but renders the notes in a `ChordNames` context, with the following result:

```
\chords { c2 f4.:m g4.:maj7 gis8:dim7 }
```

When put together, chord names, lyrics and a melody form a lead sheet,

```
<<
  \chords { c2 g:sus4 f e }
  \relative c'' {
    a4 e c8 e r4
    b2 c4( d)
  }
  \addlyrics { One day this shall be free __ }
>>
```



More information

A complete list of modifiers and other options for layout can be found in Section 6.3.1 [Chords], page 70.

2.5 Final touches

This is the final section of the tutorial; it demonstrates how to add the final touches to simple pieces, and provides an introduction to the rest of the manual.

2.5.1 Version number

The `\version` statement marks for which version of LilyPond the file was written. To mark a file for version 2.10.1, place

```
\version "2.10.10"
```

at the top of your LilyPond file.

These annotations make future upgrades of LilyPond go more smoothly. Changes in the syntax are handled with a special program, '`convert-ly`' (see Section 13.3 [Updating files with convert-ly], page 290), and it uses `\version` to determine what rules to apply.

2.5.2 Adding titles

The title, composer, opus number, and similar information are entered in the `\header` block. This exists outside of the main music expression; the `\header` block is usually placed underneath the Section 2.5.1 [Version number], page 30.

```
\version "2.10.10"
\header {
  title = "Symphony"
  composer = "Me"
  opus = "Op. 9"
}

{
  ... music ...
}
```

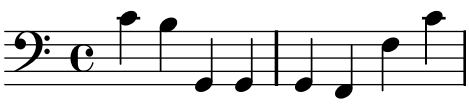
When the file is processed, the title and composer are printed above the music. More information on titling can be found in Section 10.2.1 [Creating titles], page 238.

2.5.3 Absolute note names

So far we have always used `\relative` to define pitches. This is the easiest way to enter most music, but another way of defining pitches exists: absolute mode.

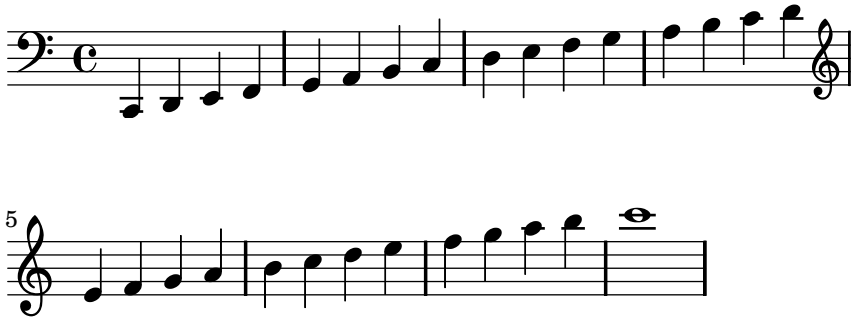
If you omit the `\relative`, LilyPond treats all pitches as absolute values. A `c'` will always mean middle C, a `b` will always mean the note one step below middle C, and a `g,` will always mean the note on the bottom staff of the bass clef.

```
{
  \clef bass
  c' b g, g,
  g, f, f c'
}
```



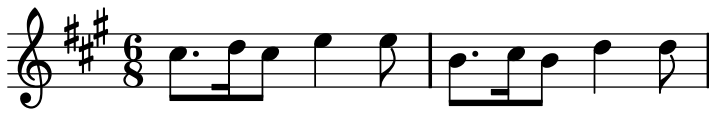
Here is a four-octave scale:

```
{
  \clef bass
  c, d, e, f,
  g, a, b, c
  d e f g
  a b c' d'
  \clef treble
  e' f' g' a'
  b' c'' d'' e''
  f'' g'' a'' b''
  c'''1
}
```



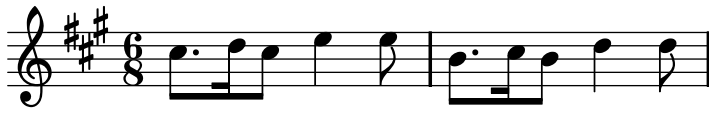
As you can see, writing a melody in the treble clef involves a lot of quote ' marks. Consider this fragment from Mozart:

```
{
  \key a \major
  \time 6/8
  cis''8. d''16 cis''8 e''4 e''8
  b'8. cis''16 b'8 d''4 d''8
}
```



All these quotes makes the input less readable and it is a source of errors. With `\relative`, the previous example is much easier to read:

```
\relative c'' {
  \key a \major
  \time 6/8
  cis8. d16 cis8 e4 e8
  b8. cis16 b8 d4 d8
}
```



If you make a mistake with an octave mark (' or ,) while working in `\relative` mode, it is very obvious – many notes will be in the wrong octave. When working in absolute mode, a single mistake will not be as visible, and will not be as easy to find.

However, absolute mode is useful for music which has large intervals, and is extremely useful for computer-generated LilyPond files.

2.5.4 Organizing pieces with identifiers

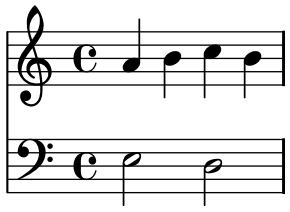
When all of the elements discussed earlier are combined to produce larger files, the music expressions get a lot bigger. In polyphonic music with many staves, the input files can become very confusing. We can reduce this confusion by using *identifiers*.

With identifiers (also known as variables or macros), we can break up complex music expressions. An identifier is assigned as follows

```
namedMusic = { ... }
```

The contents of the music expression `namedMusic` can be used later by placing a backslash in front of the name (`\namedMusic`, just like a normal LilyPond command). Identifiers must be defined *before* the main music expression.

```
violin = \new Staff { \relative c'' {
  a4 b c b
}}
cello = \new Staff { \relative c {
  \clef bass
  e2 d
}}
{
  <<
    \violin
    \cello
  >>
}
```



The name of an identifier must have alphabetic characters only: no numbers, underscores, or dashes.

It is possible to use variables for many other types of objects in the input. For example,

```
width = 4.5\cm
name = "Wendy"
aFivePaper = \paper { paperheight = 21.0 \cm }
```

Depending on its contents, the identifier can be used in different places. The following example uses the above variables:

```
\paper {
  \aFivePaper
  line-width = \width
}
{ c4^\name }
```

2.5.5 After the tutorial

After finishing the tutorial, you should probably try writing a piece or two. Start with one of the Appendix D [Templates], page 318 and add notes. If you need any notation that was not covered in the tutorial, look at the Notation Reference, starting with Chapter 6 [Basic notation], page 59. If you want to write for an instrument ensemble which is not covered in the templates, take a look at Section 3.1 [Extending the templates], page 34.

Once you have written a few short pieces, read the rest of the Learning Manual (chapters 3-5). There's nothing wrong with reading them now, of course! However, the rest of the Learning Manual assumes that you are familiar with LilyPond input. You may wish to skim these chapters right now, and come back to them after you have more experience.

2.5.6 How to read the manual

As we saw in Section 2.1.4 [How to read the tutorial], page 16, many examples in the tutorial omitted a `\relative c'' { ... }` around the printed example.

In the rest of the manual, we are much more lax about the printed examples: sometimes they may have omitted a `\relative c'' { ... }`, but in other times a different initial pitch may be used (such as `c'` or `c,,`), and in some cases the whole example is in absolute note mode! However, ambiguities like this only exist where the pitches are not important. In any example where the pitch matters, we have explicitly stated our `\relative` our our absolute-mode `{ }`.

If you are still confused about the exact LilyPond input that was used in an example, read the HTML version (if you are not already doing so) and click on the picture of the music. This will display the exact input that LilyPond used to generate this manual.

3 Putting it all together

This chapter discusses general LilyPond concepts and how to create `\score` blocks.

3.1 Extending the templates

You've read the tutorial, you know how to write music. But how can you get the staves that you want? The templates are ok, but what if you want something that isn't covered?

Start off with the template that seems closest to what you want to end up with. Let's say that you want to write something for soprano and cello. In this case, we would start with "Notes and lyrics" (for the soprano part).

```
\version "2.10.10"
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

text = \lyricmode {
  Aaa Bee Cee Dee
}

\score{
  <<
    \new Voice = "one" {
      \autoBeamOff
      \melody
    }
    \new Lyrics \lyricsto "one" \text
  >>
  \layout { }
  \midi { }
}
```

Now we want to add a cello part. Let's look at the "Notes only" example:

```
\version "2.10.10"
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

\score {
\new Staff \melody
\layout { }
\midi { }
}
```

We don't need two `\version` commands. We'll need the `melody` section. We don't want two `\score` sections – if we had two `\score`s, we'd get the two parts separately. We want them together, as a duet. Within the `\score` section, we don't need two `\layout` or `\midi`.

If we simply cut and paste the `melody` section, we would end up with two `melody` sections. So let's rename them. We'll call the section for the soprano `sopranoMusic` and the section for the cello `celloMusic`. While we're doing this, let's rename `text` to be `sopranoLyrics`. Remember to rename both instances of all these names – both the initial definition (the `melody = relative c' {` part) and the name's use (in the `\score` section).

While we're doing this, let's change the cello part's staff – celli normally use bass clef. We'll also give the cello some different notes.

```
\version "2.10.10"
sopranoMusic = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

sopranoLyrics = \lyricmode {
  Aaa Bee Cee Dee
}

celloMusic = \relative c {
  \clef bass
  \key c \major
  \time 4/4

  d4 g fis8 e d4
}

\score{
  <<
    \new Voice = "one" {
      \autoBeamOff
      \sopranoMusic
    }
    \new Lyrics \lyricsto "one" \sopranoLyrics
  >>
  \layout { }
  \midi { }
}
```

This is looking promising, but the cello part won't appear in the score – we haven't used it in the `\score` section. If we want the cello part to appear under the soprano part, we need to add

```
\new Staff \celloMusic
```

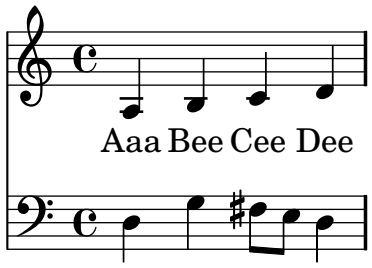
underneath the soprano stuff. We also need to add `<<` and `>>` around the music – that tells LilyPond that there's more than one thing (in this case, `Staff`) happening at once. The `\score` looks like this now

```
\score{
```

```
  <<
    <<
      \new Voice = "one" {
        \autoBeamOff
        \sopranoMusic
      }
      \new Lyrics \lyricsto "one" \sopranoLyrics
    >>
    \new Staff \celloMusic
  >>
  \layout { }
  \midi { }
}
```

This looks a bit messy; the indentation is messed up now. That is easily fixed. Here's the complete soprano and cello template.

```
\version "2.10.10"
sopranoMusic = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

sopranoLyrics = \lyricmode {
  Aaa Bee Cee Dee
}

celloMusic = \relative c {
  \clef bass
  \key c \major
  \time 4/4

  d4 g fis8 e d4
}

\score{
  <<
    <<
      \new Voice = "one" {
        \autoBeamOff
        \sopranoMusic
      }
      \new Lyrics \lyricsto "one" \sopranoLyrics
    >>
    \new Staff \celloMusic
  >>
  \layout { }
  \midi { }
}
```



3.2 How LilyPond files work

The LilyPond input format is quite free-form, giving experienced users a lot of flexibility to structure their files however they wish. However, this flexibility can make things confusing for new users. This section will explain some of this structure, but may gloss over some details in favor of simplicity. For a complete description of the input format, see Section 10.1.2 [File structure], page 234.

Most examples in this manual are little snippets – for example

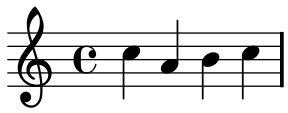
```
c4 a b c
```

As you are (hopefully) aware by now, this will not compile by itself. These examples are shorthand for complete examples. They all need at least curly braces to compile

```
{
  c4 a b c
}
```

Most examples also make use of the `\relative c'` (or `c''`) command. This is not necessary to merely compile the examples, but in most cases the output will look very odd if you omit the `\relative c'`.

```
\relative c'' {
  c4 a b c
}
```



Now we get to the only real stumbling block: LilyPond input in this form is actually *another* shorthand. Although it compiles and displays the correct output, it is shorthand for

```
\score {
  \relative c'' {
    c4 a b c
  }
}
```

A `\score` must begin with a single music expression. Remember that a music expression could be anything from a single note to a huge

```
{
  \new GrandStaff <<
    insert the whole score of a Wagner opera in here
  >>
}
```

Since everything is inside `{ ... }`, it counts as one music expression.

The `\score` can contain other things, such as

```
\score {
  { c'4 a b c' }
  \layout { }
  \midi { }
  \header { }
}
```

Some people put some of those commands outside the `\score` block – for example, `\header` is often placed above the `\score`. That's just another shorthand that LilyPond accepts.

Another great shorthand is the ability to define variables. All the templates use this

```
melody = \relative c' {
  c4 a b c
}

\score {
  { \melody }
}
```

When LilyPond looks at this file, it takes the value of `melody` (everything after the equals sign) and inserts it whenever it sees `\melody`. There's nothing special about the names – it could be `melody`, `global`, `pianorighthand`, or `foofoobarbaz`. You can use whatever variable names you want. For more details, see Section 4.2 [Saving typing with identifiers and functions], page 43.

For a complete definition of the input format, see Section 10.1.2 [File structure], page 234.

3.3 Score is a single musical expression

In the previous section, Section 3.2 [How LilyPond files work], page 37, we saw the general organization of LilyPond input files. But we seemed to skip over the most important part: how do we figure out what to write after `\score`?

We didn't skip over it at all. The big mystery is simply that there *is* no mystery. This line explains it all:

A `\score` *must begin with a single music expression.*

You may find it useful to review Section 2.3.1 [Music expressions explained], page 22. In that section, we saw how to build big music expressions from small pieces – we started from notes, then chords, etc. Now we're going to start from a big music expression and work our way down.

```
\score {
  {   % this brace begins the overall music expression
    \new GrandStaff <<
      insert the whole score of a Wagner opera in here
    >>
  }   % this brace ends the overall music expression
  \layout { }
}
```

A whole Wagner opera would easily double the length of this manual, so let's just do a singer and piano. We don't need a `GrandStaff` for this ensemble, so we shall remove it. We *do* need a singer and a piano, though.

```
\score {
  {
    <<
      \new Staff = "singer" <<
      >>
```

```
      \new PianoStaff = piano <<
      >>
    >>
  }
  \layout { }
}
```

Remember that we use << and >> to show simultaneous music. And we definitely want to show the vocal part and piano part at the same time!

```
\score {
  {
    <<
      \new Staff = "singer" <<
        \new Voice = "vocal" { }
      >>
      \new Lyrics \lyricsto vocal \new Lyrics { }
      \new PianoStaff = "piano" <<
        \new Staff = "upper" { }
        \new Staff = "lower" { }
      >>
    >>
  }
  \layout { }
}
```

Now we have a lot more details. We have the singer's staff: it contains a `Voice` (in LilyPond, this term refers to a set of notes, not necessarily vocal notes – for example, a violin generally plays one voice) and some lyrics. We also have a piano staff: it contains an upper staff (right hand) and a lower staff (left hand).

At this stage, we could start filling in notes. Inside the curly braces next to `\new Voice = vocal`, we could start writing

```
\relative c'' {
  a4 b c d
}
```

But if we did that, the `\score` section would get pretty long, and it would be harder to understand what was happening. So let's use identifiers (or variables) instead.

```
melody = { }
text = { }
upper = { }
lower = { }
\score {
  {
    <<
      \new Staff = "singer" <<
        \new Voice = "vocal" { \melody }
      >>
      \new Lyrics \lyricsto vocal \new Lyrics { \text }
      \new PianoStaff = "piano" <<
        \new Staff = "upper" { \upper }
        \new Staff = "lower" { \lower }
      >>
    >>
```

```
  }
  \layout { }
}
```

Remember that you can use almost any name you like. The limitations on identifier names are detailed in Section 10.1.2 [File structure], page 234.

When writing a `\score` section, or when reading one, just take it slowly and carefully. Start with the outer layer, then work on each smaller layer. It also really helps to be strict with indentation – make sure that each item on the same layer starts on the same horizontal position in your text editor!

3.4 An orchestral part

In orchestral music, all notes are printed twice. Once in a part for the musicians, and once in a full score for the conductor. Identifiers can be used to avoid double work. The music is entered once, and stored in a variable. The contents of that variable is then used to generate both the part and the full score.

It is convenient to define the notes in a special file. For example, suppose that the file '`horn-music.ly`' contains the following part of a horn/bassoon duo

```
hornNotes = \relative c {
  \time 2/4
  r4 f8 a cis4 f e d
}
```

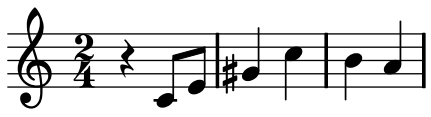
Then, an individual part is made by putting the following in a file

```
\include "horn-music.ly"
\header {
  instrument = "Horn in F"
}

{
 \transpose f c' \hornNotes
}
```

The line

```
\include "horn-music.ly"
```

substitutes the contents of '`horn-music.ly`' at this position in the file, so `hornNotes` is defined afterwards. The command `\transpose f c'` indicates that the argument, being `\hornNotes`, should be transposed by a fifth upwards. Sounding '`f`' is denoted by notated `c'`, which corresponds with the tuning of a normal French Horn in F. The transposition can be seen in the following output



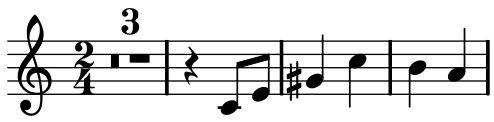
In ensemble pieces, one of the voices often does not play for many measures. This is denoted by a special rest, the multi-measure rest. It is entered with a capital '`R`' followed by a duration (`1` for a whole note, `2` for a half note, etc.). By multiplying the duration, longer rests can be constructed. For example, this rest takes 3 measures in 2/4 time

```
R2*3
```

When printing the part, multi-rests must be condensed. This is done by setting a run-time variable

```
\set Score.skipBars = ##t
```

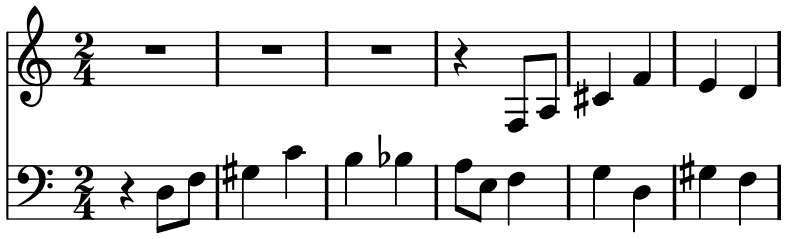
This command sets the property `skipBars` in the `Score` context to true (`##t`). Prepending the rest and this option to the music above, leads to the following result



The score is made by combining all of the music together. Assuming that the other voice is in `bassoonNotes` in the file '`bassoon-music.ly`', a score is made with

```
\include "bassoon-music.ly"
\include "horn-music.ly"

<<
  \new Staff \hornNotes
  \new Staff \bassoonNotes
>>
```

leading to



More in-depth information on preparing parts and scores can be found in the notation manual; see Section 8.3 [Orchestral music], page 195.

Setting run-time variables ('properties') is discussed in Section 9.2.3 [Changing context properties on the fly], page 220.

4 Working on LilyPond projects

This section explains how to solve or avoid certain common problems. If you have programming experience, many of these tips may seem obvious, but it is still advisable to read this chapter.

4.1 Suggestions for writing LilyPond files

Now you're ready to begin writing larger LilyPond files – not just the little examples in the tutorial, but whole pieces. But how should you go about doing it?

As long as LilyPond can understand your files and produces the output that you want, it doesn't matter what your files look like. However, there are a few other things to consider when writing lilypond files.

- What if you make a mistake? The structure of a lilypond file can make certain errors easier (or harder) to find.
- What if you want to share your files with somebody else? In fact, what if you want to alter your own files in a few years? Some lilypond files are understandable at first glance; other files may leave you scratching your head for an hour.
- What if you want to upgrade your lilypond file for use with a later version of lilypond? The input syntax changes occasionally as lilypond improves. Most changes can be done automatically with `convert-ly`, but some changes might require manual assistance. Lilypond files can be structured in order to be easier (or header) to update.

4.1.1 General suggestions

Here are a few suggestions that can help you to avoid or fix problems:

- **Include `\version` numbers in every file**. Note that all templates contain a `\version "2.10.10"` string. We highly recommend that you always include the `\version`, no matter how small your file is. Speaking from personal experience, it's quite frustrating to try to remember which version of LilyPond you were using a few years ago. `convert-ly` requires you to declare which version of LilyPond you used.
- **Include checks**: Section 6.2.5 [Bar check], page 68, Section 6.1.7 [Octave check], page 63 and Section 6.2.6 [Barnumber check], page 69. If you include checks every so often, then if you make a mistake, you can pinpoint it quicker. How often is "every so often"? It depends on the complexity of the music. For very simple music, perhaps just once or twice. For very complex music, perhaps every bar.
- **One bar per line of text**. If there is anything complicated, either in the music itself or in the output you desire, it's often good to write only one bar per line. Saving screen space by cramming eight bars per line just isn't worth it if you have to 'debug' your files.
- **Comment your files**. Use either bar numbers (every so often) or references to musical themes ("second theme in violins," "fourth variation", etc). You may not need comments when you're writing the piece for the first time, but if you want to go back to change something two or three years later, or if you pass the source over to a friend, it will be much more challenging to determine your intentions or how your file is structured if you didn't comment the file.
- **Indent your braces**. A lot of problems are caused by an imbalance in the number of `{` and `}`.
- **Explicity add durations** at the beginnings of sections and identifiers. If you specify `c4 d e` at the beginning of a phrase (instead of just `c d e`) you can save yourself some problems if you rearrange your music later.
- **Separate tweaks** from music definitions. See Section 4.2 [Saving typing with identifiers and functions], page 43 and Section 4.3 [Style sheets], page 45.

4.1.2 Typesetting existing music

If you are entering music from an existing score (i.e., typesetting a piece of existing sheet music),

- Enter one manuscript (the physical copy) system at a time (but still only one bar per line of text), and check each system when you finish it. You may use the `showLastLength` command to speed up processing – see Section 10.5 [Skipping corrected music], page 245.
- Define `mBreak = { \break }` and insert `\mBreak` in the input file whenever the manuscript has a line break. This makes it much easier to compare the LilyPond music to the original music. When you are finished proofreading your score, you may define `mBreak = { }` to remove all those line breaks. This will allow LilyPond to place line breaks wherever it feels are best.

4.1.3 Large projects

When working on a large project, having a clear structure to your lilypond files becomes vital.

- **Use an identifier for each voice**, with a minimum of structure inside the definition. The structure of the `\score` section is the most likely thing to change; the `violin` definition is extremely unlikely to change in a new version of LilyPond.

```
violin = \relative c'' {
g4 c'8. e16
}
...
\score {
  \new GrandStaff {
    \new Staff {
      \violin
    }
  }
}
```

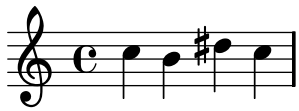
- **Separate tweaks from music definitions**. This point was made in Section 4.1.1 [General suggestions], page 42, but for large projects it is absolutely vital. We might need to change the definition of `fthenp`, but then we only need to do this once, and we can still avoid touching anything inside `violin`.

```
fthenp = _\markup{
  \dynamic f \italic \small { 2nd } \hspace #0.1 \dynamic p }
violin = \relative c'' {
g4\fthenp c'8. e16
}
```

4.2 Saving typing with identifiers and functions

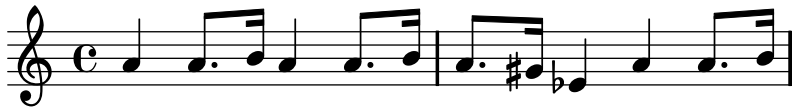
By this point, you've seen this kind of thing:

```
hornNotes = \relative c'' { c4 b dis c }
\score {
  {
    \hornNotes
  }
}
```



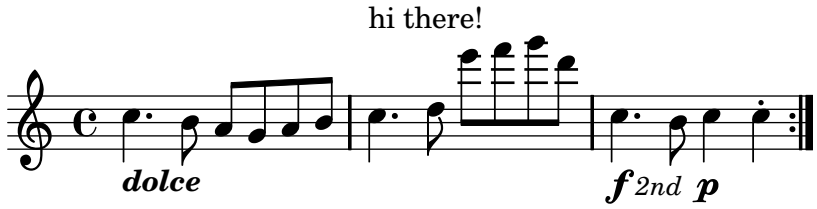
You may even realize that this could be useful in minimalist music:

```
fragA = \relative c'' { a4 a8. b16 }
fragB = \relative c'' { a8. gis16 ees4 }
violin = \new Staff { \fragA \fragA \fragB \fragA }
\score {
  {
    \violin
  }
}
```



However, you can also use these identifiers (also known as variables, macros, or (user-defined) command) for tweaks:

```
dolce = \markup{ \italic \bold dolce }
padText = { \once \override TextScript #'padding = #5.0 }
fthenp=_\markup{ \dynamic f \italic \small { 2nd } \hspace #0.1 \dynamic p }
violin = \relative c'' {
  \repeat volta 2 {
    c4._\dolce b8 a8 g a b |
    \padText
    c4.^"hi there!" d8 e' f g d |
    c,4.\fthenp b8 c4 c-. |
  }
}
\score {
  {
    \violin
  }
\layout{ragged-right=##t}
}
```



These identifiers are obviously useful for saving typing. But they're worth considering even if you only use them once – they reduce complexity. Let's look at the previous example without any identifiers. It's a lot harder to read, especially the last line.

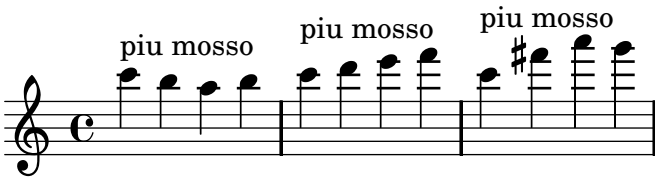
```
violin = \relative c'' {
  \repeat volta 2 {
    c4._\markup{ \italic \bold dolce } b8 a8 g a b |
    \once \override TextScript #'padding = #5.0
    c4.^"hi there!" d8 e' f g d |
    c,4.\markup{ \dynamic f \italic \small { 2nd }
```

```
        \hspace #0.1 \dynamic p } b8 c4 c-. |
  }
}
```

So far we've seen static substitution – when LilyPond sees `\padText`, it replaces it with the stuff that we've defined it to be (ie the stuff to the right of `padtext=`).

LilyPond can handle non-static substitution, too (you can think of these as functions).

```
padText =
#(define-music-function (parser location padding) (number?)
  #{
    \once \override TextScript #'padding = #$padding
  #})

\relative c''' {
  c4^"piu mosso" b a b
  \padText #1.8
  c4^"piu mosso" d e f
  \padText #2.6
  c4^"piu mosso" fis a g
}
```



Using identifiers is also a good way to reduce work if the LilyPond input syntax changes (see Section 4.4 [Updating old files], page 48). If you have a single definition (such as `\dolce`) for all your files (see Section 4.3 [Style sheets], page 45), then if the syntax changes, you only need to update your single `\dolce` definition, instead of making changes throughout every `.ly` file.

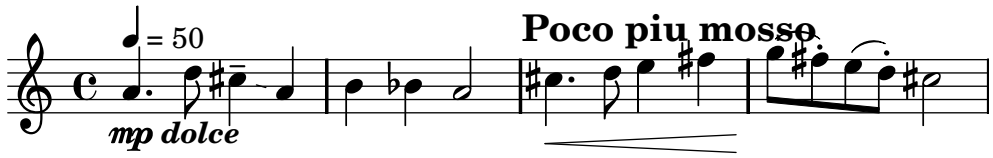
4.3 Style sheets

The output that LilyPond produces can be heavily modified; see Chapter 5 [Tweaking output], page 51 for details. But what if you have many files that you want to apply your tweaks to? Or what if you simply want to separate your tweaks from the actual music? This is quite easy to do.

Let's look at an example. Don't worry if you don't understand the parts with all the `#()`. This is explained in Section 5.6 [Advanced tweaks with Scheme], page 57.

```
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line(#:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'no-spacing-rods = ##t
  \mark \markup { \bold $markp }
#})

\relative c'' {
  \tempo 4=50
  a4.\mpdolce d8 cis4--\glissando a | b4 bes a2
```

```
  \tempoMark "Poco piu mosso"
  cis4.\< d8 e4 fis | g8(\! fis)-. e( d)-. cis2
}
```

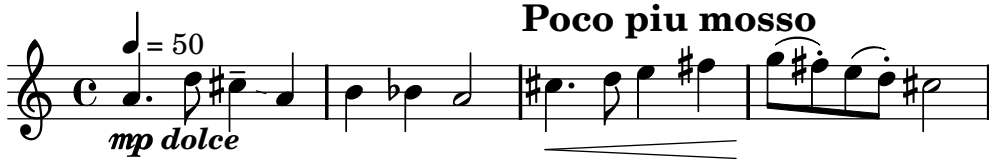


There are some problems with overlapping output; we'll fix those using the techniques in Section 5.1 [Moving objects], page 51. But let's also do something about the `mpdolce` and `tempoMark` definitions. They produce the output we desire, but we might want to use them in another piece. We could simply copy-and-paste them at the top of every file, but that's an annoyance. It also leaves those definitions in our music files, and I personally find all the `#()` somewhat ugly. Let's hide them in another file:

```
%%% save this to a file called "definitions.ly"
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line(#:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'no-spacing-rods = ##t
  \mark \markup { \bold $markp }
#})
```

Now let's modify our music (let's save this file as '`"music.ly"`').

```
\include "definitions.ly"

\relative c'' {
  \tempo 4=50
  a4.\mpdolce d8 cis4--\glissando a | b4 bes a2
  \once \override Score.RehearsalMark #'padding = #2.0
  \tempoMark "Poco piu mosso"
  cis4.\< d8 e4 fis | g8(\! fis)-. e( d)-. cis2
}
```

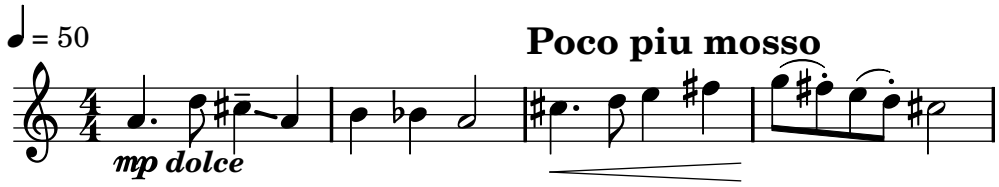


That looks better, but let's make a few changes. The glissando is hard to see, so let's make it thicker and closer to the noteheads. Let's put the metronome marking above the clef, instead of over the first note. And finally, my composition professor hates "C" time signatures, so we'd better make that "4/4" instead.

Don't change '`music.ly`', though. Replace our '`definitions.ly`' with this:

```
%%%  definitions.ly
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line( #:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
```

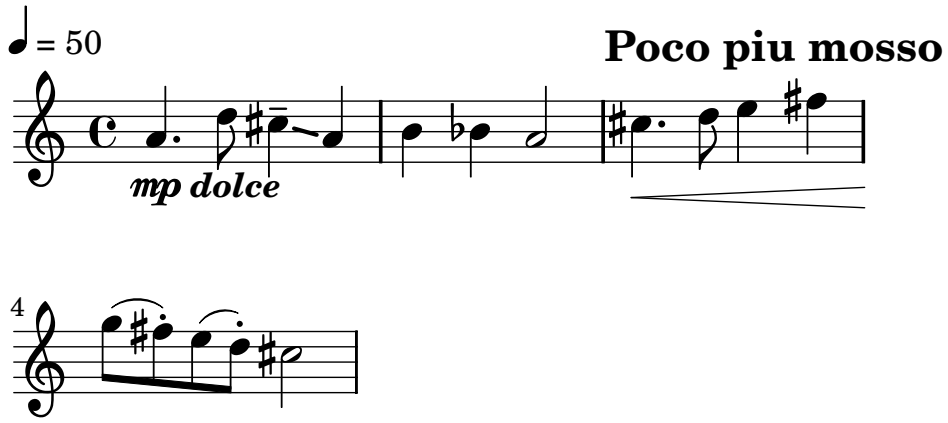
```
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'no-spacing-rods = ##t
  \mark \markup { \bold $markp }
#})

\layout{
  \context { \Score
    \override MetronomeMark #'extra-offset = #'(-9 . 0)
    \override MetronomeMark #'padding = #'3
  }
  \context { \Staff
    \override TimeSignature #'style = #'numbered
  }
  \context { \Voice
    \override Glissando #'thickness = #3
    \override Glissando #'gap = #0.1
  }
}
```



That looks nicer! But now suppose that I want to publish this piece. My composition professor doesn't like "C" time signatures, but I'm somewhat fond of them. Let's copy the current '`definitions.ly`' to '`web-publish.ly`' and modify that. Since this music is aimed at producing a pdf which will be displayed on the screen, we'll also increase the overall size of the output.

```
%%%  definitions.ly
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line( #:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'no-spacing-rods = ##t
  \mark \markup { \bold $markp }
#})

#(set-global-staff-size 23)
\layout{
  \context { \Score
    \override MetronomeMark #'extra-offset = #'(-9 . 0)
    \override MetronomeMark #'padding = #'3
  }
  \context { \Staff
  }
  \context { \Voice
    \override Glissando #'thickness = #3
```

```
    \override Glissando #'gap = #0.1
  }
}
```



Now in our music, I simply replace `\include "definitions.ly"` with `\include "web-publish.ly"`. Of course, we could make this even more convenient. We could make a '`definitions.ly`' file which contains only the definitions of `mpdolce` and `tempoMark`, a '`web-publish.ly`' file which contains only the `\layout` section listed above, and a '`university.ly`' file which contains only the tweaks to produce the output that my professor prefers. The top of '`music.ly`' would then look like this:

```
\include "definitions.ly"

%%%  Only uncomment one of these two lines!
\include "web-publish.ly"
%\include "university.ly"
```

This approach can be useful even if you are only producing one set of parts. I use half a dozen different "style sheet" files for my projects. I begin every music file with `\include "../global.ly"`, which contains

```
%%%   global.ly
\version "2.10.10"
#(ly:set-option 'point-and-click #f)
\include "../init/init-defs.ly"
\include "../init/init-layout.ly"
\include "../init/init-headers.ly"
\include "../init/init-paper.ly"
```

4.4 Updating old files

The LilyPond input syntax occasionally changes. As LilyPond itself improves, the syntax (input language) is modified accordingly. Sometimes these changes are made to make the input easier to read and write or sometimes the changes are made to accomodate new features of LilyPond.

LilyPond comes with a file that makes this updating easier: `convert-ly`. For details about how to run this program, see Section 13.3 [Updating files with convert-ly], page 290.

Unfortunately, `convert-ly` cannot handle all input changes. It takes care of simple search-and-replace changes (such as `raggedright` becoming `ragged-right`), but some changes are too complicated. The syntax changes that `convert-ly` cannot handle are listed in Section 13.3 [Updating files with convert-ly], page 290.

For example, in LilyPond 2.4 and earlier, accents and non-English letters were entered using LaTeX – for example, `"No\"el"` (this would print the French word for 'Christmas'). In LilyPond 2.6 and above, the special `"ë"` must be entered directly into the LilyPond file as an UTF-8 character. `convert-ly` cannot change all the LaTeX special characters into UTF-8 characters; you must manually update your old LilyPond files.

4.5 Troubleshooting (taking it all apart)

Sooner or later, you will write a file that LilyPond cannot compile. The messages that LilyPond gives may help you find the error, but in many cases you need to do some investigation to determine the source of the problem.

The most powerful tools for this purpose are the single line comment (indicated by `%`) and the block comment (indicated by `%{ ... %}`). If you don't know where a problem is, start commenting out huge portions of your input file. After you comment out a section, try compiling the file again. If it works, then the problem must exist in the portion you just commented. If it doesn't work, then keep on commenting out material until you have something that works.

In an extreme case, you might end up with only

```
\score {
  <<
    % \melody
    % \harmony
    % \bass
  >>
  \layout{}
}
```

(in other words, a file without any music)

If that happens, don't give up. Uncomment a bit – say, the bass part – and see if it works. If it doesn't work, then comment out all of the bass music (but leave `\bass` in the `\score` uncommented.

```
bass = \relative c' {
%{
  c4 c c c
  d d d d
%}
}
```

Now start slowly uncommenting more and more of the `bass` part until you find the problem line.

Another very useful debugging technique is constructing Section 4.6 [Minimal examples], page 49.

4.6 Minimal examples

A minimal example is an example which is as small as possible. These examples are much easier to understand than long examples. Minimal examples are used for

- Bug reports
- Sending a help request to mailists
- Adding an example to the LilyPond Snippet Repository

To construct an example which is as small as possible, the rule is quite simple: remove anything which is not necessary. When trying to remove unnecessary parts of a file, it is a very

good idea to comment out lines instead of deleting them. That way, if you discover that you actually *do* need some lines, you can uncomment them, instead of typing them in from scratch.

There are two exceptions to the "as small as possible" rule:

- Include the `\version` number.
- If possible, use `\paper{ ragged-right=##t }` at the top of your example.

The whole point of a minimal example is to make it easy to read:

- Avoid using complicated notes, keys, or time signatures, unless you wish to demonstrate something is about the behavior of those items.
- Do not use `\override` commands unless that is the point of the example.

5 Tweaking output

This chapter discusses how to modify output. LilyPond is extremely configurable; virtually every fragment of output may be changed.

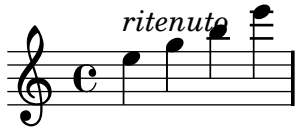
5.1 Moving objects

This may come as a surprise, but LilyPond is not perfect. Some notation elements can overlap. This is unfortunate, but (in most cases) is easily solved.

```
e4^\markup{ \italic ritenuto } g b e
```

The easiest solution is to increase the distance between the object (in this case text, but it could easily be fingerings or dynamics instead) and the note. In LilyPond, this is called the `padding` property; it is measured in staff spaces. For most objects, this value is around 1.0 or less (it varies with each object). We want to increase it, so let's try 1.5

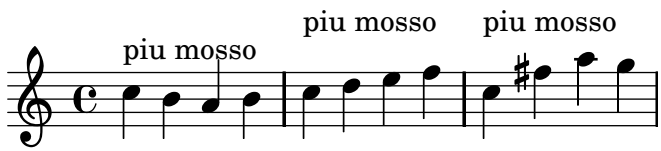
```
\once \override TextScript #'padding = #1.5
e4^\markup{ \italic ritenuto } g b e
```

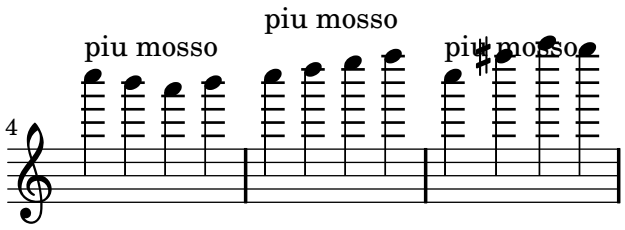


That looks better, but it isn't quite big enough. After experimenting with a few values, we think 2.3 is the best number in this case. However, this number is merely the result of experimentation and my personal taste in notation. Try the above example with 2.3... but also try higher (and lower) numbers. Which do you think looks the best?

The `staff-padding` property is closely related. `padding` controls the minimum amount of space between an object and the nearest other object (generally the note or the staff lines); `staff-padding` controls the minimum amount of space between an object and the staff. This is a subtle difference, but you can see the behavior here.

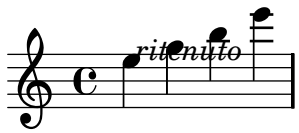
```
c4^"piu mosso" b a b
\once \override TextScript #'padding = #2.6
c4^"piu mosso" d e f
\once \override TextScript #'staff-padding = #2.6
c4^"piu mosso" fis a g
\break
c'4^"piu mosso" b a b
\once \override TextScript #'padding = #2.6
c4^"piu mosso" d e f
\once \override TextScript #'staff-padding = #2.6
c4^"piu mosso" fis a g
```





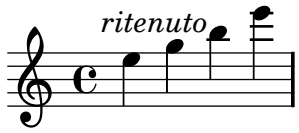
Another solution gives us complete control over placing the object – we can move it horizontally or vertically. This is done with the `extra-offset` property. It is slightly more complicated and can cause other problems. When we move objects with `extra-offset`, the movement is done after LilyPond has placed all other objects. This means that the result can overlap with other objects.

```
\once \override TextScript #'extra-offset = #'( 1.0 . -1.0 )
e4^\markup{ \italic ritenuto } g b e
```



With `extra-offset`, the first number controls the horizontal movement (left is negative); the second number controls the vertical movement (up is positive). After a bit of experimenting, we decided that these values look good

```
\once \override TextScript #'extra-offset = #'( -1.6 . 1.0 )
e4^\markup{ \italic ritenuto } g b e
```



Again, these numbers are simply the result of a few experiments and looking at the output. You might prefer the text to be slightly higher, or to the left, or whatever. Try it and look at the result!

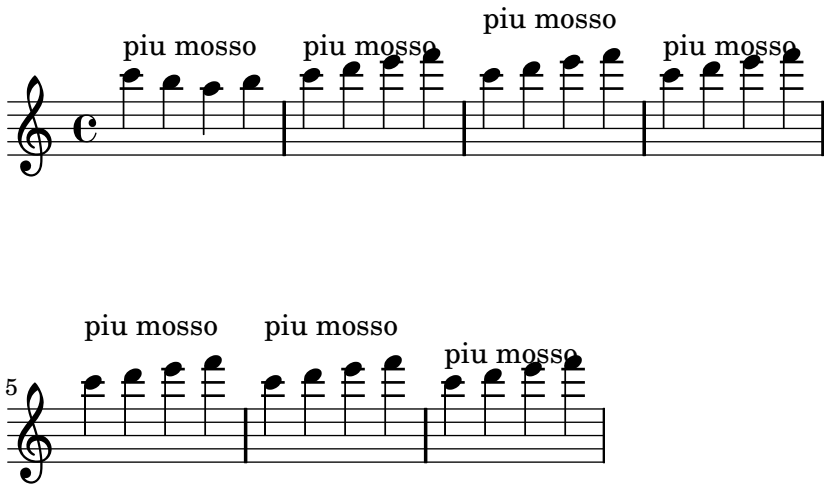
One final warning: in this section, we used

```
\once \override TextScript ...
```

This tweaks the display of text for the next note. If the note has no text, this tweak does nothing (and does **not** wait until the next bit of text). To change the behavior of everything after the command, omit the `\once`. To stop this tweak, use a `\revert`. This is explained in depth in Section 9.3 [The \override command], page 228.

```
c4^"piu mosso" b
\once \override TextScript #'padding = #2.6
  a4 b
c4^"piu mosso" d e f
\once \override TextScript #'padding = #2.6
c4^"piu mosso" d e f
c4^"piu mosso" d e f
\break
\override TextScript #'padding = #2.6
c4^"piu mosso" d e f
c4^"piu mosso" d e f
\revert TextScript #'padding
```

```
c4^"piu mosso" d e f
```



See also

This manual: Section 9.3 [The \override command], page 228, Section 5.3 [Common tweaks], page 54.

5.2 Fixing overlapping notation

In Section 5.1 [Moving objects], page 51, we saw how to move a `TextScript` object. The same mechanism can be used to move other types of objects; simply replace `TextScript` with the name of another object.

To find the object name, look at the "**see also**" at bottom of the relevant documentation page. For example, at the bottom of Section 6.6.3 [Dynamics], page 98, we see

> ### See also
>
> Program reference: `DynamicText`, `Hairpin`. Vertical positioning of these symbols is handled by `DynamicLineSpanner`.

So to move dynamics around vertically, we use

```
\override DynamicLineSpanner #'padding = #2.0
```

We cannot list every object, but here is a list of the most common objects.

Object type	**Object name**
Dynamics (vertically)	`DynamicLineSpanner`
Dynamics (horizontally)	`DynamicText`
Ties	`Tie`
Slurs	`Slur`
Articulations	`Script`
Fingerings	`Fingering`
Text e.g. `^"text"`	`TextScript`
Rehearsal / Text marks	`RehearsalMark`

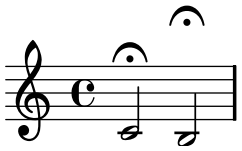
5.3 Common tweaks

Some overrides are so common that predefined commands are provided as short-cuts, such as `\slurUp` and `\stemDown`. These commands are described in the Notation Reference under the appropriate sections.

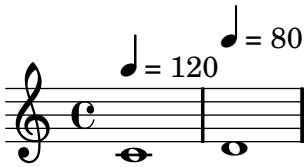
The complete list of modifications available for each type of object (like slurs or beams) are documented in the Program Reference. However, many layout objects share properties which can be used to apply generic tweaks.

- The `padding` property can be set to increase (or decrease) the distance between symbols that are printed above or below notes. This applies to all objects with `side-position-interface`.

```
c2\fermata
\override Script #'padding = #3
b2\fermata
```



```
% This will not work, see below:
\override MetronomeMark #'padding = #3
\tempo 4=120
c1
% This works:
\override Score.MetronomeMark #'padding = #3
\tempo 4=80
d1
```

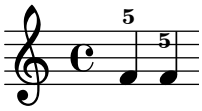


Note in the second example how important it is to figure out what context handles a certain object. Since the `MetronomeMark` object is handled in the `Score` context, property changes in the `Voice` context will not be noticed. For more details, see Section 9.3.1 [Constructing a tweak], page 228.

- The `extra-offset` property moves objects around in the output; it requires a pair of numbers. The first number controls horizontal movement; a positive number will move the object to the right. The second number controls vertical movement; a positive number will move it higher. The `extra-offset` property is a low-level feature: the formatting engine is completely oblivious to these offsets.

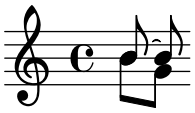
  In the following example, the second fingering is moved a little to the left, and 1.8 staff space downwards:

```
\stemUp
f-5
\once \override Fingering
    #'extra-offset = #'(-0.3 . -1.8)
f-5
```



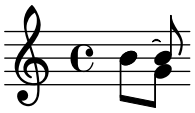
- Setting the `transparent` property will cause an object to be printed in 'invisible ink': the object is not printed, but all its other behavior is retained. The object still takes up space, it takes part in collisions, and slurs, ties, and beams can be attached to it.

  The following example demonstrates how to connect different voices using ties. Normally, ties only connect two notes in the same voice. By introducing a tie in a different voice,



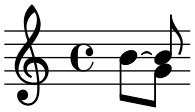
  and blanking the first up-stem in that voice, the tie appears to cross voices:

```
<< {
  \once \override Stem #'transparent = ##t
  b8~ b8\noBeam
} \\ {
  b[ g8]
} >>
```



  To make sure that the just blanked stem doesn't squeeze the too much tie, we also lengthen the stem, by setting the `length` to `8`,

```
<< {
  \once \override Stem #'transparent = ##t
  \once \override Stem #'length = #8
  b8~ b8\noBeam
} \\ {
  b[ g8]
} >>
```



Distances in LilyPond are measured in staff-spaces, while most thickness properties are measured in line-thickness. Some properties are different; for example, the thickness of beams are measured in staff-spaces. For more information, see the relevant portion of the program reference.

5.4 Default files

The Program Reference documentation contains a lot of information about LilyPond, but even more information can be gathered from looking at the internal LilyPond files.

Some default settings (such as the definitions for `\header{}`s) are stored as `.ly` files. Other settings (such as the definitions of markup commands) are stored as `.scm` (Scheme) files. Further explanation is outside the scope of this manual; users should be warned that a substantial amount of technical knowledge or time is required to understand these files.

- Linux: '*installdir*/lilypond/usr/share/lilypond/current/'
- OSX: '*installdir*/LilyPond.app/Contents/Resources/share/lilypond/current/'. To access this, either `cd` into this directory from the Terminal, or control-click on the LilyPond application and select "Show Package Contents".
- Windows: '*installdir*/LilyPond/usr/share/lilypond/current/'

The 'ly/' and 'scm/' directories will be of particular interest. Files such as 'ly/property-init.ly' and 'ly/declarations-init.ly' define all the common tweaks.

5.5 Fitting music onto fewer pages

Sometimes you can end up with one or two staves on a second (or third, or fourth...) page. This is annoying, especially if you look at previous pages and it looks like there is plenty of room left on those.

When investigating layout issues, `annotate-spacing` is an invaluable tool. This command prints the values of various layout spacing commands; see Section 11.6 [Displaying spacing], page 270 for more details. From the output of `annotate-spacing`, we can see which margins we may wish to alter.

Other than margins, there are a few other options to save space:

- You may tell LilyPond to place systems as close together as possible (to fit as many systems as possible onto a page), but then to space those systems out so that there is no blank space at the bottom of the page.

```
\paper {
  between-system-padding = #0.1
  between-system-space = #0.1
  ragged-last-bottom = ##f
  ragged-bottom = ##f
}
```

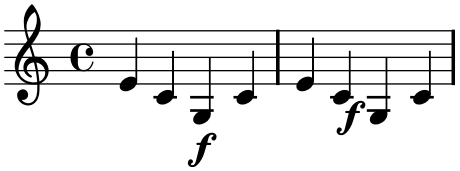
- You may force the number of systems (i.e., if LilyPond wants to typeset some music with 11 systems, you could force it to use 10).

```
\paper {
  system-count = #10
}
```

- Avoid (or reduce) objects which increase the vertical size of a system. For example, volta repeats (or alternate repeats) require extra space. If these repeats are spread over two systems, they will take up more space than one system with the volta repeats and another system without.

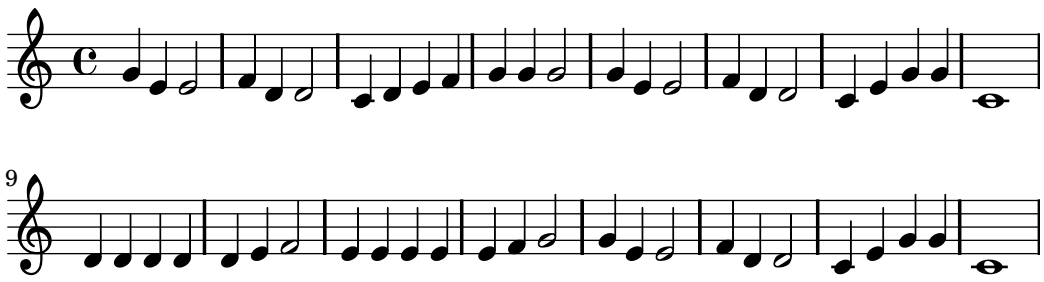
  Another example is moving dynamics which "stick out" of a system.

```
\relative c' {
  e4 c g\f c
  \override DynamicLineSpanner #'padding = #-1.8
  \override DynamicText #'extra-offset = #'( -2.1 . 0)
  e4 c g\f c
}
```



- Alter the horizontal spacing via `SpacingSpanner`. See Section 11.5.3 [Changing horizontal spacing], page 267 for more details.

```
\score {
  \relative c'' {
    g4 e e2 | f4 d d2 | c4 d e f | g4 g g2 |
    g4 e e2 | f4 d d2 | c4 e g g | c,1 |
    d4 d d d | d4 e f2 | e4 e e e | e4 f g2 |
    g4 e e2 | f4 d d2 | c4 e g g | c,1 |
  }
  \layout {
    \context {
      \Score
      \override SpacingSpanner
                #'base-shortest-duration = #(ly:make-moment 1 4)
    }
  }
}
```

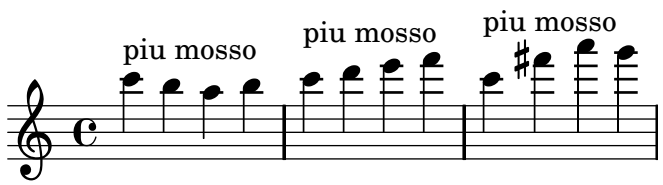


5.6 Advanced tweaks with Scheme

We have seen how LilyPond output can be heavily modified using commands like `\override TextScript #'extra-offset = ( 1 . -1)`. But we have even more power if we use Scheme. For a full explantion of this, see the Appendix B [Scheme tutorial], page 310 and Chapter 12 [Interfaces for programmers], page 271.

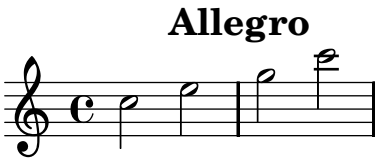
We can use Scheme to simply `\override` commands,

```
padText = #(define-music-function (parser location padding) (number?)
#{
  \once \override TextScript #'padding = #$padding
#})

\relative c''' {
  c4^"piu mosso" b a b
  \padText #1.8
  c4^"piu mosso" d e f
  \padText #2.6
  c4^"piu mosso" fis a g
}
```



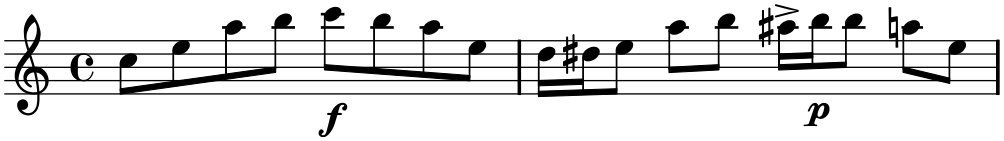
We can use it to create new commands,

```
tempoMark = #(define-music-function (parser location padding marktext)
                                    (number? string?)
#{
  \once \override Score . RehearsalMark #'padding = $padding
  \once \override Score . RehearsalMark #'no-spacing-rods = ##t
  \mark \markup { \bold $marktext }
#})

\relative c'' {
  c2 e
  \tempoMark #3.0 #"Allegro"
  g c
}
```



Even music expressions can be passed in.

```
pattern = #(define-music-function (parser location x y) (ly:music? ly:music?)
#{
  $x e8 a b $y b a e
#})

\relative c''{
  \pattern c8 c8\f
  \pattern {d16 dis} { ais16-> b\p }
}
```



5.7 Avoiding tweaks with slower processing

LilyPond can perform extra checks while it processes files. These commands will take extra time, but the result may require fewer manual tweaks.

```
%%  makes sure text scripts and lyrics are within the paper margins
\override Score.PaperColumn #'keep-inside-line = ##t
```

6 Basic notation

This chapter explains how to use basic notation features.

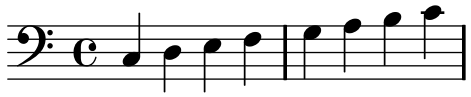
6.1 Pitches

This section discusses how to specify the pitch of notes.

6.1.1 Normal pitches

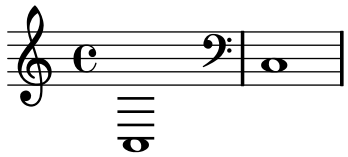
A pitch name is specified using lowercase letters `a` through `g`. An ascending C-major scale is engraved with

```
\clef bass
c d e f g a b c'
```



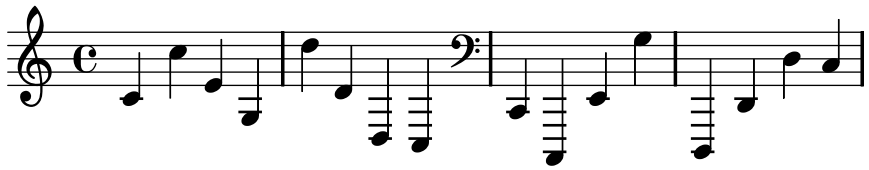
The note name `c` is engraved one octave below middle C.

```
\clef treble
c1
\clef bass
c1
```



The optional octave specification takes the form of a series of single quote (‘'’) characters or a series of comma (‘,’) characters. Each `'` raises the pitch by one octave; each `,` lowers the pitch by an octave.

```
\clef treble
c' c'' e' g d'' d' d c
\clef bass
c, c,, e, g d,, d, d c
```

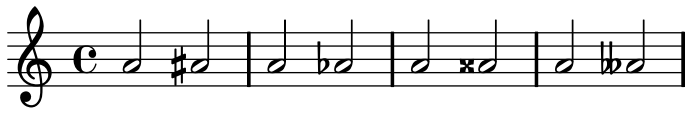


An alternate method may be used to declare which octave to engrave a pitch; this method does not require as many octave specifications (`'` and `,`). See Section 6.1.6 [Relative octaves], page 62.

6.1.2 Accidentals

A sharp is formed by adding `-is` to the end of a pitch name and a flat is formed by adding `-es`. Double sharps and double flats are obtained by adding `-isis` or `-eses` to a note name.

```
a2 ais a aes
a2 aisis a aeses
```

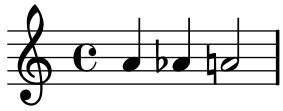


These are the Dutch note names. In Dutch, `aes` is contracted to `as`, but both forms are accepted. Similarly, both `es` and `ees` are accepted

```
a2 as e es
```

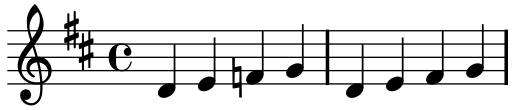
A natural will cancel the effect of an accidental or key signature. However, naturals are not encoded into the note name syntax with a suffix; a natural pitch is shown as a simple note name

```
a4 aes a2
```



The input `d e f` is interpreted as "print a D-natural, E-natural, and an F-natural," regardless of the key signature. For more information about the distinction between musical content and the presentation of that content, see Section 2.2.2 [Accidentals and key signatures], page 17.

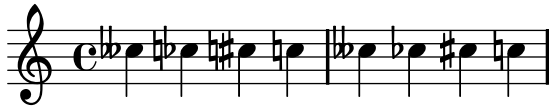
```
\key d \major
d e f g
d e fis g
```



Commonly tweaked properties

In accordance with standard typesetting rules, a natural sign is printed before a sharp or flat if a previous accidental needs to be cancelled. To change this behavior, use `\set Staff.extraNatural = ##f`

```
ceses4 ces cis c
\set Staff.extraNatural = ##f
ceses4 ces cis c
```



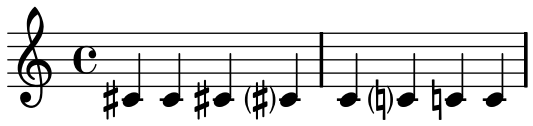
See also

Program reference: `LedgerLineSpanner`, `NoteHead`.

6.1.3 Cautionary accidentals

Normally accidentals are printed automatically, but you may also print them manually. A reminder accidental can be forced by adding an exclamation mark `!` after the pitch. A cautionary accidental (i.e., an accidental within parentheses) can be obtained by adding the question mark '?' after the pitch. These extra accidentals can be used to produce natural signs, too.

```
cis cis cis! cis? c c? c! c
```



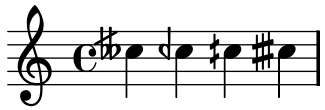
See also

The automatic production of accidentals can be tuned in many ways. For more information, see Section 9.1.1 [Automatic accidentals], page 213.

6.1.4 Micro tones

Half-flats and half-sharps are formed by adding `-eh` and `-ih`; the following is a series of Cs with increasing pitches

```
\set Staff.extraNatural = ##f
ceseh ceh cih cisih
```



Micro tones are also exported to the MIDI file.

Bugs

There are no generally accepted standards for denoting three-quarter flats, so LilyPond's symbol does not conform to any standard.

6.1.5 Note names in other languages

There are predefined sets of note names for various other languages. To use them, include the language specific init file. For example, add `\include "english.ly"` to the top of the input file. The available language files and the note names they define are

	Note Names								sharp	flat
nederlands.ly	c	d	e	f	g	a	bes	b	-is	-es
english.ly	c	d	e	f	g	a	bf	b	-s/-sharp	-f/-flat
									-x (double)	
deutsch.ly	c	d	e	f	g	a	b	h	-is	-es
norsk.ly	c	d	e	f	g	a	b	h	-iss/-is	-ess/-es
svenska.ly	c	d	e	f	g	a	b	h	-iss	-ess
italiano.ly	do	re	mi	fa	sol	la	sib	si	-d	-b
catalan.ly	do	re	mi	fa	sol	la	sib	si	-d/-s	-b
espanol.ly	do	re	mi	fa	sol	la	sib	si	-s	-b

6.1.6 Relative octaves

Octaves are specified by adding ' and , to pitch names. When you copy existing music, it is easy to accidentally put a pitch in the wrong octave and hard to find such an error. The relative octave mode prevents these errors by making the mistakes much larger: a single error puts the rest of the piece off by one octave

```
\relative startpitch musicexpr
```

or

```
\relative musicexpr
```

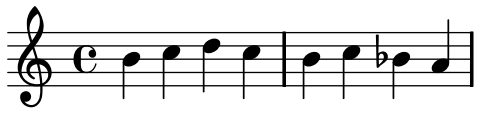
c' is used as the default if no starting pitch is defined.

The octave of notes that appear in *musicexpr* are calculated as follows: if no octave changing marks are used, the basic interval between this and the last note is always taken to be a fourth or less. This distance is determined without regarding alterations; a fisis following a ceses will be put above the ceses. In other words, a doubly-augmented fourth is considered a smaller interval than a diminished fifth, even though the doubly-augmented fourth spans seven semitones while the diminished fifth only spans six semitones.

The octave changing marks ' and , can be added to raise or lower the pitch by an extra octave. Upon entering relative mode, an absolute starting pitch can be specified that will act as the predecessor of the first note of *musicexpr*. If no starting pitch is specified, then middle C is used as a start.

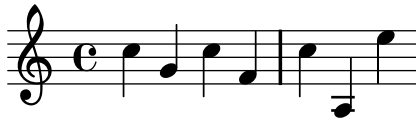
Here is the relative mode shown in action

```
\relative c'' {
  b c d c b c bes a
}
```



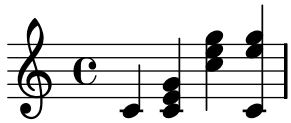
Octave changing marks are used for intervals greater than a fourth

```
\relative c'' {
  c g c f, c' a, e''
}
```



If the preceding item is a chord, the first note of the chord is used to determine the first note of the next chord

```
\relative c' {
  c <c e g>
  <c' e g>
  <c, e' g>
}
```



The pitch after `\relative` contains a note name.

The relative conversion will not affect `\transpose`, `\chordmode` or `\relative` sections in its argument. To use relative within transposed music, an additional `\relative` must be placed inside `\transpose`.

6.1.7 Octave check

Octave checks make octave errors easier to correct: a note may be followed by =*quotes* which indicates what its absolute octave should be. In the following example,

```
\relative c'' { c='' b=' d,='' }
```

the `d` will generate a warning, because a `d''` is expected (because `b'` to `d''` is only a third), but a `d'` is found. In the output, the octave is corrected to be a `d''` and the next note is calculated relative to `d''` instead of `d'`.

There is also an octave check that produces no visible output. The syntax

```
\octave pitch
```

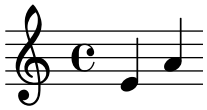
This checks that *pitch* (without quotes) yields *pitch* (with quotes) in `\relative` mode compared to the note given in the `\relative` command. If not, a warning is printed, and the octave is corrected. The *pitch* is not printed as a note.

In the example below, the first check passes without incident, since the `e` (in `relative` mode) is within a fifth of `a'`. However, the second check produces a warning, since the `e` is not within a fifth of `b'`. The warning message is printed, and the octave is adjusted so that the following notes are in the correct octave once again.

```
\relative c' {
  e
  \octave a'
  \octave b'
}
```

The octave of a note following an octave check is determined with respect to the note preceding it. In the next fragment, the last note is an `a'`, above middle C. That means that the `\octave` check passes successfully, so the check could be deleted without changing the output of the piece.

```
\relative c' {
  e
  \octave b
  a
}
```



6.1.8 Transpose

A music expression can be transposed with `\transpose`. The syntax is

```
\transpose from to musicexpr
```

This means that *musicexpr* is transposed by the interval between the pitches *from* and *to*: any note with pitch `from` is changed to `to`.

For example, consider a piece written in the key of D-major. If this piece is a little too low for its performer, it can be transposed up to E-major with

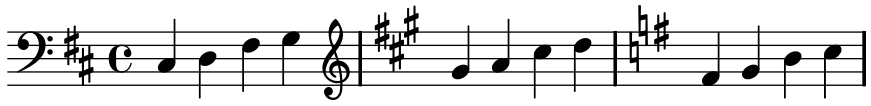
```
\transpose d e ...
```

Consider a part written for violin (a C instrument). If this part is to be played on the A clarinet (for which an A is notated as a C, and which sounds a minor third lower than notated), the following transposition will produce the appropriate part

```
\transpose a c ...
```

`\transpose` distinguishes between enharmonic pitches: both `\transpose c cis` or `\transpose c des` will transpose up half a tone. The first version will print sharps and the second version will print flats

```
mus = { \key d \major cis d fis g }
\new Staff {
  \clef "F" \mus
  \clef "G"
  \transpose c g' \mus
  \transpose c f' \mus
}
```



`\transpose` may also be used to input written notes for a transposing instrument. Pitches are normally entered into LilyPond in C (or "concert pitch"), but they may be entered in another key. For example, when entering music for a B-flat trumpet which begins on concert D, one would write

```
\transpose c bes { e4 ... }
```

To print this music in B-flat again (i.e., producing a trumpet part, instead of a concert pitch conductor's score) you would wrap the existing music with another `transpose`

```
\transpose bes c { \transpose c bes { e4 ... } }
```

See also

Program reference: `TransposedMusic`.

Example: '`input/test/smart-transpose.ly`'.

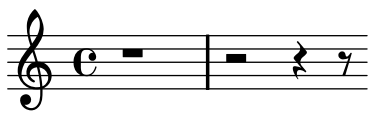
Bugs

If you want to use both `\transpose` and `\relative`, you must put `\transpose` outside of `\relative`, since `\relative` will have no effect on music that appears inside a `\transpose`.

6.1.9 Rests

Rests are entered like notes with the note name `r`

```
r1 r2 r4 r8
```



Whole bar rests, centered in middle of the bar, must be done with multi-measure rests. They can be used for a single bar as well as many bars, and are discussed in Section 8.2.1 [Multi measure rests], page 184.

To explicitly specify a rest's vertical position, write a note followed by `\rest`. A rest will be placed in the position where the note would appear,

```
a'4\rest d'4\rest
```

This makes manual formatting of polyphonic music much easier, since the automatic rest collision formatter will leave these rests alone.

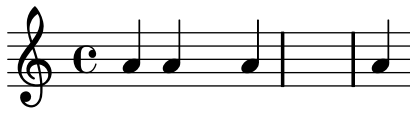
See also

Program reference: `Rest`.

6.1.10 Skips

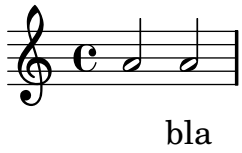
An invisible rest (also called a 'skip') can be entered like a note with note name '`s`' or with `\skip` *`duration`*

```
a4 a4 s4 a4 \skip 1 a4
```



The `s` syntax is only available in note mode and chord mode. In other situations, for example, when entering lyrics, you should use the `\skip` command

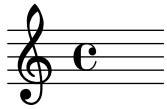
```
<<
  \relative { a'2 a2 }
  \new Lyrics \lyricmode { \skip 2 bla2 }
>>
```



The skip command is merely an empty musical placeholder. It does not produce any output, not even transparent output.

The `s` skip command does create `Staff` and `Voice` when necessary, similar to note and rest commands. For example, the following results in an empty staff.

```
{ s4 }
```



The fragment `{ \skip 4 }` would produce an empty page.

See also

Program reference: `SkipMusic`.

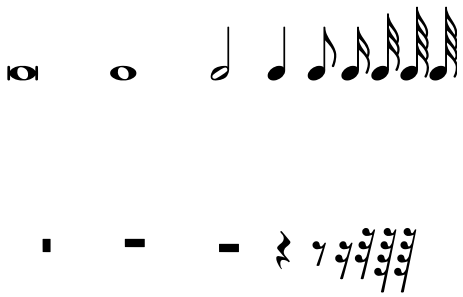
6.2 Rhythms

This section discusses rhythms, durations, and bars.

6.2.1 Durations

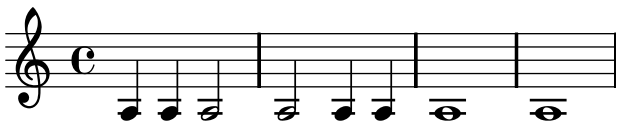
In Note, Chord, and Lyrics mode, durations are designated by numbers and dots: durations are entered as their reciprocal values. For example, a quarter note is entered using a `4` (since it is a 1/4 note), while a half note is entered using a `2` (since it is a 1/2 note). For notes longer than a whole you must use the `\longa` and `\breve` commands

```
c'\breve
c'1 c'2 c'4 c'8 c'16 c'32 c'64 c'64
r\longa r\breve
r1 r2 r4 r8 r16 r32 r64 r64
```



If the duration is omitted then it is set to the previously entered duration. The default for the first note is a quarter note.

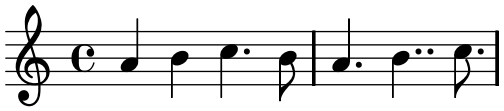
```
{ a a a2 a a4 a a1 a }
```



6.2.2 Augmentation dots

To obtain dotted note lengths, simply add a dot (‘.’) to the number. Double-dotted notes are produced in a similar way.

```
a'4 b' c''4. b'8 a'4. b'4.. c''8.
```



Predefined commands

Dots are normally moved up to avoid staff lines, except in polyphonic situations. The following commands may be used to force a particular direction manually

`\dotsUp`, `\dotsDown`, `\dotsNeutral`.

See also

Program reference: `Dots`, and `DotColumn`.

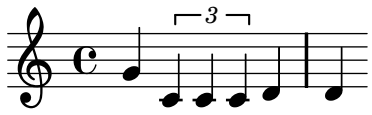
6.2.3 Tuplets

Tuplets are made out of a music expression by multiplying all durations with a fraction

```
\times fraction musicexpr
```

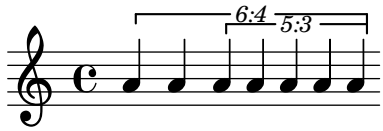
The duration of *musicexpr* will be multiplied by the fraction. The fraction's denominator will be printed over the notes, optionally with a bracket. The most common tuplet is the triplet in which 3 notes have the length of 2, so the notes are 2/3 of their written length

```
g'4 \times 2/3 {c'4 c' c'} d'4 d'4
```



Tuplets may be nested, for example,

```
\override TupletNumber #'text = #tuplet-number::calc-fraction-text
\times 4/6 {
  a4 a
  \times 3/5 { a a a a a }
}
```



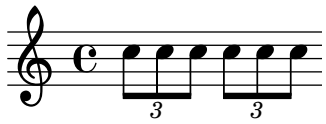
Predefined commands

`\tupletUp`, `\tupletDown`, `\tupletNeutral`.

Commonly tweaked properties

The property `tupletSpannerDuration` specifies how long each bracket should last. With this, you can make lots of tuplets while typing `\times` only once, thus saving lots of typing. In the next example, there are two triplets shown, while `\times` was only used once

```
\set tupletSpannerDuration = #(ly:make-moment 1 4)
\times 2/3 { c8 c c c c c }
```

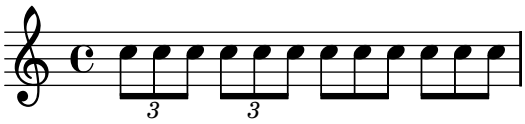


For more information about `make-moment`, see Section 8.4.2 [Time administration], page 203.

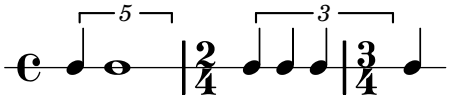
The format of the number is determined by the property `text` in `TupletNumber`. The default prints only the denominator, but if it is set to the function `tuplet-number::calc-fraction-text`, *num*:*den* will be printed instead.

To avoid printing tuplet numbers, use

```
\times 2/3 { c8 c c } \times 2/3 { c8 c c }
\override TupletNumber #'transparent = ##t
\times 2/3 { c8 c c } \times 2/3 { c8 c c }
```



Tuplet brackets can be made to run to prefatory matter or the next note



See also

Program reference: `TupletBracket`, `TupletNumber`, and `TimeScaledMusic`.

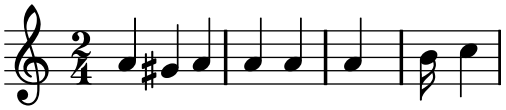
Examples: ‘`input/regression/tuplet-nest.ly`’.

6.2.4 Scaling durations

You can alter the length of duration by a fraction *N/M* appending ‘**N/M*’ (or ‘**N*’ if *M=1*). This will not affect the appearance of the notes or rests produced.

In the following example, the first three notes take up exactly two beats, but no triplet bracket is printed.

```
\time 2/4
a4*2/3 gis4*2/3 a4*2/3
a4 a4 a4*2
b16*4 c4
```



See also

This manual: Section 6.2.3 [Tuplets], page 67

6.2.5 Bar check

Bar checks help detect errors in the durations. A bar check is entered using the bar symbol, ‘`|`’. Whenever it is encountered during interpretation, it should fall on a measure boundary. If it does not, a warning is printed. In the next example, the second bar check will signal an error

```
\time 3/4 c2 e4 | g2 |
```

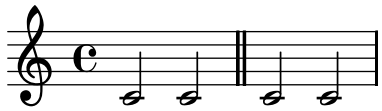
Bar checks can also be used in lyrics, for example

```
\lyricmode {
  \time 2/4
  Twin -- kle | Twin -- kle
}
```

Failed bar checks are caused by entering incorrect durations. Incorrect durations often completely garble up the score, especially if the score is polyphonic, so a good place to start correcting input is by scanning for failed bar checks and incorrect durations.

It is also possible to redefine the meaning of `|`. This is done by assigning a music expression to `pipeSymbol`,

```
pipeSymbol = \bar "||"

{ c'2 c' | c'2 c' }
```



6.2.6 Barnumber check

When copying large pieces of music, it can be helpful to check that the LilyPond bar number corresponds to the original that you are entering from. This can be checked with `\barNumberCheck`, for example,

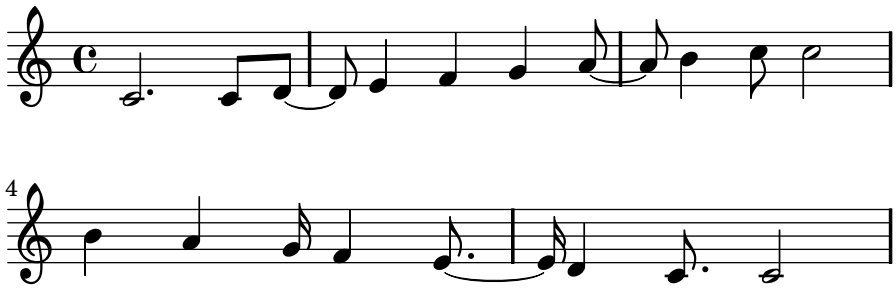
```
\barNumberCheck #123
```

will print a warning if the `currentBarNumber` is not 123 when it is processed.

6.2.7 Automatic note splitting

Long notes can be converted automatically to tied notes. This is done by replacing the `Note_heads_engraver` by the `Completion_heads_engraver`. In the following examples, notes crossing the bar line are split and tied.

```
\new Voice \with {
  \remove "Note_heads_engraver"
  \consists "Completion_heads_engraver"
} {
  c2. c8 d4 e f g a b c8 c2 b4 a g16 f4 e d c8. c2
}
```



This engraver splits all running notes at the bar line, and inserts ties. One of its uses is to debug complex scores: if the measures are not entirely filled, then the ties exactly show how much each measure is off.

If you want to allow line breaking on the bar lines where `Completion_heads_engraver` splits notes, you must also remove `Forbid_line_breaks_engraver`.

Bugs

Not all durations (especially those containing tuplets) can be represented exactly with normal notes and dots, but the engraver will not insert tuplets.

`Completion_heads_engraver` only affects notes; it does not split rests.

See also

Examples: ‘`input/regression/completion-heads.ly`’.

Program reference: `Completion_heads_engraver`.

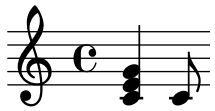
6.3 Polyphony

Polyphony in music refers to having more than one voice occurring in a piece of music. Polyphony in LilyPond refers to having more than one voice on the same staff.

6.3.1 Chords

A chord is formed by a enclosing a set of pitches between `<` and `>`. A chord may be followed by a duration, and a set of articulations, just like simple notes

```
<c e g>4 <c>8
```



For more information about chords, see Section 7.2 [Chord names], page 112.

6.3.2 Stems

Whenever a note is found, a `Stem` object is created automatically. For whole notes and rests, they are also created but made invisible.

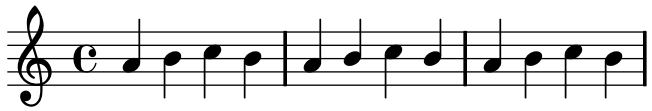
Predefined commands

`\stemUp`, `\stemDown`, `\stemNeutral`.

Commonly tweaked properties

To change the direction of stems in the middle of the staff, use

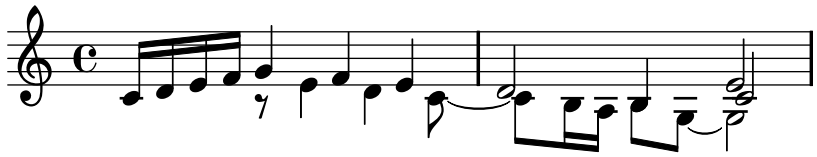
```
a4 b c b
\override Stem #'neutral-direction = #up
a4 b c b
\override Stem #'neutral-direction = #down
a4 b c b
```



6.3.3 Basic polyphony

The easiest way to enter fragments with more than one voice on a staff is to enter each voice as a sequence (with `{...}`), and combine them simultaneously, separating the voices with `\\`

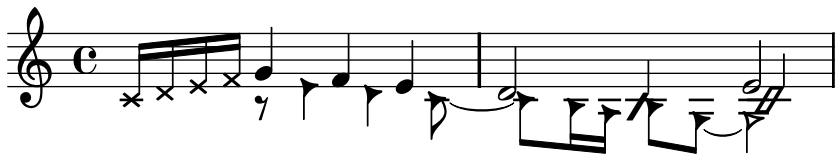
```
\new Staff \relative c' {
  c16 d e f
  <<
    { g4 f e | d2 e2 } \\
    { r8 e4 d c8 ~ | c b16 a b8 g ~ g2 } \\
    { s2. | s4 b4 c2 }
  >>
}
```



The separator causes `Voice` contexts[1] to be instantiated. They bear the names `"1"`, `"2"`, etc. In each of these contexts, vertical direction of slurs, stems, etc., is set appropriately.

These voices are all separate from the voice that contains the notes just outside the `<< \\ >>` construct. This should be noted when making changes at the voice level. This also means that slurs and ties cannot go into or out of a `<< \\ >>` construct. Conversely, parallel voices from separate `<< \\ >>` constructs on the same staff are the the same voice. Here is the same example, with different noteheads for each voice. Note that the change to the note-head style in the main voice does not affect the inside of the `<< \\ >>` constructs. Also, the change to the second voice in the first `<< \\ >>` construct is effective in the second `<< \\ >>`, and the voice is tied across the two constructs.

```
\new Staff \relative c' {
  \override NoteHead #'style = #'cross
  c16 d e f
  <<
    { g4 f e } \\
    { \override NoteHead #'style = #'triangle
    r8 e4 d c8 ~ }
  >> |
  <<
    { d2 e2 } \\
    { c8 b16 a b8 g ~ g2 } \\
    { \override NoteHead #'style = #'slash s4 b4 c2 }
  >>
}
```



Polyphony does not change the relationship of notes within a `\relative { }` block. Each note is calculated relative to the note immediately preceding it.

```
\relative { noteA << noteB \\ noteC >> noteD }
```

`noteC` is relative to `noteB`, not `noteA`; `noteD` is relative to `noteC`, not `noteB` or `noteA`.

6.3.4 Explicitly instantiating voices

`Voice` contexts can also be instantiated manually inside a `<< >>` block to create polyphonic music, using `\voiceOne`, up to `\voiceFour` to assign stem directions and a horizontal shift for each part.

Specifically,

```
<< \upper \\ \lower >>
```

is equivalent to

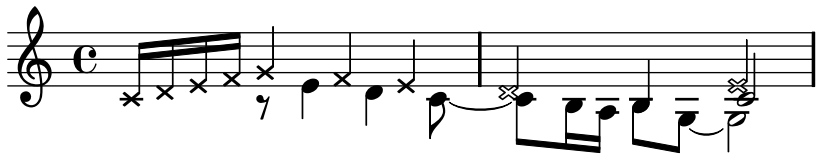
```
<<
  \new Voice = "1" { \voiceOne \upper }
  \new Voice = "2" { \voiceTwo \lower }
>>
```

[1] Polyphonic voices are sometimes called “layers” in other notation packages

The `\voiceXXX` commands set the direction of stems, slurs, ties, articulations, text annotations, augmentation dots of dotted notes, and fingerings. `\voiceOne` and `\voiceThree` make these objects point upwards, while `\voiceTwo` and `\voiceFour` make them point downwards. The command `\oneVoice` will revert back to the normal setting.

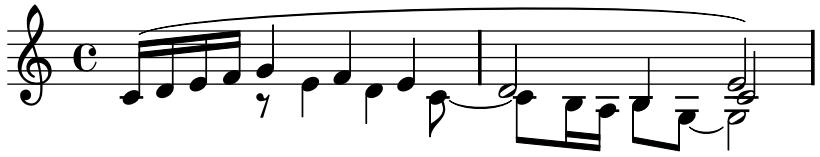
An expression that appears directly inside a `<< >>` belongs to the main voice. This is useful when extra voices appear while the main voice is playing. Here is a more correct rendition of the example from the previous section. The crossed noteheads demonstrate that the main melody is now in a single voice context.

```
\new Staff \relative c' {
  \override NoteHead #'style = #'cross
  c16 d e f
  \voiceOne
  <<
    { g4 f e | d2 e2 }
    \new Voice="1" { \voiceTwo
      r8 e4 d c8 ~ | c8 b16 a b8 g ~ g2
      \oneVoice
    }
    \new Voice { \voiceThree
      s2. | s4 b4 c2
      \oneVoice
    }
  >>
  \oneVoice
}
```



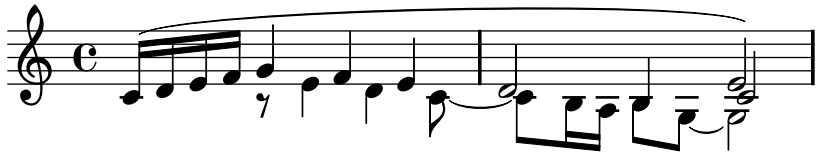
The correct definition of the voices allows the melody to be slurred.

```
\new Staff \relative c' {
  c16^( d e f
  \voiceOne
  <<
    { g4 f e | d2 e2) }
    \context Voice="1" { \voiceTwo
      r8 e4 d c8 ~ | c8 b16 a b8 g ~ g2
      \oneVoice
    }
    \new Voice { \voiceThree
      s2. s4 b4 c2
      \oneVoice
    }
  >>
  \oneVoice
}
```



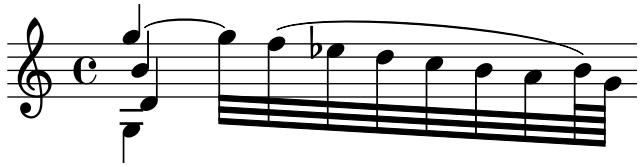
Avoiding the \\ separator also allows nesting polyphony constructs, which in some case might be a more natural way to typeset the music.

```
\new Staff \relative c' {
  c16^( d e f
  \voiceOne
  <<
    { g4 f e | d2 e2) }
    \context Voice="1" { \voiceTwo
      r8 e4 d c8 ~ |
      <<
        {c8 b16 a b8 g ~ g2}
        \new Voice { \voiceThree
          s4 b4 c2
          \oneVoice
        }
      >>
    \oneVoice
    }
  >>
  \oneVoice
}
```



In some instances of complex polyphonic music, you may need additional voices to avoid collisions between notes. Additional voices are added by defining an identifier, as shown below:

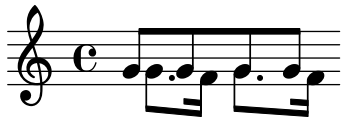
```
voiceFive = #(context-spec-music (make-voice-props-set 4) 'Voice)

\relative c''' <<
  { \voiceOne g4 ~  \stemDown g32[ f( es d c b a b64 )g] } \\
  { \voiceThree  b4} \\
  { \voiceFive d,} \\
  { \voiceTwo g,}
>>
```



6.3.5 Collision Resolution

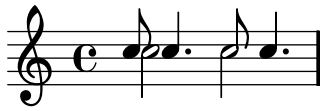
Normally, note heads with a different number of dots are not merged, but when the object property `merge-differently-dotted` is set in the `NoteCollision` object, they are merged:

```
\new Voice << {
  g8 g8
  \override Staff.NoteCollision
    #'merge-differently-dotted = ##t
  g8 g8
} \\ { g8.[ f16] g8.[ f16] } >>
```



Similarly, you can merge half note heads with eighth notes, by setting `merge-differently-headed`:

```
\new Voice << {
  c8 c4.
  \override Staff.NoteCollision
    #'merge-differently-headed = ##t
c8 c4. } \\ { c2 c2 } >>
```



`merge-differently-headed` and `merge-differently-dotted` only apply to opposing stem directions (ie. Voice 1 & 2).

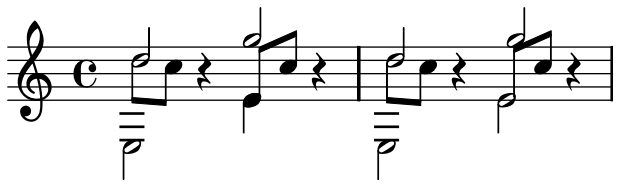
LilyPond also vertically shifts rests that are opposite of a stem, for example

```
\new Voice << c''4 \\ r4 >>
```

If three or more notes line up in the same column, `merge-differently-headed` cannot successfully complete the merge of the two notes that should be merged. To allow the merge to work properly, apply a `\shift` to the note that should not be merged. In the first measure of following example, `merge-differently-headed` does not work (the half-note head is solid). In the second measure, `\shiftOn` is applied to move the top `g` out of the column, and `merge-differently-headed` works properly.

```
\override Staff.NoteCollision #'merge-differently-headed = ##t
<<
  { d=''2 g2 } \\
  { \oneVoice d=''8 c8 r4 e,8 c'8 r4 } \\
  { \voiceFour e,,2 e'2}
>>
<<
  { d'=''2 \shiftOn g2 } \\
  { \oneVoice d=''8 c8 r4 e,8 c'8 r4 } \\
  { \voiceFour e,,2 e'2}
```

```
>>
```



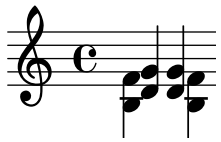
Predefined commands

`\oneVoice`, `\voiceOne`, `\voiceTwo`, `\voiceThree`, `\voiceFour`.

`\shiftOn`, `\shiftOnn`, `\shiftOnnn`, `\shiftOff`: these commands specify the degree to which chords of the current voice should be shifted. The outer voices (normally: voice one and two) have `\shiftOff`, while the inner voices (three and four) have `\shiftOn`. `\shiftOnn` and `\shiftOnnn` define further shift levels.

When LilyPond cannot cope, the `force-hshift` property of the `NoteColumn` object and pitched rests can be used to override typesetting decisions.

```
\relative <<
{
  <d g>
  <d g>
} \\ {
  <b f'>
  \once \override NoteColumn #'force-hshift = #1.7
  <b f'>
} >>
```



See also

Program reference: the objects responsible for resolving collisions are `NoteCollision` and `RestCollision`.

Examples: ‘`input/regression/collision-dots.ly`’, ‘`input/regression/collision-head-chords.ly`’, ‘`input/regression/collision-heads.ly`’, ‘`input/regression/collision-mesh.ly`’, and ‘`input/regression/collisions.ly`’.

Bugs

When using `merge-differently-headed` with an upstem eighth or a shorter note, and a downstem half note, the eighth note gets the wrong offset.

There is no support for clusters where the same note occurs with different accidentals in the same chord. In this case, it is recommended to use enharmonic transcription, or to use special cluster notation (see Section 8.4.4 [Clusters], page 205).

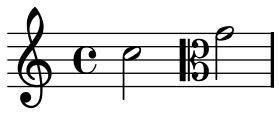
6.4 Staff notation

This section describes music notation that occurs on staff level, such as key signatures, clefs and time signatures.

6.4.1 Clef

The clef indicates which lines of the staff correspond to which pitches. The clef is set with the `\clef` command

```
{ c''2 \clef alto g'2 }
```

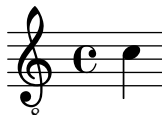


Supported clefs include

Clef	**Position**
`treble`, violin, G, G2	G clef on 2nd line
`alto`, `C`	C clef on 3rd line
`tenor`	C clef on 4th line.
`bass`, `F`	F clef on 4th line
`french`	G clef on 1st line, so-called French violin clef
`soprano`	C clef on 1st line
`mezzosoprano`	C clef on 2nd line
`baritone`	C clef on 5th line
`varbaritone`	F clef on 3rd line
`subbass`	F clef on 5th line
`percussion`	percussion clef
`tab`	tablature clef

By adding `_8` or `^8` to the clef name, the clef is transposed one octave down or up, respectively, and `_15` and `^15` transposes by two octaves. The argument *clefname* must be enclosed in quotes when it contains underscores or digits. For example,

```
\clef "G_8" c4
```

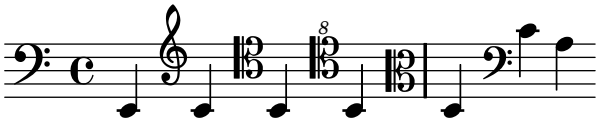


Commonly tweaked properties

The command `\clef "treble_8"` is equivalent to setting `clefGlyph`, `clefPosition` (which controls the Y position of the clef), `middleCPosition` and `clefOctavation`. A clef is printed when any of these properties are changed. The following example shows possibilities when setting properties manually.

```
{
  \set Staff.clefGlyph = #"clefs.F"
  \set Staff.clefPosition = #2
  c'4
  \set Staff.clefGlyph = #"clefs.G"
  c'4
```

```
  \set Staff.clefGlyph = #"clefs.C"
  c'4
  \set Staff.clefOctavation = #7
  c'4
  \set Staff.clefOctavation = #0
  \set Staff.clefPosition = #0
  c'4
  \clef "bass"
  c'4
  \set Staff.middleCPosition = #4
  c'4
}
```



See also

Manual: Section 6.5.7 [Grace notes], page 91.

Program reference: `Clef`.

6.4.2 Key signature

The key signature indicates the tonality in which a piece is played. It is denoted by a set of alterations (flats or sharps) at the start of the staff.

Setting or changing the key signature is done with the `\key` command

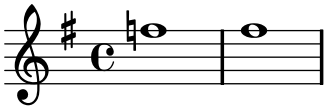
`\key` *pitch type*

Here, *type* should be `\major` or `\minor` to get *pitch*-major or *pitch*-minor, respectively. You may also use the standard mode names (also called "church modes"): `\ionian`, `\locrian`, `\aeolian`, `\mixolydian`, `\lydian`, `\phrygian`, and `\dorian`.

This command sets the context property `Staff.keySignature`. Non-standard key signatures can be specified by setting this property directly.

Accidentals and key signatures often confuse new users, because unaltered notes get natural signs depending on the key signature. For more information, see Section 6.1.2 [Accidentals], page 60 or Section 2.2.2 [Accidentals and key signatures], page 17.

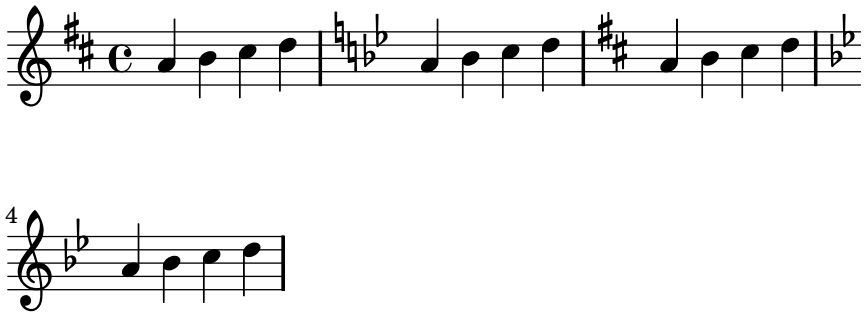
```
\key g \major
f1
fis
```



Commonly tweaked properties

A natural sign is printed to cancel any previous accidentals. This can be suppressed by setting the `Staff.printKeyCancellation` property.

```
\key d \major
a b cis d
\key g \minor
a bes c d
\set Staff.printKeyCancellation = ##f
\key d \major
a b cis d
\key g \minor
a bes c d
```



See also

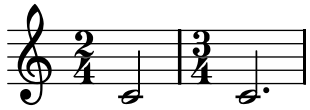
Program reference: `KeyCancellation`, `KeySignature`.

6.4.3 Time signature

Time signature indicates the metrum of a piece: a regular pattern of strong and weak beats. It is denoted by a fraction at the start of the staff.

The time signature is set with the `\time` command

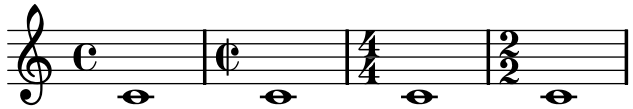
```
\time 2/4 c'2 \time 3/4 c'2.
```



Commonly tweaked properties

The symbol that is printed can be customized with the `style` property. Setting it to `#'()` uses fraction style for 4/4 and 2/2 time,

```
\time 4/4 c'1
\time 2/2 c'1
\override Staff.TimeSignature #'style = #'()
\time 4/4 c'1
\time 2/2 c'1
```

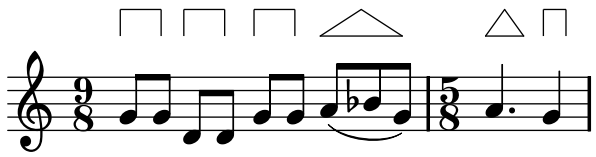


There are many more options for its layout. See Section 7.7.6 [Ancient time signatures], page 152 for more examples.

`\time` sets the property `timeSignatureFraction`, `beatLength` and `measureLength` in the `Timing` context, which is normally aliased to `Score`. The property `measureLength` determines where bar lines should be inserted, and how automatic beams should be generated. Changing the value of `timeSignatureFraction` also causes the symbol to be printed.

More options are available through the Scheme function `set-time-signature`. In combination with the `Measure_grouping_engraver`, it will create `MeasureGrouping` signs. Such signs ease reading rhythmically complex modern music. In the following example, the 9/8 measure is subdivided in 2, 2, 2 and 3. This is passed to `set-time-signature` as the third argument `(2 2 2 3)`

```
\score {
  \relative c'' {
    #(set-time-signature 9 8 '(2 2 2 3))
    g8[ g] d[ d] g[ g] a8[( bes g]) |
    #(set-time-signature 5 8 '(3 2))
    a4. g4
  }
  \layout {
    \context {
      \Staff
      \consists "Measure_grouping_engraver"
    }
  }
}
```



See also

Program reference: `TimeSignature`, and `Timing_translator`.

Examples: ‘`input/test/compound-time.ly`’.

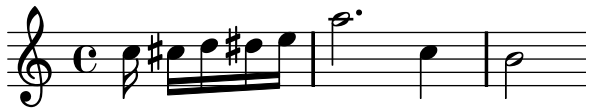
Bugs

Automatic beaming does not use the measure grouping specified with `set-time-signature`.

6.4.4 Partial measures

Partial measures, such as an anacrusis or upbeat, are entered using the

```
\partial 16*5 c16 cis d dis e | a2. c,4 | b2
```



The syntax for this command is

```
\partial duration
```

where `duration` is the rhythmic length to be added before the next bar.

This is internally translated into

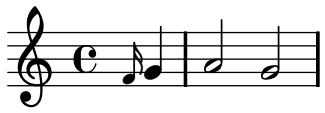
```
\set Timing.measurePosition = -length of duration
```

The property `measurePosition` contains a rational number indicating how much of the measure has passed at this point. Note that this is a negative number; `\partial 4` is internally translated to mean "there is a quarter note left in the bar".

Bugs

This command does not take into account grace notes at the start of the music. When a piece starts with graces notes in the pickup, then the `\partial` should follow the grace notes

```
\grace f16
\partial 4
g4
a2 g2
```



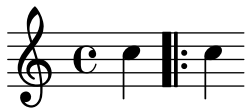
`\partial` is only intended to be used at the beginning of a piece. If you use it after the beginning, some odd warnings may occur.

6.4.5 Bar lines

Bar lines delimit measures, but are also used to indicate repeats. Normally they are inserted automatically. Line breaks may only happen on bar lines.

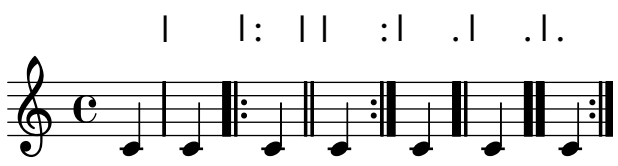
Special types of bar lines can be forced with the `\bar` command

```
c4 \bar "|:" c4
```

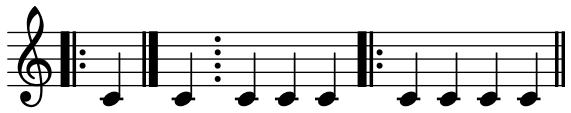


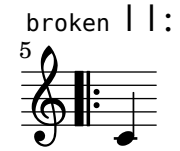
The following bar types are available

```
|   |:  ||   :|   .|   .|.
```



```
:|:   |.   :    unbroken ||:
```





In addition, you can specify `"||:"`, which is equivalent to `"|:"` except at line breaks, where it gives a double bar line at the end of the line and a start repeat at the beginning of the next line.

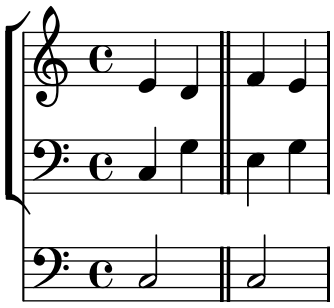
To allow a line break where there is no visible bar line, use

```
\bar ""
```

This will insert an invisible bar line and allow line breaks at this point (without increasing the bar number counter).

In scores with many staves, a `\bar` command in one staff is automatically applied to all staves. The resulting bar lines are connected between different staves of a `StaffGroup`, `PianoStaff`, or `ChoirStaff`.

```
<<
  \new StaffGroup <<
    \new Staff {
      e'4 d'
      \bar "||"
      f' e'
    }
    \new Staff { \clef bass c4 g e g }
  >>
  \new Staff { \clef bass c2 c2 }
>>
```



Commonly tweaked properties

The command `\bar` *bartype* is a short cut for doing `\set Timing.whichBar =` *bartype*. Whenever `whichBar` is set to a string, a bar line of that type is created.

A bar line is created whenever the `whichBar` property is set. At the start of a measure it is set to the contents of `Timing.defaultBarType`. The contents of `repeatCommands` are used to override default measure bars.

You are encouraged to use `\repeat` for repetitions. See Section 6.7 [Repeats], page 103.

See also

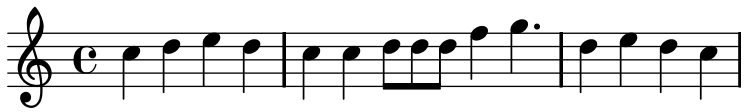
In this manual: Section 6.7 [Repeats], page 103, Section 6.4.7 [System start delimiters], page 82.

Program reference: `BarLine` (created at `Staff` level), `SpanBar` (across staves).

6.4.6 Unmetered music

Bar lines and bar numbers are calculated automatically. For unmetered music (cadenzas, for example), this is not desirable. To turn off automatic bar lines and bar numbers, use the commands `\cadenzaOn` and `\cadenzaOff`.

```
c4 d e d
\cadenzaOn
c4 c d8 d d f4 g4.
\cadenzaOff
\bar "|"
d4 e d c
```



Bugs

LilyPond will only insert line breaks and page breaks at a barline. Unless the unmetered music ends before the end of the staff line, you will need to insert invisible bar lines

```
\bar ""
```

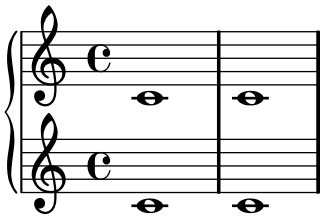
to indicate where breaks can occur.

6.4.7 System start delimiters

Many scores consist of more than one staff. These staves can be joined in four different ways

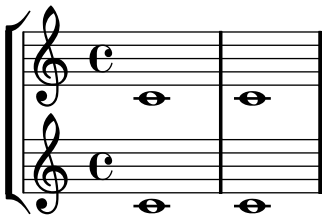
- The group is started with a brace at the left, and bar lines are connected. This is done with the `GrandStaff` context.

```
\new GrandStaff
\relative <<
  \new Staff { c1 c }
  \new Staff { c c }
>>
```



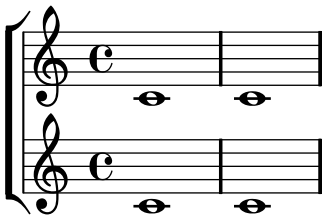
- The group is started with a bracket, and bar lines are connected. This is done with the `StaffGroup` context

```
\new StaffGroup
\relative <<
  \new Staff { c1 c }
  \new Staff { c c }
>>
```



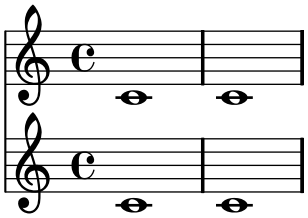
- The group is started with a bracket, but bar lines are not connected. This is done with the `ChoirStaff` context.

```
\new ChoirStaff
\relative <<
  \new Staff { c1 c }
  \new Staff { c c }
>>
```



- The group is started with a vertical line. Bar lines are not connected. This is the default for the score.

```
\relative <<
  \new Staff { c1 c }
  \new Staff { c c }
>>
```



See also

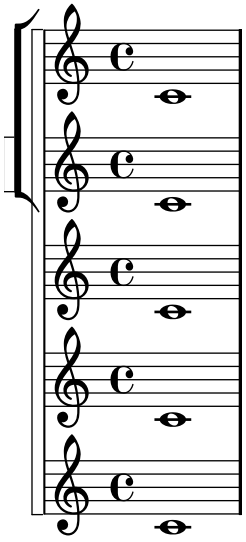
The bar lines at the start of each system are `SystemStartBar`, `SystemStartBrace`, and `SystemStartBracket`. Only one of these types is created in every context, and that type is determined by the property `systemStartDelimiter`.

Commonly tweaked properties

System start delimiters may be deeply nested,

```
\new StaffGroup
\relative <<
  \set StaffGroup.systemStartDelimiterHierarchy
    = #'(SystemStartSquare (SystemStartBracket a (SystemStartSquare b)) d)
  \new Staff { c1 }
  \new Staff { c1 }
  \new Staff { c1 }
  \new Staff { c1 }
```

```
    \new Staff { c1 }
  >>
```



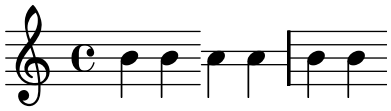
6.4.8 Staff symbol

Notes, dynamic signs, etc., are grouped with a set of horizontal lines, called a staff (plural 'staves'). In LilyPond, these lines are drawn using a separate layout object called `staff symbol`.

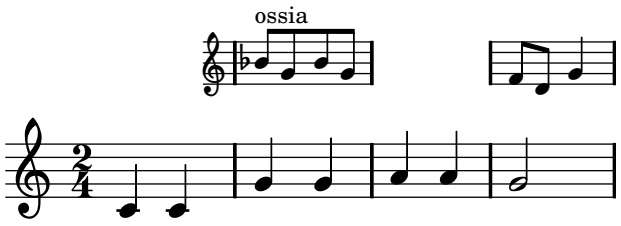
The staff symbol may be tuned in the number, thickness and distance of lines, using properties. This is demonstrated in the example files '`input/test/staff-lines.ly`', '`input/test/staff-size.ly`'.

In addition, staves may be started and stopped at will. This is done with `\startStaff` and `\stopStaff`.

```
b4 b
\override Staff.StaffSymbol #'line-count = 2
\stopStaff \startStaff
b b
\revert Staff.StaffSymbol #'line-count
\stopStaff \startStaff
b b
```



In combination with Frenched staves, this may be used to typeset ossia sections. An example is shown here



See also

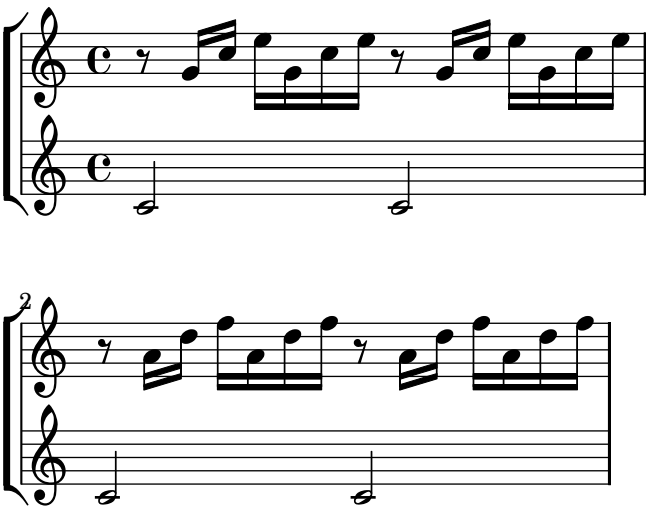
Program reference: `StaffSymbol`.

Examples: ‘`input/test/staff-lines.ly`’, ‘`input/test/ossia.ly`’, ‘`input/test/staff-size.ly`’, ‘`input/regression/staff-line-positions.ly`’.

6.4.9 Writing music in parallel

Music for multiple parts can be interleaved

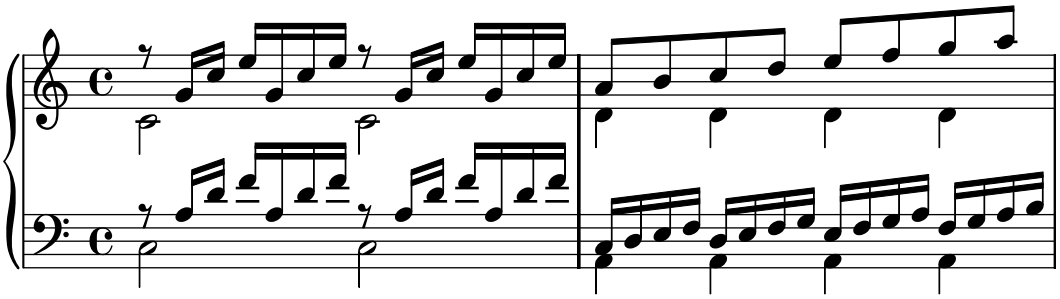
```
\parallelMusic #'(voiceA voiceB) {
  r8     g'16[ c''] e''[ g' c'' e''] r8     g'16[ c''] e''[ g' c'' e''] |
  c'2                                c'2                                |
  r8     a'16[ d''] f''[ a' d'' f''] r8     a'16[ d''] f''[ a' d'' f''] |
  c'2                                c'2                                |
}
\new StaffGroup <<
  \new Staff \new Voice \voiceA
  \new Staff \new Voice \voiceB
>>
```



This works quite well for piano music

```
music = {
  \key c \major
  \time 4/4
  \parallelMusic #'(voiceA voiceB voiceC voiceD) {
    % Bar 1
    r8  g'16[ c''] e''[ g' c'' e''] r8  g'16[ c''] e''[ g' c''
e''] |
    c'2                                c'2 |
    r8  a16[ d'] f'[ a d' f']       r8  a16[ d'] f'[ a d' f'] |
    c2                                 c2 |

    % Bar 2
    a'8 b'      c'' d''    e'' f''    g'' a'' |
    d'4         d'         d'         d' |
    c16 d e f   d e f g    e f g a    f g a b |
    a,4         a,4        a,4        a,4 |
```

```
    % Bar 3 ...
  }
}

\score {
  \new PianoStaff <<
    \music
    \new Staff <<
      \voiceA \\
      \voiceB
    >>
    \new Staff {
      \clef bass
      <<
        \voiceC \\
        \voiceD
      >>
    }
  >>
}
```



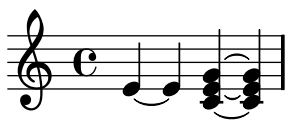
6.5 Connecting notes

This section deals with notation that affects groups of notes.

6.5.1 Ties

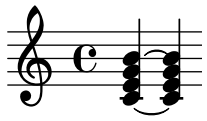
A tie connects two adjacent note heads of the same pitch. The tie in effect extends the length of a note. Ties should not be confused with slurs, which indicate articulation, or phrasing slurs, which indicate musical phrasing. A tie is entered using the tilde symbol '~'

```
e' ~ e' <c' e' g'> ~ <c' e' g'>
```

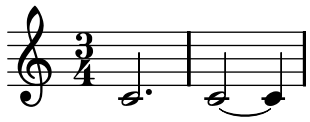


When a tie is applied to a chord, all note heads whose pitches match are connected. When no note heads match, no ties will be created. Chords may be partially tied by placing the tie inside the chord,

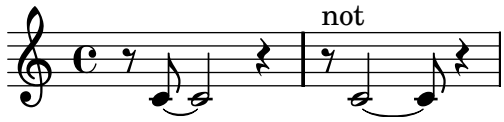
```
<c~ e g~ b> <c e g b>
```



A tie is just a way of extending a note duration, similar to the augmentation dot. The following example shows two ways of notating exactly the same concept

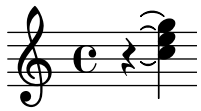


Ties are used either when the note crosses a bar line, or when dots cannot be used to denote the rhythm. When using ties, larger note values should be aligned to subdivisions of the measure, such as



If you need to tie a lot of notes over bars, it may be easier to use automatic note splitting (see Section 6.2.7 [Automatic note splitting], page 69). This mechanism automatically splits long notes, and ties them across bar lines.

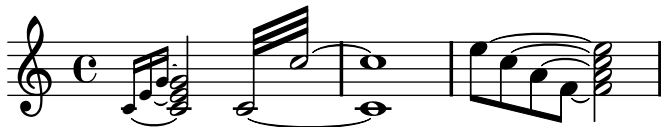
When a second alternative of a repeat starts with a tied note, you have to repeat the tie. This can be achieved with `\repeatTie`,



Commonly tweaked properties

Ties are sometimes used to write out arpeggios. In this case, two tied notes need not be consecutive. This can be achieved by setting the `tieWaitForNote` property to true. The same feature is also useful, for example, to tie a tremolo to a chord. For example,

```
\set tieWaitForNote = ##t
\grace { c16[~ e~ g]~ } <c, e g>2
\repeat "tremolo" 8 { c32~ c'~ } <c c,>1
e8~ c~ a~ f~ <e' c a f>2
```



Ties may be engraved manually by changing the `tie-configuration` property. The first number indicates the distance from the center of the staff in staff-spaces, and the second number indicates the direction (1=up, -1=down).

```
<c e g>2~ <c e g> |
\override TieColumn #'tie-configuration =
  #'((0.0 . 1) (-2.0 . 1) (-4.0 . 1))
<c e g>~ <c e g> |
```

Predefined commands

`\tieUp`, `\tieDown`, `\tieNeutral`, `\tieDotted`, `\tieDashed`, `\tieSolid`.

See also

In this manual: Section 6.2.7 [Automatic note splitting], page 69.

Program reference: `Tie`.

Examples: ‘`input/regression/tie-arpeggio.ly`’ ‘`input/regression/tie-manual.ly`’

Bugs

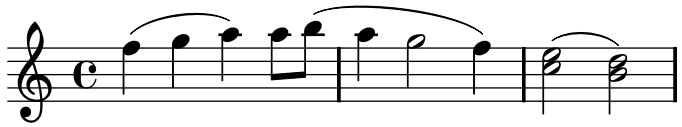
Switching staves when a tie is active will not produce a slanted tie.

Changing clefs or octavations during a tie is not really well-defined. In these cases, a slur may be preferable.

6.5.2 Slurs

A slur indicates that notes are to be played bound or *legato*. They are entered using parentheses

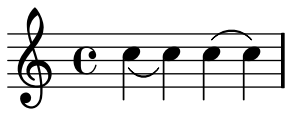
```
f( g a) a8 b( a4 g2 f4)
<c e>2( <b d>2)
```



The direction of a slur can be specified with `\slur`*`DIR`*, where *`DIR`* is either `Up`, `Down`, or `Neutral` (automatically selected).

However, there is a convenient shorthand for forcing slur directions. By adding `_` or `^` before the opening parentheses, the direction is also set. For example,

```
c4_( c) c^( c)
```

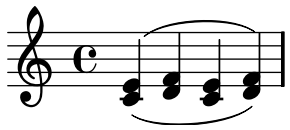


Only one slur can be printed at once. If you need to print a long slur over a few small slurs, please see Section 6.5.3 [Phrasing slurs], page 89.

Commonly tweaked properties

Some composers write two slurs when they want legato chords. This can be achieved in LilyPond by setting `doubleSlurs`,

```
\set doubleSlurs = ##t
<c e>4 ( <d f> <c e> <d f> )
```



Predefined commands

`\slurUp`, `\slurDown`, `\slurNeutral`, `\slurDashed`, `\slurDotted`, `\slurSolid`.

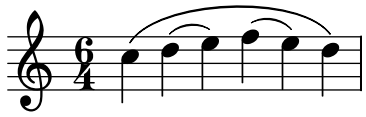
See also

Program reference: internals document, `Slur`.

6.5.3 Phrasing slurs

A phrasing slur (or phrasing mark) connects notes and is used to indicate a musical sentence. It is written using `\(` and `\)` respectively

```
\time 6/4 c'\( d( e) f( e) d\)
```



Typographically, the phrasing slur behaves almost exactly like a normal slur. However, they are treated as different objects. A `\slurUp` will have no effect on a phrasing slur; instead, use `\phrasingSlurUp`, `\phrasingSlurDown`, and `\phrasingSlurNeutral`.

You cannot have simultaneous phrasing slurs.

Predefined commands

`\phrasingSlurUp`, `\phrasingSlurDown`, `\phrasingSlurNeutral`.

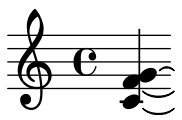
See also

Program reference: `PhrasingSlur`.

6.5.4 Laissez vibrer ties

L.v. ties (laissez vibrer) indicate that notes must not be damped at the end. It is used in notation for piano, harp and other string and percussion instruments. They can be entered using `\laissezVibrer`,

```
<c f g>\laissezVibrer
```



See also

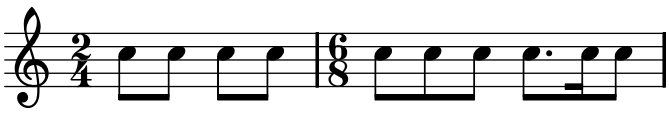
Program reference: `LaissezVibrerTie` `LaissezVibrerTieColumn`

Example files: '`input/regression/laissez-vibrer-tie.ly`'

6.5.5 Automatic beams

LilyPond inserts beams automatically

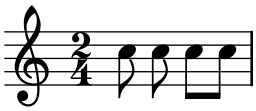
```
\time 2/4 c8 c c c \time 6/8 c c c c8. c16 c8
```



When these automatic decisions are not good enough, beaming can be entered explicitly. It is also possible to define beaming patterns that differ from the defaults. See Section 9.1.2 [Setting automatic beam behavior], page 215 for details.

Individual notes may be marked with `\noBeam` to prevent them from being beamed

```
\time 2/4 c8 c\noBeam c c
```



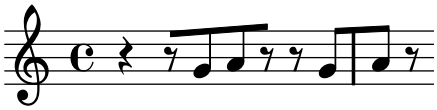
See also

Program reference: `Beam`.

6.5.6 Manual beams

In some cases it may be necessary to override the automatic beaming algorithm. For example, the autobeamer will not put beams over rests or bar lines. Such beams are specified manually by marking the begin and end point with `[` and `]`

```
{
  r4 r8[ g' a r8] r8 g[ | a] r8
}
```

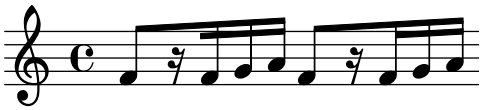


Commonly tweaked properties

Normally, beaming patterns within a beam are determined automatically. If necessary, the properties `stemLeftBeamCount` and `stemRightBeamCount` can be used to override the defaults. If either property is set, its value will be used only once, and then it is erased

```
{
   f8[ r16
      f g a]
   f8[ r16
   \set stemLeftBeamCount = #1
```

```
        f g a]
}
```



The property `subdivideBeams` can be set in order to subdivide all 16th or shorter beams at beat positions, as defined by the `beatLength` property.

```
c16[ c c c c c c c]
\set subdivideBeams = ##t
c16[ c c c c c c c]
\set Score.beatLength = #(ly:make-moment 1 8)
c16[ c c c c c c c]
```

For more information about `make-moment`, see Section 8.4.2 [Time administration], page 203.

Line breaks are normally forbidden when beams cross bar lines. This behavior can be changed by setting `allowBeamBreak`.

Bugs

Kneed beams are inserted automatically when a large gap is detected between the note heads. This behavior can be tuned through the object.

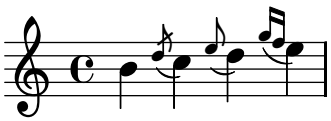
Automatically kneed cross-staff beams cannot be used together with hidden staves. See Section 8.3.2 [Hiding staves], page 197.

Beams do not avoid collisions with symbols around the notes, such as texts and accidentals.

6.5.7 Grace notes

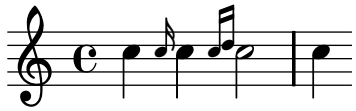
Grace notes are ornaments that are written out. The most common ones are acciaccatura, which should be played as very short. It is denoted by a slurred small note with a slashed stem. The appoggiatura is a grace note that takes a fixed fraction of the main note, and is denoted as a slurred note in small print without a slash. They are entered with the commands `\acciaccatura` and `\appoggiatura`, as demonstrated in the following example

```
b4 \acciaccatura d8 c4 \appoggiatura e8 d4
\acciaccatura { g16[ f] } e4
```



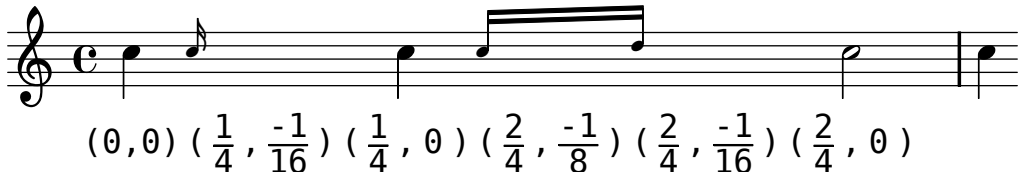
Both are special forms of the `\grace` command. By prefixing this keyword to a music expression, a new one is formed, which will be printed in a smaller font and takes up no logical time in a measure.

```
c4 \grace c16 c4
\grace { c16[ d16] } c2 c4
```



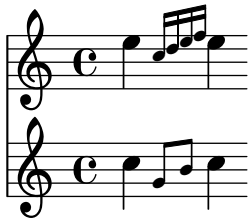
Unlike `\acciaccatura` and `\appoggiatura`, the `\grace` command does not start a slur.

Internally, timing for grace notes is done using a second, 'grace' timing. Every point in time consists of two rational numbers: one denotes the logical time, one denotes the grace timing. The above example is shown here with timing tuples



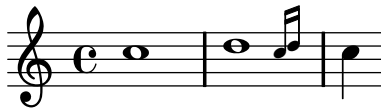
The placement of grace notes is synchronized between different staves. In the following example, there are two sixteenth grace notes for every eighth grace note

```
<< \new Staff { e4 \grace { c16[ d e f] } e4 }
   \new Staff { c4 \grace { g8[ b] } c4 } >>
```



If you want to end a note with a grace, use the `\afterGrace` command. It takes two arguments: the main note, and the grace notes following the main note.

```
c1 \afterGrace d1 { c16[ d] } c4
```



This will put the grace notes after a "space" lasting 3/4 of the length of the main note. The fraction 3/4 can be changed by setting `afterGraceFraction`, ie.

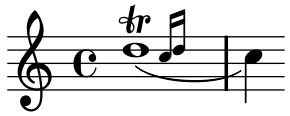
```
afterGraceFraction = #(cons 7 8)
```

will put the grace note at 7/8 of the main note.

The same effect can be achieved manually by doing

```
\new Voice {
  << { d1^\trill_( }
     { s2 \grace { c16[ d] } } >>
  c4)
```

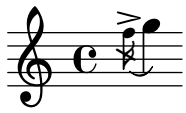
```
}
```



By adjusting the duration of the skip note (here it is a half-note), the space between the main-note and the grace is adjusted.

A `\grace` section will introduce special typesetting settings, for example, to produce smaller type, and set directions. Hence, when introducing layout tweaks, they should be inside the grace section, for example,

```
\new Voice {
  \acciaccatura {
    \stemDown
    f16->
    \stemNeutral
  }
  g4
}
```



The overrides should also be reverted inside the grace section.

The layout of grace sections can be changed throughout the music using the function `add-grace-property`. The following example undefines the `Stem` direction for this grace, so that stems do not always point up.

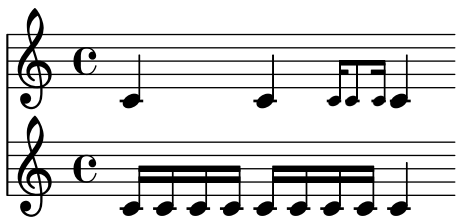
```
\new Staff {
  #(add-grace-property 'Voice 'Stem 'direction '())
  ...
}
```

Another option is to change the variables `startGraceMusic`, `stopGraceMusic`, `startAcciaccaturaMusic`, `stopAcciaccaturaMusic`, `startAppoggiaturaMusic`, `stopAppoggiaturaMusic`. More information is in the file '`ly/grace-init.ly`'.

The slash through the stem in acciaccaturas can be obtained in other situations by `\override Stem #'stroke-style = #"grace"`.

Commonly tweaked properties

Grace notes may be forced to use floating spacing,



See also

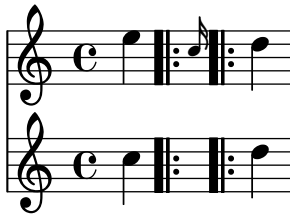
Program reference: `GraceMusic`.

Bugs

A score that starts with a `\grace` section needs an explicit `\new Voice` declaration, otherwise the main note and the grace note end up on different staves.

Grace note synchronization can also lead to surprises. Staff notation, such as key signatures, bar lines, etc., are also synchronized. Take care when you mix staves with grace notes and staves without, for example,

```
<< \new Staff { e4 \bar "|:" \grace c16 d4 }
   \new Staff { c4 \bar "|:" d4 } >>
```



This can be remedied by inserting grace skips of the corresponding durations in the other staves. For the above example

```
\new Staff { c4 \bar "|:" \grace s16 d4 }
```

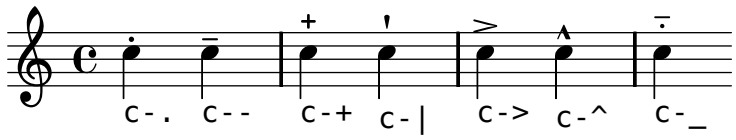
Grace sections should only be used within sequential music expressions. Nesting or juxtaposing grace sections is not supported, and might produce crashes or other errors.

6.6 Expressive marks

Expressive marks help musicians to bring more to the music than simple notes and rhythms.

6.6.1 Articulations

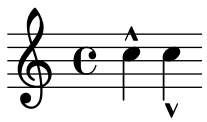
A variety of symbols can appear above and below notes to indicate different characteristics of the performance. They are added to a note by adding a dash and the character signifying the articulation. They are demonstrated here



The meanings of these shorthands can be changed. See '`ly/script-init.ly`' for examples.

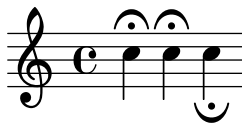
The script is automatically placed, but the direction can be forced as well. Like other pieces of LilyPond code, `_` will place them below the staff, and `^` will place them above.

```
c''4^^ c''4_^
```

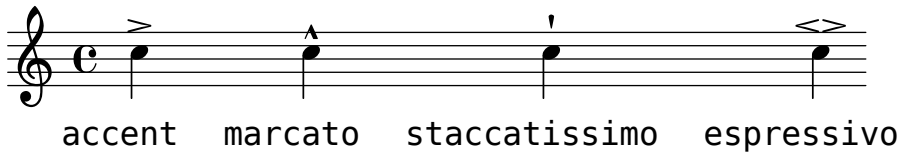


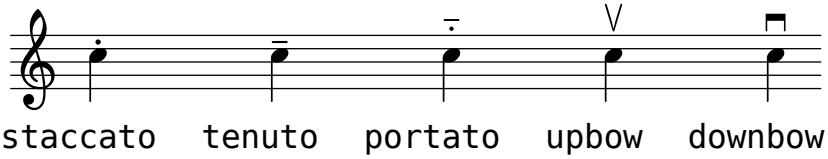
Other symbols can be added using the syntax *note**name*. Again, they can be forced up or down using ^ and _, e.g.,

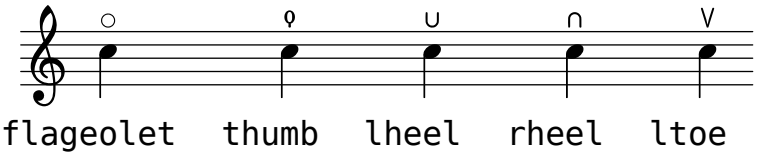
```
c\fermata c^\fermata c_\fermata
```

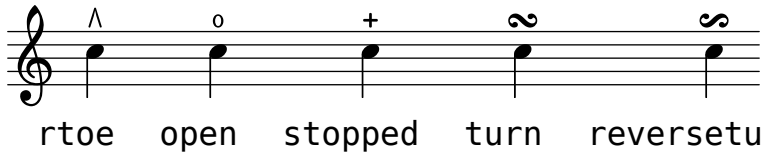


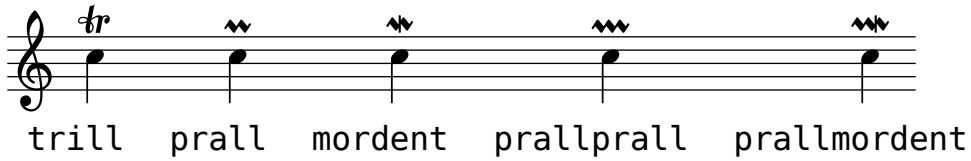
Here is a chart showing all scripts available,

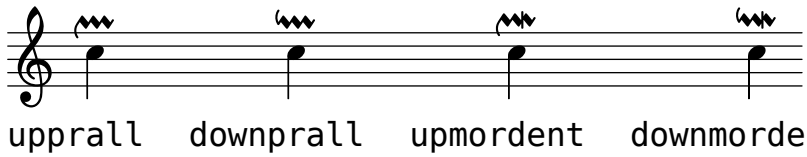


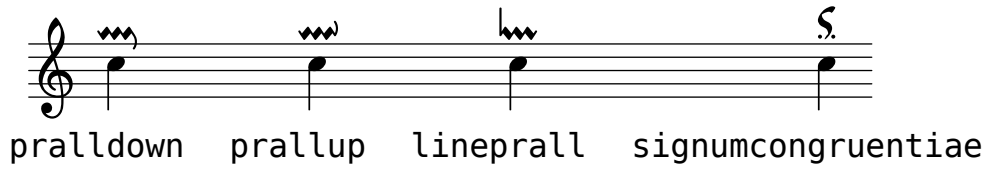


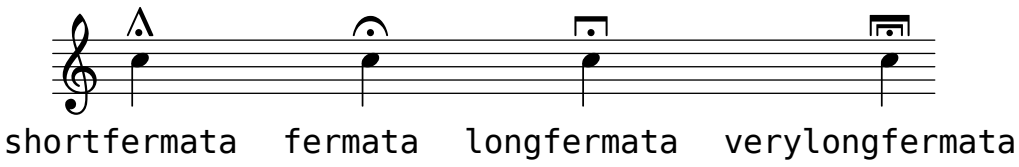


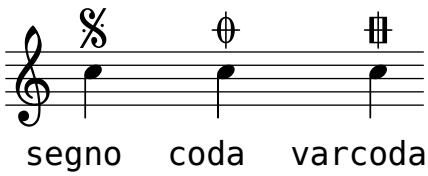








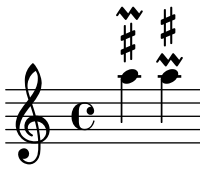




Commonly tweaked properties

The vertical ordering of scripts is controlled with the `script-priority` property. The lower this number, the closer it will be put to the note. In this example, the `TextScript` (the sharp symbol) first has the lowest priority, so it is put lowest in the first example. In the second, the prall trill (the `Script`) has the lowest, so it is on the inside. When two objects have the same priority, the order in which they are entered decides which one comes first.

```
\once \override TextScript #'script-priority = #-100
a4^\prall^\markup { \sharp }

\once \override Script #'script-priority = #-100
a4^\prall^\markup { \sharp }
```



See also

Program reference: `Script`.

Bugs

These signs appear in the printed output but have no effect on the MIDI rendering of the music.

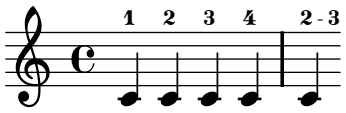
6.6.2 Fingering instructions

Fingering instructions can be entered using

note-digit

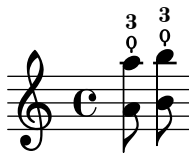
For finger changes, use markup texts

```
c4-1 c-2 c-3 c-4
c^\markup { \finger "2 - 3" }
```



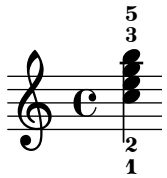
You can use the thumb-script to indicate that a note should be played with the thumb (e.g., in cello music)

```
<a_\thumb a'-3>8 <b_\thumb b'-3>
```



Fingerings for chords can also be added to individual notes of the chord by adding them after the pitches

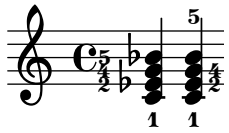
```
< c-1 e-2 g-3 b-5 >4
```



Commonly tweaked properties

You may exercise greater control over fingering chords by setting `fingeringOrientations`

```
\set fingeringOrientations = #'(left down)
<c-1 es-2 g-4 bes-5 > 4
\set fingeringOrientations = #'(up right down)
<c-1 es-2 g-4 bes-5 > 4
```



Using this feature, it is also possible to put fingering instructions very close to note heads in monophonic music,

```
\set fingeringOrientations = #'(right)
<es'-2>4
```

See also

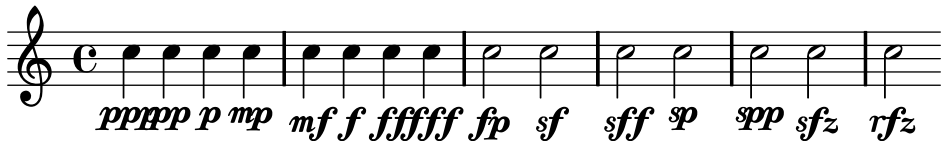
Program reference: `Fingering`.

Examples: ‘`input/regression/finger-chords.ly`’.

6.6.3 Dynamics

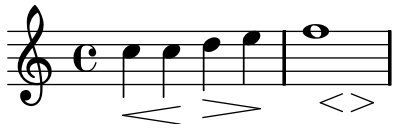
Absolute dynamic marks are specified using a command after a note `c4\ff`. The available dynamic marks are `\ppppp`, `\pppp`, `\ppp`, `\pp`, `\p`, `\mp`, `\mf`, `\f`, `\ff`, `\fff`, `\ffff`, `\fp`, `\sf`, `\sff`, `\sp`, `\spp`, `\sfz`, and `\rfz`.

```
c\ppp c\pp c \p c\mp c\mf c\f c\ff c\fff
c2\fp c\sf c\sff c\sp c\spp c\sfz c\rfz
```



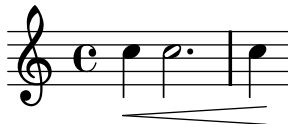
A crescendo mark is started with `\<` and terminated with `\!` or an absolute dynamic. A decrescendo is started with `\>` and is also terminated with `\!` or an absolute dynamic. `\cr` and `\decr` may be used instead of `\<` and `\>`. Because these marks are bound to notes, you must use spacer notes if multiple marks are needed during one note

```
c\< c\! d\> e\!
<< f1 { s4 s4\< s4\! \> s4\! } >>
```



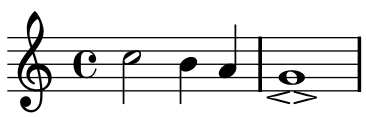
A hairpin normally starts at the left edge of the beginning note and ends on the right edge of the ending note. If the ending note falls on the downbeat, the hairpin ends on the immediately preceding barline. This may be modified by setting the `hairpinToBarline` property,

```
\set hairpinToBarline = ##f
c4\< c2. c4\!
```



In some situations the `\espressivo` articulation mark may be suitable to indicate a crescendo and decrescendo on the one note,

```
c2 b4 a g1\espressivo
```

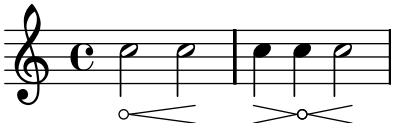


This may give rise to very short hairpins. Use `minimum-length` in `Voice.Hairpin` to lengthen them, for example

```
\override Voice.Hairpin #'minimum-length = #5
```

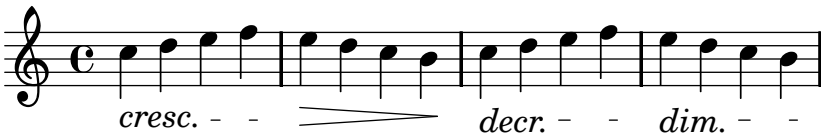
Hairpins may be printed with a circled tip (al niente notation) by setting the `circled-tip` property,

```
\override Hairpin #'circled-tip = ##t
c2\< c\!
c4\> c\< c2\!
```



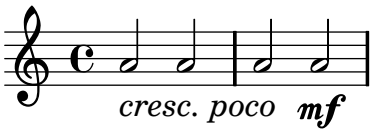
You can also use text saying *cresc.* instead of hairpins

```
\setTextCresc
c\< d e f\!
\setHairpinCresc
e\> d c b\!
\setTextDecresc
c\> d e f\!
\setTextDim
e\> d c b\!
```



You can also supply your own texts

```
\set crescendoText = \markup { \italic "cresc. poco" }
\set crescendoSpanner = #'dashed-line
a'2\< a a a\!\mf
```



To create new dynamic marks or text that should be aligned with dynamics, see Section 8.1.8 [New dynamic marks], page 184.

Commonly tweaked properties

Dynamics that occur at, begin on, or end on, the same note will be vertically aligned. If you want to ensure that dynamics are aligned when they do not occur on the same note, you can increase the `staff-padding` property.

```
\override DynamicLineSpanner #'staff-padding = #4
```

You may also use this property if the dynamics are colliding with other notation.

Crescendi and decrescendi that end on the first note of a new line are not printed. To change this behavior, use

```
\override Score.Hairpin #'after-line-breaking = ##t
```

Text style dynamic changes (such as *cresc.* and *dim.*) are printed with a dashed line showing their extent. To surpress printing this line, use

```
\override DynamicTextSpanner #'dash-period = #-1.0
```

Predefined commands

`\dynamicUp`, `\dynamicDown`, `\dynamicNeutral`.

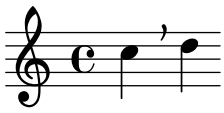
See also

Program reference: `DynamicText`, `Hairpin`. Vertical positioning of these symbols is handled by `DynamicLineSpanner`.

6.4.4 Breath marks

Breath marks are entered using `\breathe`

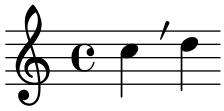
```
c'4 \breathe d4
```



Commonly tweaked properties

The glyph of the breath mark can be tuned by overriding the `text` property of the `BreathingSign` layout object with any markup text. For example,

```
c'4
\override BreathingSign #'text
  = #(make-musicglyph-markup "scripts.rvarcomma")
\breathe
d4
```



See also

Program reference: `BreathingSign`.

Examples: '`input/regression/breathing-sign.ly`'.

6.4.5 Trills

Short trills are printed like normal articulation; see Section 6.4.1 [Articulations], page 94.

Long running trills are made with `\startTrillSpan` and `\stopTrillSpan`,

```
\new Voice {
  << { c1 \startTrillSpan }
     { s2. \grace { d16[\stopTrillSpan e] } } >>
  c4 }
```

Trills that should be executed on an explicitly specified pitch can be typeset with the command `pitchedTrill`,

```
\pitchedTrill c4\startTrillSpan fis
f\stopTrillSpan
```

The first argument is the main note. The pitch of the second is printed as a stemless note head in parentheses.

Predefined commands

`\startTrillSpan`, `\stopTrillSpan`.

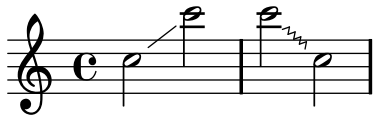
See also

Program reference: `TrillSpanner`.

6.6.6 Glissando

A glissando is a smooth change in pitch. It is denoted by a line or a wavy line between two notes. It is requested by attaching `\glissando` to a note

```
c2\glissando c'
\override Glissando #'style = #'zigzag
c2\glissando c,
```



See also

Program reference: `Glissando`.

Example files: '`input/regression/glissando.ly`'.

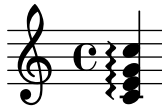
Bugs

Printing text over the line (such as *gliss.*) is not supported.

6.6.7 Arpeggio

You can specify an arpeggio sign (also known as broken chord) on a chord by attaching an `\arpeggio` to a chord

```
<c e g c>\arpeggio
```

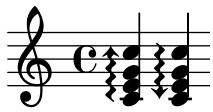


A square bracket on the left indicates that the player should not arpeggiate the chord

```
\arpeggioBracket
<c' e g c>\arpeggio
```

The direction of the arpeggio is sometimes denoted by adding an arrowhead to the wiggly line

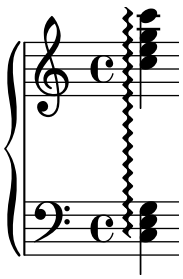
```
\new Voice {
  \arpeggioUp
  <c e g c>\arpeggio
  \arpeggioDown
  <c e g c>\arpeggio
}
```



Commonly tweaked properties

When an arpeggio crosses staves, you may attach an arpeggio to the chords in both staves and set `PianoStaff.connectArpeggios`

```
\new PianoStaff <<
  \set PianoStaff.connectArpeggios = ##t
  \new Staff { <c' e g c>\arpeggio }
  \new Staff { \clef bass <c,, e g>\arpeggio }
>>
```



Predefined commands

`\arpeggio`, `\arpeggioUp`, `\arpeggioDown`, `\arpeggioNeutral`, `\arpeggioBracket`.

See also

Notation manual: Section 6.5.1 [Ties], page 86, for writing out arpeggios.

Program reference: `Arpeggio`.

Bugs

It is not possible to mix connected arpeggios and unconnected arpeggios in one `PianoStaff` at the same point in time.

6.6.8 Falls and doits

Falls and doits can be added to notes using the `\bendAfter` command,

6.7 Repeats

Repetition is a central concept in music, and multiple notations exist for repetitions.

6.7.1 Repeat types

The following types of repetition are supported

`unfold` Repeated music is fully written (played) out. This is useful when entering repetitious music. This is the only kind of repeat that is included in MIDI output.

`volta` Repeats are not written out, but alternative endings (volte) are printed, left to right with brackets. This is the standard notation for repeats with alternatives. These are not played in MIDI output by default.

`tremolo` Make tremolo beams. These are not played in MIDI output by default.

`percent` Make beat or measure repeats. These look like percent signs. These are not played in MIDI output by default. Percent repeats must be declared within a `Voice` context.

6.7.2 Repeat syntax

LilyPond has one syntactic construct for specifying different types of repeats. The syntax is

```
\repeat variant repeatcount repeatbody
```

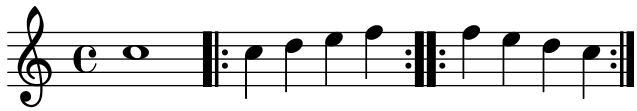
If you have alternative endings, you may add

```
\alternative {
  alternative1
  alternative2
  alternative3
  ...
}
```

where each *alternative* is a music expression. If you do not give enough alternatives for all of the repeats, the first alternative is assumed to be played more than once.

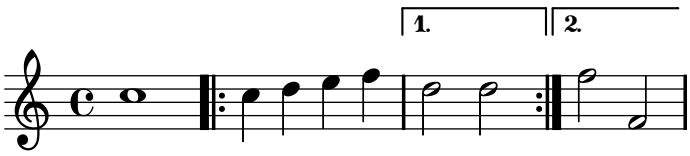
Standard repeats are used like this

```
c1
\repeat volta 2 { c4 d e f }
\repeat volta 2 { f e d c }
```



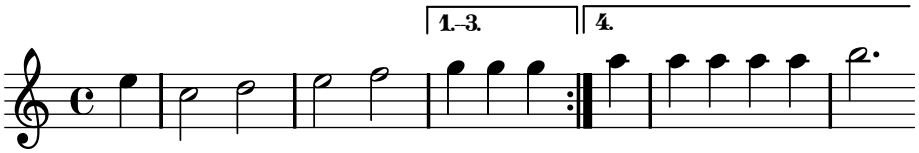
With alternative endings

```
c1
\repeat volta 2 {c4 d e f}
\alternative { {d2 d} {f f,} }
```



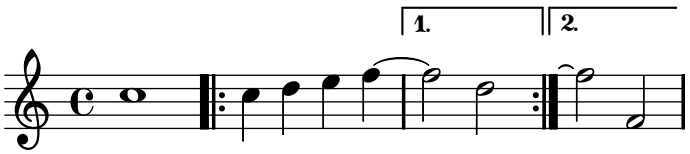
In the following example, the first ending is not a complete bar (it only had 3 beats). The beginning of the second ending contains the 4th beat from the first ending. This "extra" beat in the second ending is due to the first time ending, and has nothing to do with the `\partial` at the beginning of the example.

```
\new Staff {
  \partial 4
  \repeat volta 4 { e | c2 d2 | e2 f2 | }
  \alternative { { g4 g g } { a | a a a a | b2. } }
}
```



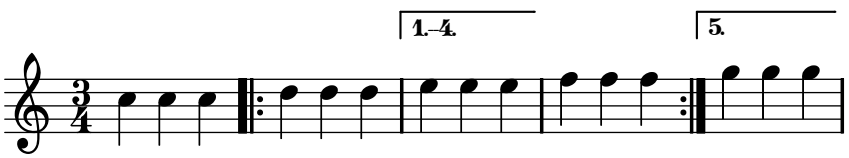
Ties may be added to a second ending,

```
c1
\repeat volta 2 {c4 d e f ~ }
\alternative { {f2 d} {f\repeatTie f,} }
```



It is possible to shorten volta brackets by setting `voltaSpannerDuration`. In the next example, the bracket only lasts one measure, which is a duration of 3/4.

```
\relative c''{
  \time 3/4
  c c c
  \set Staff.voltaSpannerDuration = #(ly:make-moment 3 4)
  \repeat "volta" 5 { d d d }
  \alternative { { e e e f f f }
  { g g g } }
}
```



See also

Examples:

Brackets for the repeat are normally only printed over the topmost staff. This can be adjusted by setting the `voltaOnThisStaff` property; see '`input/regression/volta-multi-staff.ly`'.

Bugs

A nested repeat like

```
\repeat ...
\repeat ...
\alternative
```

is ambiguous, since it is is not clear to which `\repeat` the `\alternative` belongs. This ambiguity is resolved by always having the `\alternative` belong to the inner `\repeat`. For clarity, it is advisable to use braces in such situations.

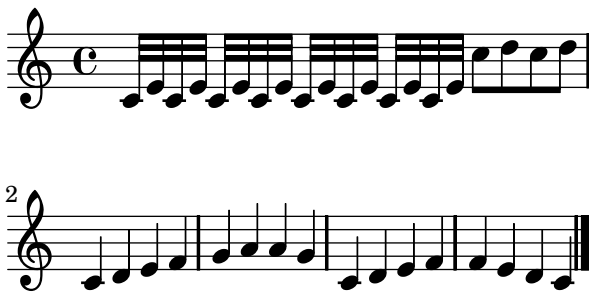
Timing information is not remembered at the start of an alternative, so after a repeat timing information must be reset by hand; for example, by setting `Score.measurePosition` or entering `\partial`. Similarly, slurs or ties are also not repeated.

Volta brackets are not vertically aligned.

6.7.3 Repeats and MIDI

With a little bit of tweaking, all types of repeats can be present in the MIDI output. This is achieved by applying the `\unfoldRepeats` music function. This function changes all repeats to unfold repeats.

```
\unfoldRepeats {
  \repeat tremolo 8 {c'32 e' }
  \repeat percent 2 { c''8 d'' }
  \repeat volta 2 {c'4 d' e' f'}
  \alternative {
    { g' a' a' g' }
    {f' e' d' c' }
  }
}
\bar "|."
```



When creating a score file using `\unfoldRepeats` for MIDI, it is necessary to make two `\score` blocks: one for MIDI (with unfolded repeats) and one for notation (with volta, tremolo, and percent repeats). For example,

```
\score {
  ..music..
  \layout { .. }
}
```

```
\score {
  \unfoldRepeats ..music..
  \midi { .. }
}
```

6.7.4 Manual repeat commands

The property `repeatCommands` can be used to control the layout of repeats. Its value is a Scheme list of repeat commands.

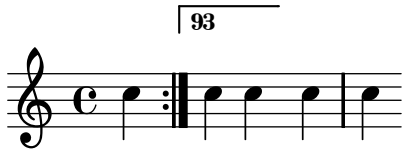
`start-repeat`
: Print a `|:` bar line.

`end-repeat`
: Print a `:|` bar line.

`(volta` *text*`)`
: Print a volta bracket saying *text*: The text can be specified as a text string or as a markup text, see Section 8.1.4 [Text markup], page 170. Do not forget to change the font, as the default number font does not contain alphabetic characters;

`(volta #f)`
: Stop a running volta bracket.

```
c4
  \set Score.repeatCommands = #'((volta "93") end-repeat)
c4 c4
  \set Score.repeatCommands = #'((volta #f))
c4 c4
```



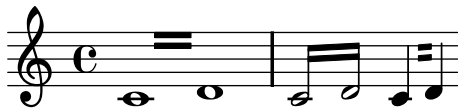
See also

Program reference: `VoltaBracket`, `RepeatedMusic`, `VoltaRepeatedMusic`, `UnfoldedRepeatedMusic`, and `FoldedRepeatedMusic`.

6.7.5 Tremolo repeats

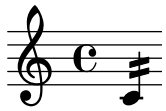
To place tremolo marks between notes, use `\repeat` with tremolo style

```
\new Voice \relative c' {
  \repeat "tremolo" 8 { c16 d16 }
  \repeat "tremolo" 4 { c16 d16 }
  \repeat "tremolo" 2 { c16 d16 }
}
```



Tremolo marks can also be put on a single note. In this case, the note should not be surrounded by braces.

```
\repeat "tremolo" 4 c'16
```



Similar output is obtained using the tremolo subdivision, described in Section 6.7.6 [Tremolo subdivisions], page 107.

See also

In this manual: Section 6.7.6 [Tremolo subdivisions], page 107, Section 6.7 [Repeats], page 103.

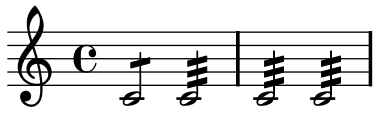
Program reference: `Beam`, `StemTremolo`.

Example files: ‘`input/regression/chord-tremolo.ly`’, ‘`input/regression/stem-tremolo.ly`’.

6.7.6 Tremolo subdivisions

Tremolo marks can be printed on a single note by adding ‘:[*number*]’ after the note. The number indicates the duration of the subdivision, and it must be at least 8. A *length* value of 8 gives one line across the note stem. If the length is omitted, the last value (stored in `tremoloFlags`) is used

```
c'2:8 c':32 | c': c': |
```



Bugs

Tremolos entered in this way do not carry over into the MIDI output.

See also

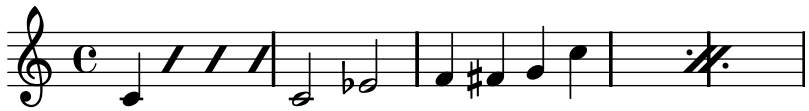
In this manual: Section 6.7.5 [Tremolo repeats], page 106.

Elsewhere: `StemTremolo`.

6.7.7 Measure repeats

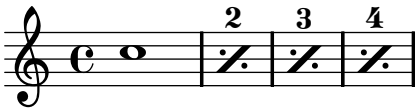
In the `percent` style, a note pattern can be repeated. It is printed once, and then the pattern is replaced with a special sign. Patterns of one and two measures are replaced by percent-like signs, patterns that divide the measure length are replaced by slashes. Percent repeats must be declared within a `Voice` context.

```
\new Voice \relative c' {
  \repeat "percent" 4 { c4 }
  \repeat "percent" 2 { c2 es2 f4 fis4 g4 c4 }
}
```



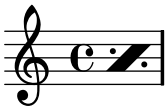
Measure repeats of more than 2 measures get a counter, if you switch on the `countPercentRepeats` property,

```
\new Voice {
\set countPercentRepeats = ##t
  \repeat "percent" 4 { c1 }
}
```



Isolated percents can also be printed. This is done by putting a multi-measure rest with a different print function,

```
\override MultiMeasureRest #'stencil
  = #ly:multi-measure-rest::percent
R1
```



See also

Program reference: `RepeatSlash`, `PercentRepeat`, `DoublePercentRepeat`, `DoublePercentRepeatCounter`, `PercentRepeatCounter`, `PercentRepeatedMusic`.

7 Instrument-specific notation

This chapter explains how to use notation for specific instruments.

7.1 Piano music

Piano staves are two normal staves coupled with a brace. The staves are largely independent, but sometimes voices can cross between the two staves. The same notation is also used for harps and other key instruments. The `PianoStaff` is especially built to handle this cross-staffing behavior. In this section we discuss the `PianoStaff` and some other pianistic peculiarities.

Bugs

Dynamics are not centered, but workarounds do exist. See the "piano centered dynamics" template in Section D.2 [Piano templates], page 320.

The distance between the two staves is the same for all systems in the score. It is possible to override this per system, but it does require an arcane command incantation. See '`input/test/piano-staff-distance.ly`'.

7.1.1 Automatic staff changes

Voices can be made to switch automatically between the top and the bottom staff. The syntax for this is

```
\autochange ...music...
```

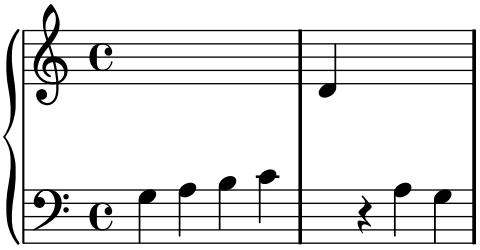
This will create two staves inside the current PianoStaff, called `up` and `down`. The lower staff will be in bass clef by default.

A `\relative` section that is outside of `\autochange` has no effect on the pitches of *music*, so, if necessary, put `\relative` inside `\autochange` like

```
\autochange \relative ... ...
```

The autochanger switches on basis of the pitch (middle C is the turning point), and it looks ahead skipping over rests to switch in advance. Here is a practical example

```
\new PianoStaff
  \autochange \relative c'
  {
    g4 a b c d r4 a g
  }
```



See also

In this manual: Section 7.1.2 [Manual staff switches], page 110.

Program reference: `AutoChangeMusic`.

Bugs

The staff switches may not end up in optimal places. For high quality output, staff switches should be specified manually.

`\autochange` cannot be inside `\times`.

7.1.2 Manual staff switches

Voices can be switched between staves manually, using the command

```
\change Staff = staffname music
```

The string *staffname* is the name of the staff. It switches the current voice from its current staff to the Staff called *staffname*. Typically *staffname* is `"up"` or `"down"`. The `Staff` referred to must already exist, so usually the setup for a score will start with a setup of the staves,

```
<<
  \new Staff = "up" {
    \skip 1 * 10  % keep staff alive
    }
  \new Staff = "down" {
    \skip 1 * 10  % idem
    }
>>
```

and the `Voice` is inserted afterwards

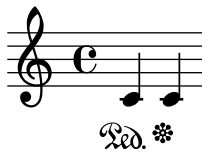
```
\context Staff = down
  \new Voice { ... \change Staff = up ... }
```

7.1.3 Pedals

Pianos have pedals that alter the way sound is produced. Generally, a piano has three pedals, sustain, una corda, and sostenuto.

Piano pedal instruction can be expressed by attaching `\sustainDown`, `\sustainUp`, `\unaCorda`, `\treCorde`, `\sostenutoDown` and `\sostenutoUp` to a note or chord

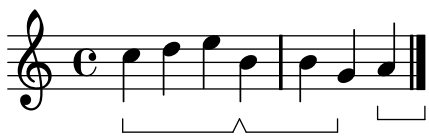
```
c'4\sustainDown c'4\sustainUp
```



What is printed can be modified by setting `pedal`*X*`Strings`, where *X* is one of the pedal types: `Sustain`, `Sostenuto` or `UnaCorda`. Refer to `SustainPedal` in the program reference for more information.

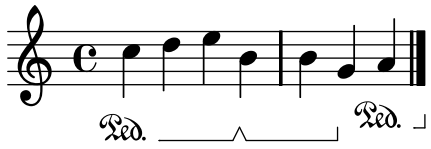
Pedals can also be indicated by a sequence of brackets, by setting the `pedalSustainStyle` property to bracket objects

```
\set Staff.pedalSustainStyle = #'bracket
c\sustainDown d e
b\sustainUp\sustainDown
b g \sustainUp a \sustainDown \bar "|."
```



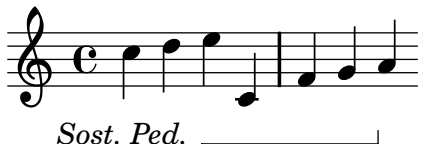
A third style of pedal notation is a mixture of text and brackets, obtained by setting the `pedalSustainStyle` property to `mixed`

```
\set Staff.pedalSustainStyle = #'mixed
c\sustainDown d e
b\sustainUp\sustainDown
b g \sustainUp a \sustainDown \bar "|."
```



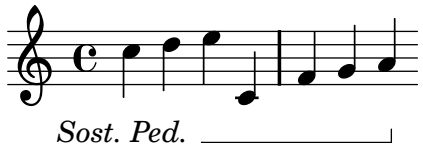
The default '*Ped.' style for sustain and damper pedals corresponds to style `#'text`. The sostenuto pedal uses `mixed` style by default.

```
c\sostenutoDown d e c, f g a\sostenutoUp
```



For fine-tuning the appearance of a pedal bracket, the properties `edge-width`, `edge-height`, and `shorten-pair` of `PianoPedalBracket` objects (see `PianoPedalBracket` in the Program reference) can be modified. For example, the bracket may be extended to the right edge of the note head

```
\override Staff.PianoPedalBracket #'shorten-pair = #'(0 . -1.0)
c\sostenutoDown d e c, f g a\sostenutoUp
```



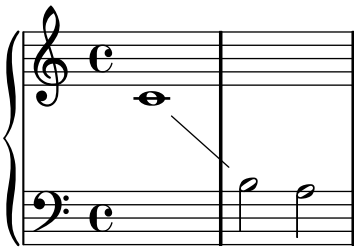
See also

In this manual: Section 6.5.4 [Laissez vibrer ties], page 89

7.1.4 Staff switch lines

Whenever a voice switches to another staff, a line connecting the notes can be printed automatically. This is switched on by setting `followVoice` to true

```
\new PianoStaff <<
  \new Staff="one" {
    \set followVoice = ##t
    c1
    \change Staff=two
    b2 a
  }
  \new Staff="two" { \clef bass \skip 1*2 }
>>
```



See also

Program reference: `VoiceFollower`.

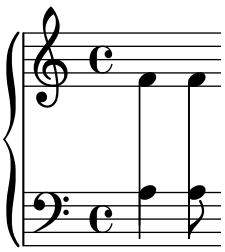
Predefined commands

`\showStaffSwitch`, `\hideStaffSwitch`.

7.1.5 Cross staff stems

Chords that cross staves may be produced by increasing the length of the stem in the lower staff, so it reaches the stem in the upper staff, or vice versa.

```
stemExtend = \once \override Stem #'length = #22
noFlag = \once \override Stem #'flag-style = #'no-flag
\new PianoStaff <<
  \new Staff {
    \stemDown \stemExtend
    f'4
    \stemExtend \noFlag
    f'8
  }
  \new Staff {
    \clef bass
    a4 a8
  }
>>
```



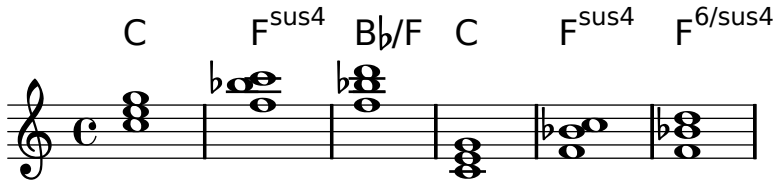
7.2 Chord names

7.2.1 Introducing chord names

LilyPond has support for printing chord names. Chords may be entered in musical chord notation, i.e., `< .. >`, but they can also be entered by name. Internally, the chords are represented as a set of pitches, so they can be transposed

```
twoWays = \transpose c c' {
  \chordmode {
    c1 f:sus4 bes/f
  }
```

```
  <c e g>
  <f bes c'>
  <f bes d'>
}

<< \new ChordNames \twoWays
   \new Voice \twoWays >>
```



This example also shows that the chord printing routines do not try to be intelligent. The last chord (`f bes d`) is not interpreted as an inversion.

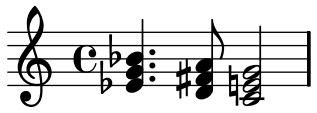
Note that the duration of chords must be specified outside the `<>`.

```
<c e g>2
```

7.2.2 Chords mode

In chord mode sets of pitches (chords) are entered with normal note names. A chord is entered by the root, which is entered like a normal pitch

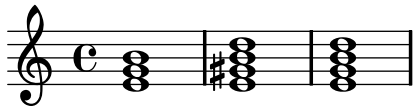
```
\chordmode { es4. d8 c2 }
```



The mode is introduced by the keyword `\chordmode`.

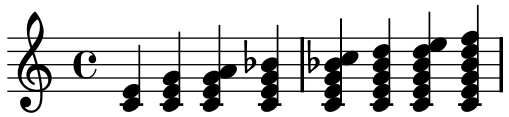
Other chords may be entered by suffixing a colon and introducing a modifier (which may include a number if desired)

```
\chordmode { e1:m e1:7 e1:m7 }
```



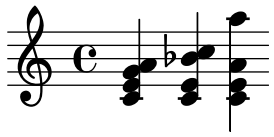
The first number following the root is taken to be the 'type' of the chord, thirds are added to the root until it reaches the specified number

```
\chordmode { c:3 c:5 c:6 c:7 c:8 c:9 c:10 c:11 }
```



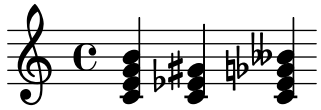
More complex chords may also be constructed adding separate steps to a chord. Additions are added after the number following the colon and are separated by dots

```
\chordmode { c:5.6 c:3.7.8 c:3.6.13 }
```



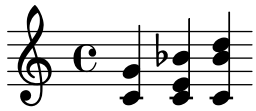
Chord steps can be altered by suffixing a `-` or `+` sign to the number

```
\chordmode { c:7+ c:5+.3- c:3-.5-.7- }
```



Removals are specified similarly and are introduced by a caret. They must come after the additions

```
\chordmode { c^3 c:7^5 c:9^3.5 }
```



Modifiers can be used to change pitches. The following modifiers are supported

`m` The minor chord. This modifier lowers the 3rd and (if present) the 7th step.

`dim` The diminished chord. This modifier lowers the 3rd, 5th and (if present) the 7th step.

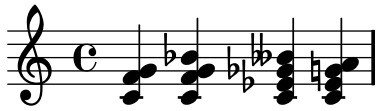
`aug` The augmented chord. This modifier raises the 5th step.

`maj` The major 7th chord. This modifier raises the 7th step if present.

`sus` The suspended 4th or 2nd. This modifier removes the 3rd step. Append either `2` or `4` to add the 2nd or 4th step to the chord.

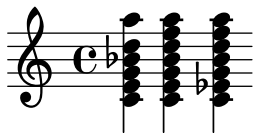
Modifiers can be mixed with additions

```
\chordmode { c:sus4 c:7sus4 c:dim7 c:m6 }
```



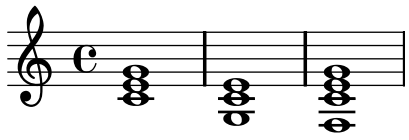
Since an unaltered 11 does not sound good when combined with an unaltered 13, the 11 is removed in this case (unless it is added explicitly)

```
\chordmode { c:13 c:13.11 c:m13 }
```



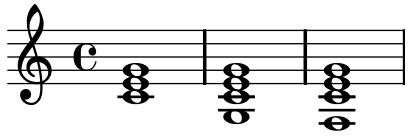
An inversion (putting one pitch of the chord on the bottom), as well as bass notes, can be specified by appending */pitch* to the chord

```
\chordmode { c1 c/g c/f }
```



A bass note can be added instead of transposed out of the chord, by using */+pitch*.

```
\chordmode { c1 c/+g c/+f }
```

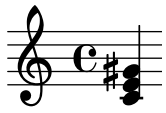


Chords is a mode similar to `\lyricmode`, etc. Most of the commands continue to work, for example, `r` and `\skip` can be used to insert rests and spaces, and property commands may be used to change various settings.

Bugs

Each step can only be present in a chord once. The following simply produces the augmented chord, since `5+` is interpreted last

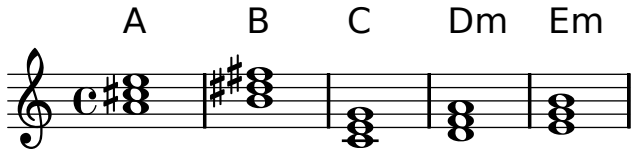
```
\chordmode { c:5.5-.5+ }
```



7.2.3 Printing chord names

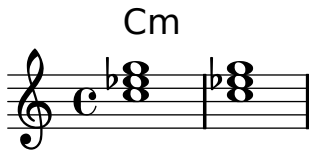
For displaying printed chord names, use the `ChordNames` context. The chords may be entered either using the notation described above, or directly using `<` and `>`

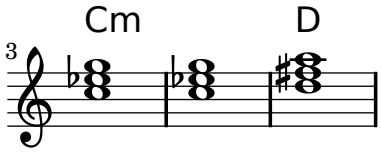
```
harmonies = {
  \chordmode {a1 b c} <d' f' a'> <e' g' b'>
}
<<
  \new ChordNames \harmonies
  \new Staff \harmonies
>>
```



You can make the chord changes stand out by setting `ChordNames.chordChanges` to true. This will only display chord names when there is a change in the chords scheme and at the start of a new line

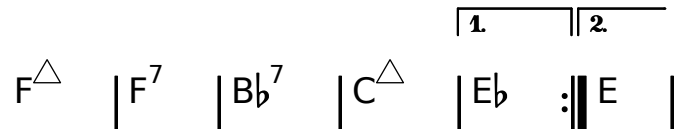
```
harmonies = \chordmode {
  c1:m c:m \break c:m c:m d
}
<<
  \new ChordNames {
    \set chordChanges = ##t
    \harmonies }
  \new Staff \transpose c c' \harmonies
>>
```





The previous examples all show chords over a staff. This is not necessary. Chords may also be printed separately. It may be necessary to add `Volta_engraver` and `Bar_engraver` for showing repeats.

```
\new ChordNames \with {
  \override BarLine #'bar-size = #4
  voltaOnThisStaff = ##t
  \consists Bar_engraver
  \consists "Volta_engraver"
}
\chordmode { \repeat volta 2 {
  f1:maj7 f:7 bes:7
  c:maj7
} \alternative {
  es e
}
}
```



The default chord name layout is a system for Jazz music, proposed by Klaus Ignatzek (see Appendix A [Literature list], page 309). It can be tuned through the following properties

`chordNameExceptions`
: This is a list that contains the chords that have special formatting.

  The exceptions list should be encoded as

  ```
  { <c f g bes>1 \markup { \super "7" "wahh" } }
  ```

  To get this information into `chordNameExceptions` takes a little manoeuvring. The following code transforms `chExceptionMusic` (which is a sequential music) into a list of exceptions.

  ```
  (sequential-music-to-chord-exceptions chExceptionMusic #t)
  ```

  Then,

  ```
  (append
   (sequential-music-to-chord-exceptions chExceptionMusic #t)
   ignatzekExceptions)
  ```

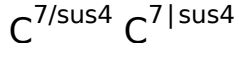
  adds the new exceptions to the default ones, which are defined in '`ly/chord-modifier-init.ly`'.

  For an example of tuning this property, see also '`input/regression/chord-name-exceptions.ly`'.

`majorSevenSymbol`
: This property contains the markup object used for the 7th step, when it is major. Predefined options are `whiteTriangleMarkup` and `blackTriangleMarkup`. See '`input/regression/chord-name-major7.ly`' for an example.

`chordNameSeparator`
: Different parts of a chord name are normally separated by a slash. By setting `chordNameSeparator`, you can specify other separators, e.g.,

  ```
  \new ChordNames \chordmode {
    c:7sus4
    \set chordNameSeparator
      = \markup { \typewriter "|" }
    c:7sus4
  }
  ```

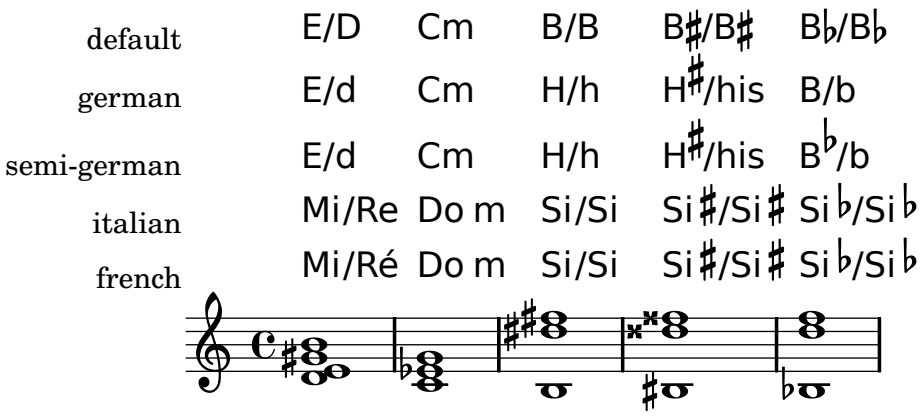
  

`chordRootNamer`
: The root of a chord is usually printed as a letter with an optional alteration. The transformation from pitch to letter is done by this function. Special note names (for example, the German "H" for a B-chord) can be produced by storing a new function in this property.

`chordNoteNamer`
: The default is to print single pitch, e.g., the bass note, using the `chordRootNamer`. The `chordNoteNamer` property can be set to a specialized function to change this behavior. For example, the base can be printed in lower case.

`chordPrefixSpacer`
: The "m" for minor chords is usually printed right after the root of the chord. By setting `chordPrefixSpacer`, you can fix a spacer between the root and "m". The spacer is not used when the root is altered.

The predefined variables `\germanChords`, `\semiGermanChords`, `\italianChords` and `\frenchChords` set these variables. The effect is demonstrated here,



There are also two other chord name schemes implemented: an alternate Jazz chord notation, and a systematic scheme called Banter chords. The alternate Jazz notation is also shown on the chart in Section C.1 [Chord name chart], page 312. Turning on these styles is described in the input file '`input/test/chord-names-jazz.ly`'.

Predefined commands

`\germanChords`, `\semiGermanChords`. `\italianChords`. `\frenchChords`.

See also

Examples: '`input/regression/chord-name-major7.ly`', '`input/regression/chord-name-exceptions.ly`', '`input/test/chord-names-jazz.ly`'.

Init files: '`scm/chords-ignatzek.scm`', and '`scm/chord-entry.scm`'.

Bugs

Chord names are determined solely from the list of pitches. Chord inversions are not identified, and neither are added bass notes. This may result in strange chord names when chords are entered with the `< .. >` syntax.

7.3 Vocal music

Since LilyPond input files are text, there are two issues to consider when working with vocal music:

- Song texts must be entered as text, not notes. For example, the input `d` should be interpreted as a one letter syllable, not the note D.
- Song texts must be aligned with the notes of their melody.

There are a few different ways to define lyrics; the simplest way is to use the `\addlyrics` function.

Commonly tweaked properties

Checking to make sure that text scripts and lyrics are within the margins is a relatively large computational task. To speed up processing, lilypond does not perform such calculations by default; to enable it, use

```
\override Score.PaperColumn #'keep-inside-line = ##t
```

To make lyrics avoid barlines as well, use

```
\layout {
  \context {
    \Lyrics
      \consists "Bar_engraver"
      \consists "Separating_line_group_engraver"
```

```
        \override BarLine #'transparent = ##t
    }
  }
```

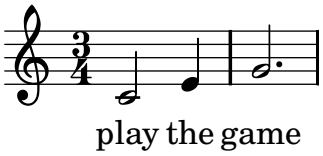
7.3.1 Setting simple songs

The easiest way to add lyrics to a melody is to append

```
\addlyrics { the lyrics }
```

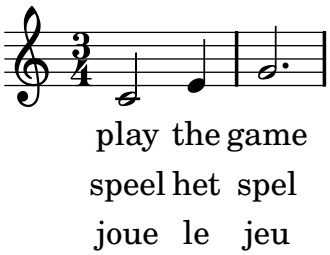
to a melody. Here is an example,

```
\time 3/4
\relative { c2 e4 g2. }
\addlyrics { play the game }
```



More stanzas can be added by adding more `\addlyrics` sections

```
\time 3/4
\relative { c2 e4 g2. }
\addlyrics { play the game }
\addlyrics { speel het spel }
\addlyrics { joue le jeu }
```



The command `\addlyrics` cannot handle polyphony settings. For these cases you should use `\lyricsto` and `\lyricmode`.

7.3.2 Entering lyrics

Lyrics are entered in a special input mode. This mode is introduced by the keyword `\lyricmode`, or by using `\addlyrics` or `\lyricsto`. In this mode you can enter lyrics, with punctuation and accents, and the input `d` is not parsed as a pitch, but rather as a one letter syllable. Syllables are entered like notes, but with pitches replaced by text. For example,

```
\lyricmode { Twin-4 kle4 twin- kle litt- le star2 }
```

There are two main methods to specify the horizontal placement of the syllables, either by specifying the duration of each syllable explicitly, like in the example above, or by automatically aligning the lyrics to a melody or other voice of music, using `\addlyrics` or `\lyricsto`. For more details see Section 7.3.4 [The Lyrics context], page 121.

A word or syllable of lyrics begins with an alphabetic character, and ends with any space or digit. The following characters can be any character that is not a digit or white space. One important consequence of this is that a word can end with `}`. The following example is usually a mistake in the input file. The syllable includes a `}`, so the opening brace is not balanced

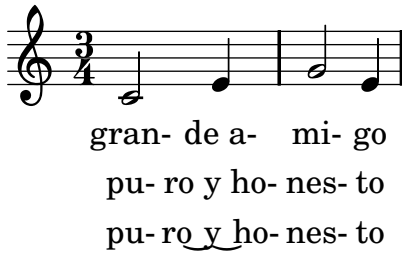
```
\lyricmode { twinkle}
```

Similarly, a period which follows an alphabetic sequence is included in the resulting string. As a consequence, spaces must be inserted around property commands

```
\override Score . LyricText #'font-shape = #'italic
```

In order to assign more than one syllable to a single note, you can surround them with quotes or use a _ character, to get spaces between syllables, or use tilde symbol (~) to get a lyric tie.

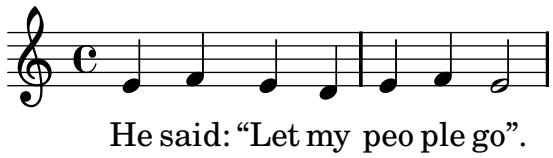
```
\time 3/4
\relative { c2 e4 g2 e4 }
\addlyrics { gran- de_a- mi- go }
\addlyrics { pu- "ro y ho-" nes- to }
\addlyrics { pu- ro~y~ho- nes- to }
```



The lyric ties is implemented with the Unicode character U+203F, so be sure to have a font (Like DejaVuLGC) installed that includes this glyph.

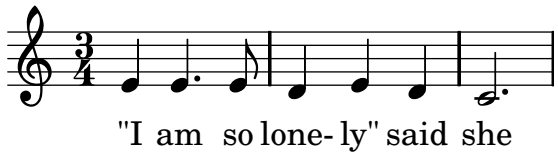
To enter lyrics with characters from non-English languages, or with accented and special characters (such as the heart symbol or slanted quotes), simply insert the characters directly into the input file and save it with utf-8 encoding. See Section 10.1.7 [Text encoding], page 238 for more info.

```
\relative { e4 f e d e f e2 }
\addlyrics { He said:   Let my peo ple go . }
```



To use normal quotes in lyrics, add a backslash before the quotes. For example,

```
\relative c' { \time 3/4 e4 e4. e8 d4 e d c2. }
\addlyrics { "\"I" am so lone- "ly\"" said she }
```



The full definition of a word start in Lyrics mode is somewhat more complex.

A word in Lyrics mode begins with: an alphabetic character, _, ?, !, :, ', the control characters ^A through ^F, ^Q through ^W, ^Y, ^^, any 8-bit character with ASCII code over 127, or a two-character combination of a backslash followed by one of ‘', , ", or ^.

To define indentifiers containing lyrics, the function `lyricmode` must be used.

```
verseOne = \lyricmode { Joy to the world the Lord is come }
\score {
  <<
    \new Voice = "one" \relative c'' {
      \autoBeamOff
      \time 2/4
      c4 b8. a16 g4. f8 e4 d c2
    }
    \addlyrics { \verseOne }
  >>
}
```

See also

Program reference: `LyricText`, `LyricSpace`.

7.3.3 Hyphens and extenders

Centered hyphens are entered as '`--`' between syllables. The hyphen will have variable length depending on the space between the syllables and it will be centered between the syllables.

When a lyric is sung over many notes (this is called a melisma), this is indicated with a horizontal line centered between a syllable and the next one. Such a line is called an extender line, and it is entered as '`__`'.

In tighly engraved music, hyphens can be removed. Whether this happens can be controlled with the `minimum-distance` (minimum distance between two syllables) and the `minimum-length` (threshold below which hyphens are removed).

See also

Program reference: `LyricHyphen`, `LyricExtender`.

7.3.4 The Lyrics context

Lyrics are printed by interpreting them in the context called `Lyrics`.

```
\new Lyrics \lyricmode ...
```

This will place the lyrics according to the durations that were entered. The lyrics can also be aligned under a given melody automatically. In this case, it is no longer necessary to enter the correct duration for each syllable. This is achieved by combining the melody and the lyrics with the `\lyricsto` expression

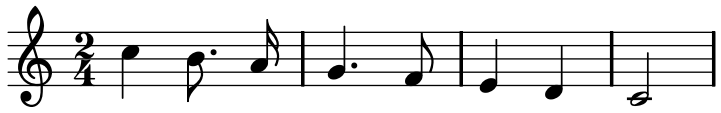
```
\new Lyrics \lyricsto name ...
```

This aligns the lyrics to the notes of the `Voice` context called *name*, which must already exist. Therefore normally the `Voice` is specified first, and then the lyrics are specified with `\lyricsto`. The command `\lyricsto` switches to `\lyricmode` mode automatically, so the `\lyricmode` keyword may be omitted.

The following example uses different commands for entering lyrics.

```
<<
  \new Voice = "one" \relative c'' {
    \autoBeamOff
    \time 2/4
    c4 b8. a16 g4. f8 e4 d c2
  }
  \new Lyrics \lyricmode { Joy4 to8. the16 world!4. the8 Lord4 is come.2 }
  \new Lyrics \lyricmode { Joy to the earth! the Sa -- viour reigns. }
```

```
  \new Lyrics \lyricsto "one" { No more let sins and sor -- rows grow. }
>>
```



Joy to the world! the Lord is come.
Joy to the earth! the Sa - viour
No more let sins and sor-rows grow.

8
reigns.

The second stanza is not properly aligned because the durations were not specified. A solution for that would be to use `\lyricsto`.

The `\addlyrics` command is actually just a convenient way to write a more complicated LilyPond structure that sets up the lyrics.

```
{ MUSIC }
\addlyrics { LYRICS }
```

is the same as

```
\new Voice = "blah" { music }
\new Lyrics \lyricsto "blah" { LYRICS }
```

For different or more complex orderings, the best way is to setup the hierarchy of staves and lyrics first, e.g.,

```
\new ChoirStaff <<
  \new Voice = "soprano" { music }
  \new Lyrics = "sopranoLyrics" { s1 }
  \new Lyrics = "tenorLyrics" { s1 }
  \new Voice = "tenor" { music }
>>
```

and then combine the appropriate melodies and lyric lines

```
\context Lyrics = sopranoLyrics \lyricsto "soprano"
  the lyrics
```

The final input would resemble

```
<<\new ChoirStaff << setup the music >>
  \lyricsto "soprano" etc
  \lyricsto "alto" etc
  etc
>>
```

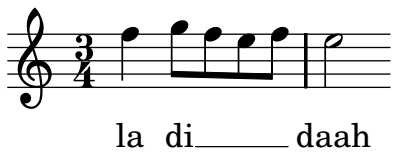
See also

Program reference: `LyricCombineMusic`, `Lyrics`.

7.3.5 Melismata

The `\lyricsto` command detects melismata: it only puts one syllable under a tied or slurred group of notes. If you want to force an unslurred group of notes to be a melisma, insert `\melisma` after the first note of the group, and `\melismaEnd` after the last one, e.g.,

```
<<
  \new Voice = "lala" {
    \time 3/4
    f4 g8
    \melisma
    f e f
    \melismaEnd
    e2
  }
  \new Lyrics \lyricsto "lala" {
    la di __ daah
  }
>>
```



In addition, notes are considered a melisma if they are manually beamed, and automatic beaming (see Section 9.1.2 [Setting automatic beam behavior], page 215) is switched off.

A complete example of a SATB score setup is in section Section D.4 [Vocal ensembles], page 329.

Predefined commands

`\melisma`, `\melismaEnd`

See also

Program reference: `Melisma_translator`.

‘`input/regression/lyric-combine-new.ly`’.

Bugs

Melismata are not detected automatically, and extender lines must be inserted by hand.

7.3.6 Another way of entering lyrics

Lyrics can also be entered without `\lyricsto`. In this case the duration of each syllable must be entered explicitly, for example,

```
play2 the4 game2.
sink2 or4 swim2.
```

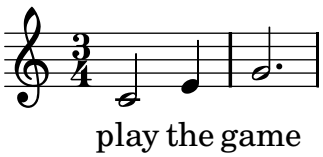
The alignment to a melody can be specified with the `associatedVoice` property,

```
\set associatedVoice = #"lala"
```

The value of the property (here: `"lala"`) should be the name of a `Voice` context. Without this setting, extender lines will not be formatted properly.

Here is an example demonstrating manual lyric durations,

```
<< \new Voice = "melody" {
     \time 3/4
     c2 e4 g2.
  }
  \new Lyrics \lyricmode {
    \set associatedVoice = #"melody"
    play2 the4 game2.
  } >>
```



7.3.7 Flexibility in placement

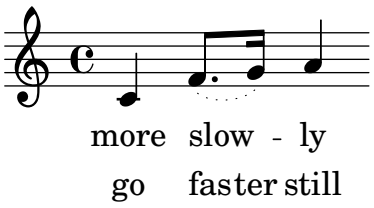
Often, different stanzas of one song are put to one melody in slightly differing ways. Such variations can still be captured with `\lyricsto`.

7.3.7.1 Lyrics to multiple notes of a melisma

One possibility is that the text has a melisma in one stanza, but multiple syllables in another one. One solution is to make the faster voice ignore the melisma. This is done by setting `ignoreMelismata` in the Lyrics context.

There is one tricky aspect: the setting for `ignoreMelismata` must be set one syllable *before* the non-melismatic syllable in the text, as shown here,

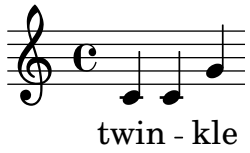
```
<<
  \relative \new Voice = "lahlah" {
    \set Staff.autoBeaming = ##f
    c4
    \slurDotted
    f8.[( g16])
    a4
  }
  \new Lyrics \lyricsto "lahlah" {
    more slow -- ly
  }
  \new Lyrics \lyricsto "lahlah" {
    \set ignoreMelismata = ##t % applies to "fas"
    go fas -- ter
    \unset ignoreMelismata
    still
  }
>>
```



The `ignoreMelismata` applies to the syllable “fas”, so it should be entered before “go”.

The reverse is also possible: making a lyric line slower than the standard. This can be achieved by insert `\skip`s into the lyrics. For every `\skip`, the text will be delayed another note. For example,

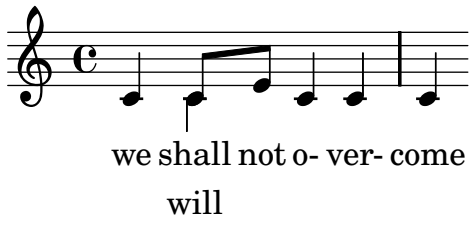
```
\relative { c c g' }
\addlyrics {
  twin -- \skip 4
  kle
}
```



7.3.7.2 Divisi lyrics

You can display alternate (or divisi) lyrics by naming voice contexts and attaching lyrics to those specific contexts.

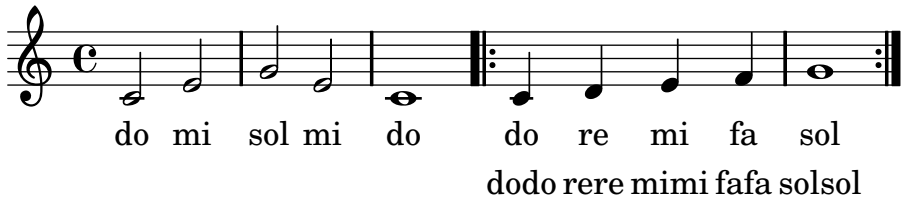
```
\score{ <<
  \new Voice = "melody" {
    \relative c' {
      c4
      <<
        { \voiceOne c8 e }
        \new Voice = "splitpart" { \voiceTwo c4 }
      >>
      \oneVoice c4 c | c
    }
  }
  \new Lyrics \lyricsto "melody" { we shall not o- ver- come }
  \new Lyrics \lyricsto "splitpart" { will }
>> }
```

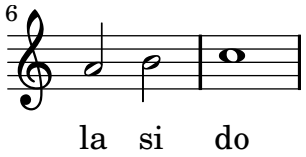


You can use this trick to display different lyrics for a repeated section.

```
\score{ <<
  \new Voice = "melody" \relative c' {
    c2 e | g e | c1 |
    \new Voice = "verse" \repeat volta 2 {c4 d e f | g1 | }
    a2 b | c1}
  \new Lyrics = "mainlyrics" \lyricsto melody \lyricmode {
    do mi sol mi do
    la si do }
```

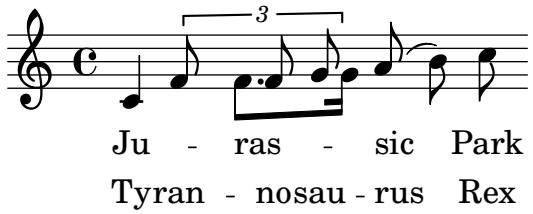
```
  \context Lyrics = "mainlyrics" \lyricsto verse \lyricmode {
   do re mi fa sol }
  \new Lyrics = "repeatlyrics" \lyricsto verse \lyricmode {
   dodo rere mimi fafa solsol }
>>
}
```





7.3.7.3 Switching the melody associated with a lyrics line

More complex variations in text underlay are possible. It is possible to switch the melody for a line of lyrics during the text. This is done by setting the `associatedVoice` property. In the example



the text for the first stanza is set to a melody called “lahlah”,

```
\new Lyrics \lyricsto "lahlah" {
  Ju -- ras -- sic Park
}
```

The second stanza initially is set to the `lahlah` context, but for the syllable “ran”, it switches to a different melody. This is achieved with

```
\set associatedVoice = alternative
```

Here, `alternative` is the name of the `Voice` context containing the triplet.

Again, the command must be one syllable too early, before “Ty” in this case.

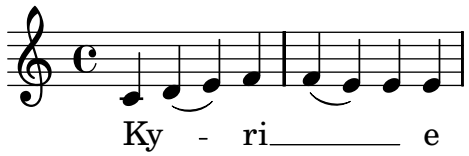
```
\new Lyrics \lyricsto "lahlah" {
  \set associatedVoice = alternative % applies to "ran"
  Ty --
  ran --
  no --
  \set associatedVoice = lahlah % applies to "rus"
  sau -- rus Rex
}
```

The underlay is switched back to the starting situation by assigning `lahlah` to `associatedVoice`.

7.3.7.4 Specifying melismata within the lyrics

It is also possible to define melismata entirely in the lyrics. This can be done by entering _ for every note that is part of the melisma.

```
{ \set melismaBusyProperties = #'()
  c d( e) f f( e) e e  }
\addlyrics
 { Ky -- _ _ ri __ _ _ _  e }
```

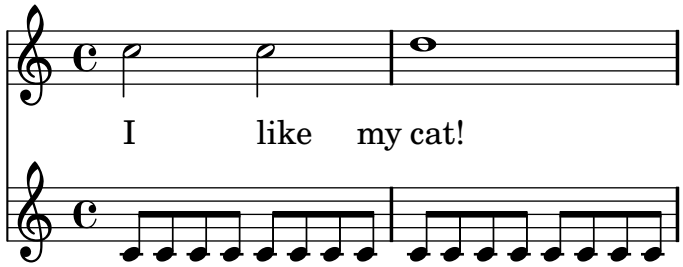


In this case, you can also have ties and slurs in the melody if you set `melismaBusyProperties`, as is done in the example above.

7.3.7.5 Lyrics independent of notes

In some complex vocal music, it may be desirable to place lyrics completely independently of notes. Music defined inside `lyricrhythm` disappears into the `Devnull` context, but the rhythms can still be used to place the lyrics.

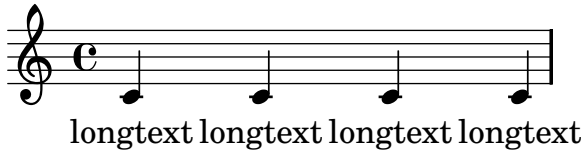
```
voice = {
  c''2
  \tag #'music { c''2 }
  \tag #'lyricrhythm { c''4. c''8 }
  d''1
}

lyr = \lyricmode { I like my cat! }

<<
  \new Staff \keepWithTag #'music \voice
  \new Devnull="nowhere" \keepWithTag #'lyricrhythm \voice
  \new Lyrics \lyricsto "nowhere" \lyr
  \new Staff { c'8 c' c' c' c' c' c' c'
  c' c' c' c' c' c' c' c' }
>>
```

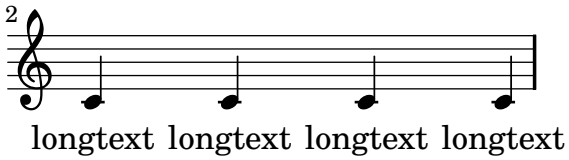


7.3.8 Spacing lyrics

To increase the spacing between lyrics, set the minimum-distance property of LyricSpace.

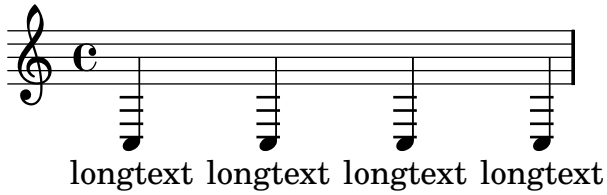
```
{
  c c c c
  \override Lyrics.LyricSpace #'minimum-distance = #1.0
  c c c c
}
\addlyrics {
  longtext longtext longtext longtext
  longtext longtext longtext longtext
}
```

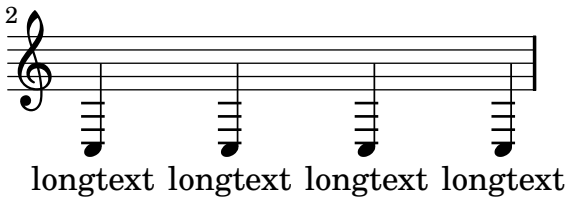




To make this change for all lyrics in the score, set the property in the layout.

```
\score {
  {
  c c c c
  c c c c
  }
  \addlyrics {
  longtext longtext longtext longtext
  longtext longtext longtext longtext
  }
  \layout {
    \context {
      \Lyrics
      \override LyricSpace #'minimum-distance = #1.0
    }
  }
}
```



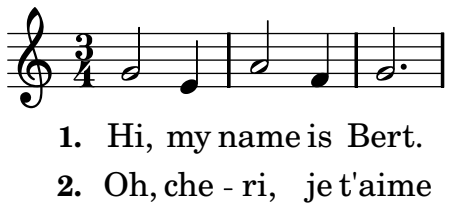


7.3.9 More about stanzas

7.3.9.1 Adding stanza numbers

Stanza numbers can be added by setting `stanza`, e.g.,

```
\new Voice {
  \time 3/4 g2 e4 a2 f4 g2.
} \addlyrics {
  \set stanza = "1. "
  Hi, my name is Bert.
} \addlyrics {
  \set stanza = "2. "
  Oh, che -- ri, je t'aime
}
```

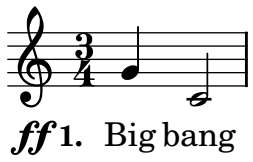


These numbers are put just before the start of the first syllable.

7.3.9.2 Adding dynamics marks

Stanzas differing in loudness may be indicated by putting a dynamics mark before each stanza. In Lilypond, everthing coming in front of a stanza goes into the `StanzaNumber` object; dynamics marks are no different. For technical reasons, you have to set the stanza outside `\lyricmode`:

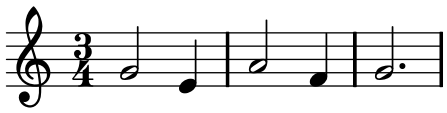
```
text = {
  \set stanza = \markup { \dynamic "ff" "1. " }
  \lyricmode {
    Big bang
  }
}

<<
  \new Voice = "tune" {
    \time 3/4
    g'4 c'2
  }
\new Lyrics \lyricsto "tune" \text
>>
```



7.3.9.3 Adding singer names

Names of singers can also be added. They are printed at the start of the line, just like instrument names. They are created by setting `vocalName`. A short version may be entered as `shortVocalName`.

```
\new Voice {
  \time 3/4 g2 e4 a2 f4 g2.
} \addlyrics {
  \set vocalName = "Bert "
  Hi, my name is Bert.
} \addlyrics {
  \set vocalName = "Ernie "
  Oh, che -- ri, je t'aime
}
```



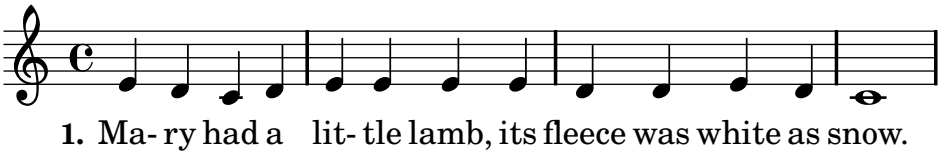
Bert Hi, my name is Bert.
Ernie Oh, che - ri, je t'aime

7.3.9.4 Printing stanzas at the end

Sometimes it is appropriate to have one stanza set to the music, and the rest added in verse form at the end of the piece. This can be accomplished by adding the extra verses into a `\markup` section outside of the main score block. Notice that there are two different ways to force linebreaks when using `\markup`.

```
melody = \relative c' {
e d c d | e e e e |
d d e d | c1 |
}

text = \lyricmode {
\set stanza = "1." Ma- ry had a lit- tle lamb,
its fleece was white as snow.
}

\book{
  \score{ <<
    \new Voice = "one" { \melody }
    \new Lyrics \lyricsto "one" \text
        >>
    \layout { }
  }
  \markup { \column{
    \line{ Verse 2. }
    \line{ All the children laughed and played }
    \line{ To see a lamb at school. }
    }
  }
  \markup{
```

```
      \wordwrap-string #"
      Verse 3.

      Mary took it home again,

      It was against the rule."
    }
  }
```



Verse 2.
All the children laughed and played
To see a lamb at school.

Verse 3.
Mary took it home again,
It was against the rule.

7.3.9.5 Printing stanzas at the end in multiple columns

When a piece of music has many verses, they are often printed in multiple columns across the page. An outdented verse number often introduces each verse. The following example shows how to produce such output in Lilypond.

```
melody = \relative c' {
  c c c c | d d d d
}

text = \lyricmode {
  \set stanza = "1." This is verse one.
  It has two lines.
}

\score{ <<
    \new Voice = "one" { \melody }
    \new Lyrics \lyricsto "one" \text
   >>
  \layout { }
}

\markup {
  \fill-line {
    \hspace #0.1 % moves the column off the left margin; can be removed if
        % space on the page is tight
     \column {
      \line { \bold "2."
        \column {
          "This is verse two."
          "It has two lines."
        }
      }
      \hspace #0.1 % adds vertical spacing between verses
      \line { \bold "3."
        \column {
          "This is verse three."
          "It has two lines."
        }
      }
    }
    \hspace #0.1  % adds horizontal spacing between columns; if they are
        % still too close, add more " " pairs until the result
        % looks good
     \column {
      \line { \bold "4."
        \column {
          "This is verse four."
          "It has two lines."
        }
      }
      \hspace #0.1 % adds vertical spacing between verses
      \line { \bold "5."
```

```
        \column {
          "This is verse five."
          "It has two lines."
        }
      }
    }
  \hspace #0.1 % gives some extra space on the right margin; can
      % be removed if page space is tight
  }
}
```

2. This is verse two.
It has two lines.

3. This is verse three.
It has two lines.

4. This is verse four.
It has two lines.

5. This is verse five.
It has two lines.

See also

Program reference: `LyricText`, `StanzaNumber`, `VocalName`.

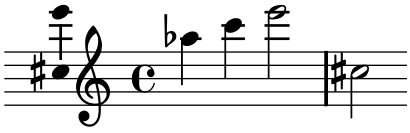
7.3.10 Ambitus

The term *ambitus* denotes a range of pitches for a given voice in a part of music. It may also denote the pitch range that a musical instrument is capable of playing. Ambits are printed on vocal parts, so performers can easily determine it meets their capabilities.

Ambits are denoted at the beginning of a piece near the initial clef. The range is graphically specified by two note heads that represent the minimum and maximum pitch. To print such ambits, add the `Ambitus_engraver` to the `Voice` context, for example,

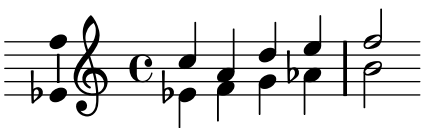
```
\layout {
  \context {
    \Voice
    \consists Ambitus_engraver
  }
}
```

This results in the following output



If you have multiple voices in a single staff and you want a single ambitus per staff rather than per each voice, add the `Ambitus_engraver` to the `Staff` context rather than to the `Voice` context. Here is an example,

```
\new Staff \with {
  \consists "Ambitus_engraver"
}
<<
  \new Voice \with {
    \remove "Ambitus_engraver"
  } \relative c'' {
    \override Ambitus #'X-offset = #-1.0
    \voiceOne
    c4 a d e f2
  }
  \new Voice \with {
    \remove "Ambitus_engraver"
  } \relative c' {
    \voiceTwo
    es4 f g as b2
  }
>>
```



This example uses one advanced feature,

```
\override Ambitus #'X-offset = #-1.0
```

This code moves the ambitus to the left. The same effect could have been achieved with `extra-offset`, but then the formatting system would not reserve space for the moved object.

See also

Program reference: `Ambitus`, `AmbitusLine`, `AmbitusNoteHead`, `AmbitusAccidental`.

Examples: '`input/regression/ambitus.ly`'.

Bugs

There is no collision handling in the case of multiple per-voice ambitus.

7.3.11 Other vocal issues

"Parlato" is spoken without pitch but still with rhythm; it is notated by cross noteheads. This is demonstrated in Section 8.4.5 [Special noteheads], page 205.

7.4 Rhythmic music

Rhythmic music is primarily used for percussion and drum notation, but it can also be used to show the rhythms of melodies.

7.4.1 Showing melody rhythms

Sometimes you might want to show only the rhythm of a melody. This can be done with the rhythmic staff. All pitches of notes on such a staff are squashed, and the staff itself has a single line

```
\new RhythmicStaff {
  \time 4/4
  c4 e8 f g2 | r4 g r2 | g1:32 | r1 |
}
```

See also

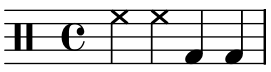
Program reference: `RhythmicStaff`.

Examples: '`input/regression/rhythmic-staff.ly`'.

7.4.2 Entering percussion

Percussion notes may be entered in `\drummode` mode, which is similar to the standard mode for entering notes. Each piece of percussion has a full name and an abbreviated name, and both can be used in input files

```
\drums {
  hihat hh bassdrum bd
}
```



The complete list of drum names is in the init file '`ly/drumpitch-init.ly`'.

See also

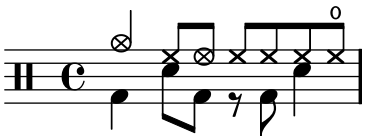
Program reference: `note-event`.

7.4.3 Percussion staves

A percussion part for more than one instrument typically uses a multiline staff where each position in the staff refers to one piece of percussion.

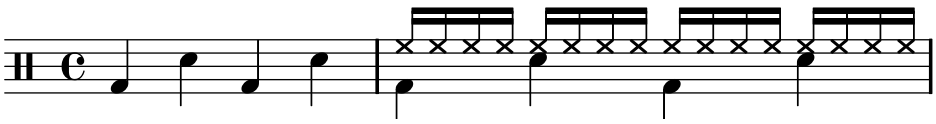
To typeset the music, the notes must be interpreted in a `DrumStaff` and `DrumVoice` contexts

```
up = \drummode { crashcymbal4 hihat8 halfopenhihat hh hh hh openhihat }
down = \drummode { bassdrum4 snare8 bd r bd sn4 }
  \new DrumStaff <<
    \new DrumVoice { \voiceOne \up }
    \new DrumVoice { \voiceTwo \down }
  >>
```



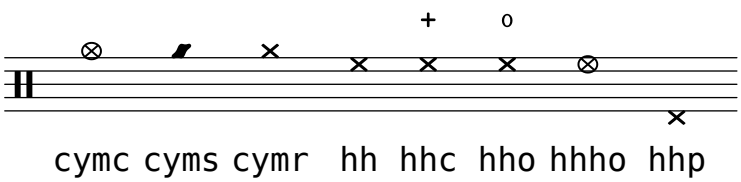
The above example shows verbose polyphonic notation. The short polyphonic notation, described in Section 6.3.3 [Basic polyphony], page 70, can also be used if the `DrumVoices` are instantiated by hand first. For example,

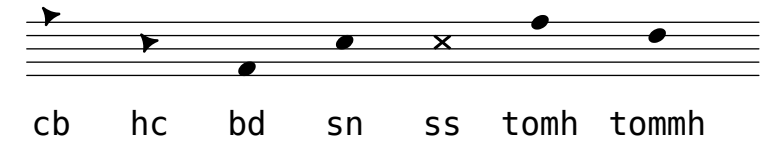
```
\new DrumStaff <<
  \new DrumVoice = "1" { s1 *2 }
  \new DrumVoice = "2" { s1 *2 }
  \drummode {
    bd4 sn4 bd4 sn4
    <<
      { \repeat unfold 16 hh16 }
      \\
      { bd4 sn4 bd4 sn4 }
    >>
  }
>>
```

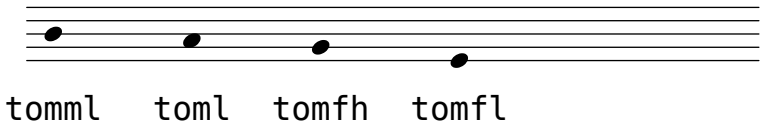


There are also other layout possibilities. To use these, set the property `drumStyleTable` in context `DrumVoice`. The following variables have been predefined

`drums-style`
: This is the default. It typesets a typical drum kit on a five-line staff

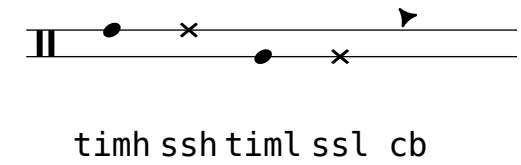




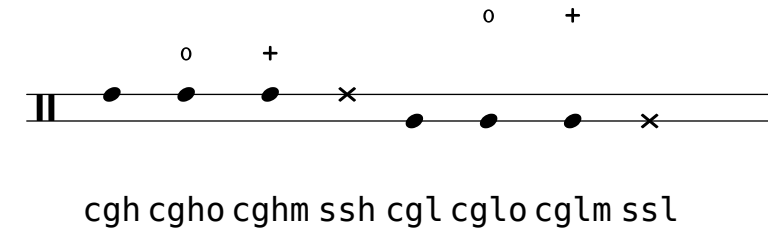


The drum scheme supports six different toms. When there are fewer toms, simply select the toms that produce the desired result, i.e., to get toms on the three middle lines you use `tommh`, `tomml`, and `tomfh`.

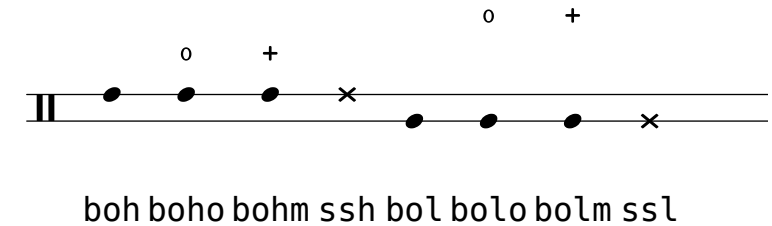
`timbales-style`
: This typesets timbales on a two line staff



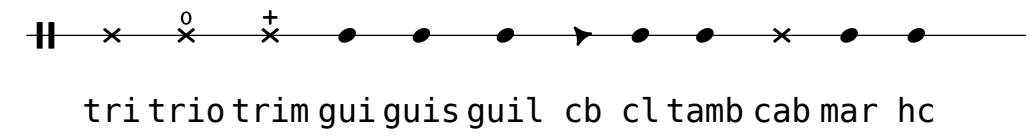
`congas-style`
: This typesets congas on a two line staff



`bongos-style`
: This typesets bongos on a two line staff

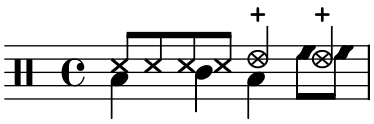


`percussion-style`
: To typeset all kinds of simple percussion on one line staves.



If you do not like any of the predefined lists you can define your own list at the top of your file

```
#(define mydrums '(
         (bassdrum     default   #f         -1)
         (snare        default   #f         0)
         (hihat        cross     #f         1)
         (pedalhihat   xcircle   "stopped"  2)
         (lowtom       diamond   #f         3)))
up = \drummode { hh8 hh hh hh hhp4 hhp }
down = \drummode { bd4 sn bd toml8 toml }

\new DrumStaff <<
  \set DrumStaff.drumStyleTable = #(alist->hash-table mydrums)
  \new DrumVoice { \voiceOne \up }
  \new DrumVoice { \voiceTwo \down }
>>
```



See also

Init files: ‘`ly/drumpitch-init.ly`’.

Program reference: `DrumStaff`, `DrumVoice`.

Bugs

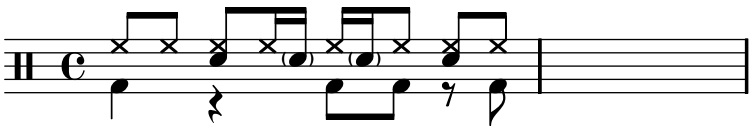
Because general MIDI does not contain rim shots, the sidestick is used for this purpose instead.

7.4.4 Ghost notes

Ghost notes for drums and percussion may be created using the `\parenthesize` command detailed in Section 8.5.8 [Parentheses], page 211. However, the default `\drummode` does not include the `Parenthesis_engraver` plugin which allows this. You must add the plugin explicitly in the context definition as detailed in Section 9.2.3 [Changing context properties on the fly], page 220.

```
\new DrumStaff \with {
  \consists "Parenthesis_engraver"
} <<
  \context DrumVoice  = "1"  { s1 *2 }
  \context DrumVoice  = "2" { s1 *2 }
  \drummode {
    <<
      {
        hh8[ hh] <hh sn> hh16
        < \parenthesize sn > hh < \parenthesize
        sn > hh8 <hh sn> hh
      } \\ {
        bd4 r4 bd8 bd r8 bd
      }
    >>
```

```
  }
>>
```

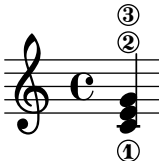


Also note that you must add chords (< > brackets) around each `\parenthesize` statement.

7.5 Guitar

7.5.1 String number indications

String numbers can be added to chords, by indicating the string number with *\number*,



See also '`input/regression/string-number.ly`'.

See also

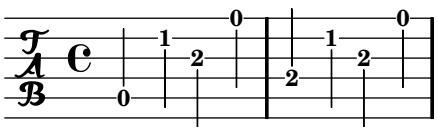
Program reference: `StringNumber`.

7.5.2 Tablatures basic

Tablature notation is used for notating music for plucked string instruments. Pitches are not denoted with note heads, but by numbers indicating on which string and fret a note must be played. LilyPond offers limited support for tablature.

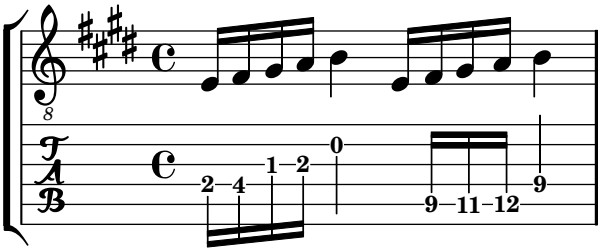
The string number associated to a note is given as a backslash followed by a number, e.g., `c4\3` for a C quarter on the third string. By default, string 1 is the highest one, and the tuning defaults to the standard guitar tuning (with 6 strings). The notes are printed as tablature, by using `TabStaff` and `TabVoice` contexts

```
\new TabStaff {
  a,4\5 c'\2 a\3 e'\1
  e\4 c'\2 a\3 e'\1
}
```



When no string is specified, the first string that does not give a fret number less than `minimumFret` is selected. The default value for `minimumFret` is 0

```
e16 fis gis a b4
\set TabStaff.minimumFret = #8
e16 fis gis a b4
```



Commonly tweaked properties

To print tablatures with stems down and horizontal beams, initialize the `TabStaff` with this code:

```
\stemDown
\override Beam #'damping = #100000
```

See also

Program reference: `TabStaff`, `TabVoice`.

Bugs

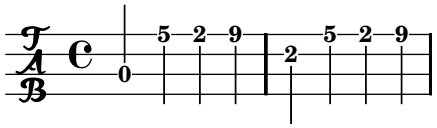
Chords are not handled in a special way, and hence the automatic string selector may easily select the same string to two notes in a chord.

7.5.3 Non-guitar tablatures

You can change the tuning of the strings. A string tuning is given as a Scheme list with one integer number for each string, the number being the pitch (measured in semitones relative to middle C) of an open string. The numbers specified for `stringTuning` are the numbers of semitones to subtract or add, starting the specified pitch by default middle C, in string order. LilyPond automatically calculates the number of strings by looking at `stringTuning`.

In the next example, `stringTunings` is set for the pitches e, a, d, and g

```
\new TabStaff <<
  \set TabStaff.stringTunings = #'(-5 -10 -15 -20)
  {
    a,4 c' a e' e c' a e'
  }
>>
```



LilyPond comes with predefined string tunings for banjo, mandolin, guitar and bass guitar.

```
\set TabStaff.stringTunings = #bass-tuning
```

The default string tuning is `guitar-tuning` (the standard EADGBE tuning). Some other predefined tunings are `guitar-open-g-tuning`, `mandolin-tuning` and `banjo-open-g-tuning`.

See also

The file '`scm/output-lib.scm`' contains the predefined string tunings. Program reference: `Tab_note_heads_engraver`.

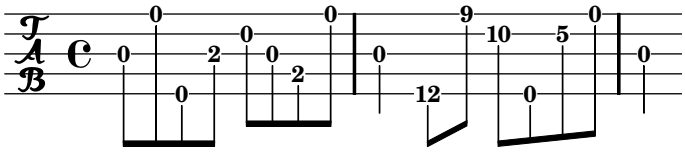
Bugs

No guitar special effects have been implemented.

7.5.4 Banjo tablatures

LilyPond has basic support for five stringed banjo. When making tablatures for five stringed banjo, use the banjo tablature format function to get correct fret numbers for the fifth string:

```
\new TabStaff <<
  \set TabStaff.tablatureFormat = #fret-number-tablature-format-banjo
  \set TabStaff.stringTunings = #banjo-open-g-tuning
  {
    \stemDown
    g8 d' g'\5 a b g e d' |
    g4 d''8\5 b' a'\2 g'\5 e'\2 d' |
    g4
  }
>>
```



A number of common tunings for banjo are predefined in LilyPond: `banjo-c-tuning` (gCGBD), `banjo-modal-tuning` (gDGCD), `banjo-open-d-tuning` (aDF#AD) and `banjo-open-dm-tuning` (aDFAD).

These tunings may be converted to four string banjo tunings using the `four-string-banjo` function:

```
\set TabStaff.stringTunings = #(four-string-banjo banjo-c-tuning)
```

See also

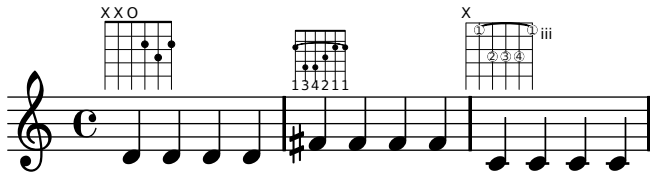
The file '`scm/output-lib.scm`' contains predefined banjo tunings.

7.5.5 Fret diagrams

Fret diagrams can be added to music as a markup to the desired note. The markup contains information about the desired fret diagram, as shown in the following example

```
\new Voice {
  d'^\markup \fret-diagram #"6-x;5-x;4-o;3-2;2-3;1-2;"
  d' d' d'
  fis'^\markup \override #'(size . 0.75) {
    \override #'(finger-code . below-string) {
      \fret-diagram-verbose #'((place-fret 6 2 1) (barre 6 1 2)
                               (place-fret 5 4 3) (place-fret 4 4 4)
                               (place-fret 3 3 2) (place-fret 2 2 1)
                               (place-fret 1 2 1))
    }
  }
  fis' fis' fis'
  c'^\markup \override #'(dot-radius . 0.35) {
    \override #'(finger-code . in-dot) {
      \override #'(dot-color . white) {
        \fret-diagram-terse #"x;3-1-(;5-2;5-3;5-4;3-1-);"
      }
    }
```

```
  }
  c' c' c'
}
```



There are three different fret-diagram markup interfaces: standard, terse, and verbose. The three interfaces produce equivalent markups, but have varying amounts of information in the markup string. Details about the markup interfaces are found at Section 8.1.6 [Overview of text markup commands], page 174.

You can set a number of graphical properties according to your preference. Details about the property interface to fret diagrams are found at `fret-diagram-interface`.

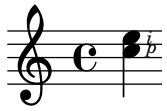
See also

Examples: ‘`input/test/fret-diagram.ly`’

7.5.6 Right hand fingerings

Right hand fingerings in chords can be entered using *note*`-\rightHandFinger` *finger*

```
<c-\rightHandFinger #1 e-\rightHandFinger #2 >
```



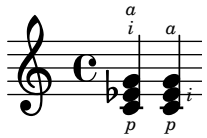
for brevity, you can abbreviate `\rightHandFinger` to something short, for example `RH`,

```
#(define RH rightHandFinger)
```

Commonly tweaked properties

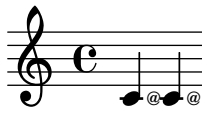
You may exercise greater control over right handing fingerings by setting `strokeFingerOrientations`,

```
#(define RH rightHandFinger)
{
  \set strokeFingerOrientations = #'(up down)
  <c-\RH #1 es-\RH #2 g-\RH #4 > 4
  \set strokeFingerOrientations = #'(up right down)
  <c-\RH #1 es-\RH #2 g-\RH #4 > 4
}
```



The letters used for the fingerings are contained in the property `digit-names`, but they can also be set individually by supplying `\rightHandFinger` with a string argument, as in the following example

```
#(define RH rightHandFinger)
{
  \set strokeFingerOrientations = #'(right)
  \override StrokeFinger #'digit-names = ##("x" "y" "z" "!" "@")
  <c-\RH #5 >4
  <c-\RH "@">4
}
```



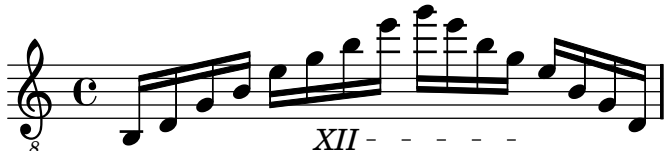
See also

Internalls: `StrokeFinger`

7.5.7 Other guitar issues

This example demonstrates how to include guitar position and barring indications.

```
\clef "G_8"
b16 d16 g16 b16 e16
\textSpannerDown
\override TextSpanner #'edge-text = #'("XII " . "")
  g16\startTextSpan
  b16 e16 g16 e16 b16 g16\stopTextSpan
e16 b16 g16 d16
```



Stopped (X) note heads are used in guitar music to signal a place where the guitarist must play a certain note or chord, with its fingers just touching the strings instead of fully pressing them. This gives the sound a percussive noise-like sound that still maintains part of the original pitch. It is notated with cross noteheads; this is demonstrated in Section 8.4.5 [Special noteheads], page 205.

7.6 Bagpipe

7.6.1 Bagpipe definitions

LilyPond contains special definitions for music for the Scottish highland bagpipe; to use them, add

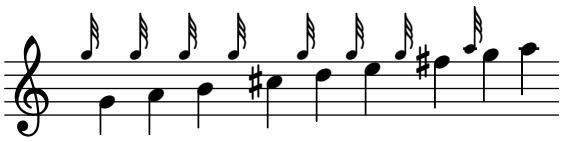
```
\include "bagpipe.ly"
```

at the top of your input file. This lets you add the special gracenotes common to bagpipe music with short commands. For example, you could write `\taor` instead of

```
\grace { \small G32[ d G e] }
```

`bagpipe.ly` also contains pitch definitions for the bagpipe notes in the appropiate octaves, so you do not need to worry about `\relative` or `\transpose`.

```
\include "bagpipe.ly"
{ \grg G4 \grg a \grg b \grg c \grg d \grg e \grg f \grA g A }
```



Bagpipe music nominally uses the key of D Major (even though that isn't really true). However, since that is the only key that can be used, the key signature is normally not written out. To set this up correctly, always start your music with `\hideKeySignature`. If you for some reason want to show the key signature, you can use `\showKeySignature` instead.

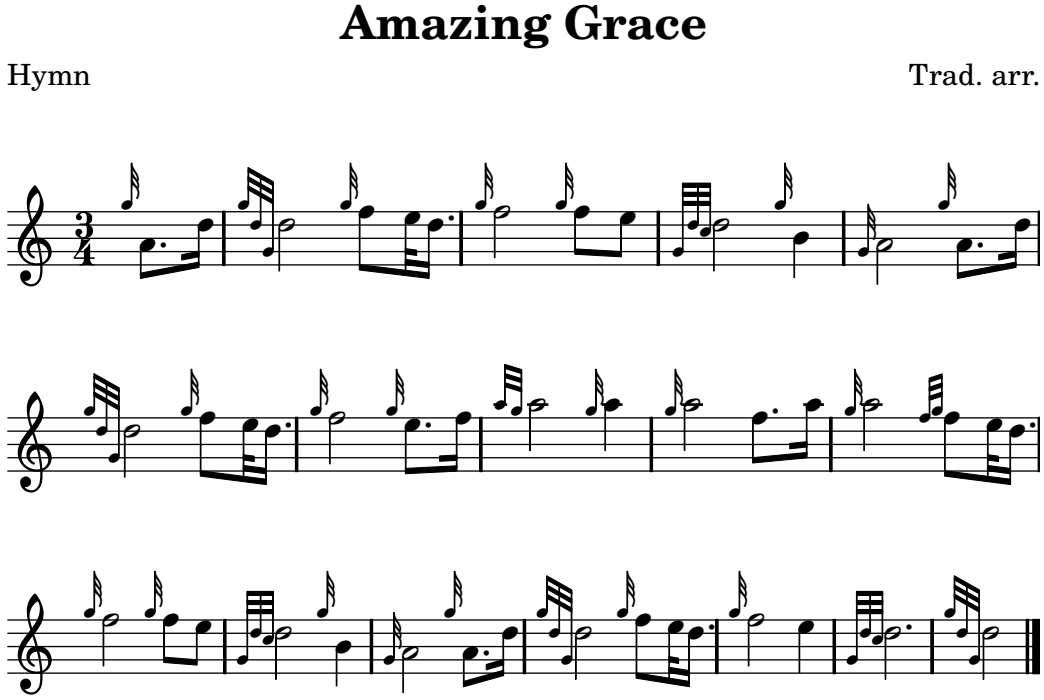
Some modern music use cross fingering on c and f to flatten those notes. This can be indicated by `cflat` or `fflat`. Similarly, the piobaireachd high g can be written `gflat` when it occurs in light music.

7.6.2 Bagpipe example

This is what the well known tune Amazing Grace looks like in bagpipe notation.

```
\include "bagpipe.ly"
\layout {
  indent = 0.0\cm
  \context { \Score \remove "Bar_number_engraver" }
}

\header {
  title = "Amazing Grace"
  meter = "Hymn"
  arranger = "Trad. arr."
}

{
  \hideKeySignature
  \time 3/4
  \grg \partial 4 a8. d16
  \slurd d2 \grg f8[ e32 d16.]
  \grg f2 \grg f8 e
  \thrwd d2 \grg b4
  \grG a2 \grg a8. d16
  \slurd d2 \grg f8[ e32 d16.]
  \grg f2 \grg e8. f16
  \dblA A2 \grg A4
  \grg A2 f8. A16
  \grg A2 \hdblf f8[ e32 d16.]
  \grg f2 \grg f8 e
  \thrwd d2 \grg b4
  \grG a2 \grg a8. d16
  \slurd d2 \grg f8[ e32 d16.]
```

```
  \grg f2 e4
  \thrwd d2.
  \slurd d2
  \bar "|."
}
```



7.7 Ancient notation

Support for ancient notation includes features for mensural notation and Gregorian Chant notation. There is also limited support for figured bass notation.

Many graphical objects provide a `style` property, see

- Section 7.7.1 [Ancient note heads], page 148,
- Section 7.7.2 [Ancient accidentals], page 148,
- Section 7.7.3 [Ancient rests], page 149,
- Section 7.7.4 [Ancient clefs], page 149,
- Section 7.7.5 [Ancient flags], page 152,
- Section 7.7.6 [Ancient time signatures], page 152.

By manipulating such a grob property, the typographical appearance of the affected graphical objects can be accommodated for a specific notation flavor without the need for introducing any new notational concept.

In addition to the standard articulation signs described in section Section 6.6.1 [Articulations], page 94, specific articulation signs for ancient notation are provided.

- Section 7.7.7 [Ancient articulations], page 153

Other aspects of ancient notation can not that easily be expressed in terms of just changing a style property of a graphical object or adding articulation signs. Some notational concepts are introduced specifically for ancient notation,

- Section 7.7.8 [Custodes], page 154,
- Section 7.7.9 [Divisiones], page 155,

- Section 7.7.10 [Ligatures], page 155.

If this all is too much of documentation for you, and you just want to dive into typesetting without worrying too much about the details on how to customize a context, you may have a look at the predefined contexts. Use them to set up predefined style-specific voice and staff contexts, and directly go ahead with the note entry,

- Section 7.7.11 [Gregorian Chant contexts], page 162,
- Section 7.7.12 [Mensural contexts], page 162.

There is limited support for figured bass notation which came up during the baroque period.

- Section 7.7.14 [Figured bass], page 164

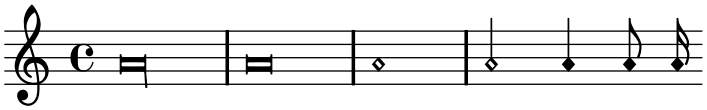
Here are all suptopics at a glance:

7.7.1 Ancient note heads

For ancient notation, a note head style other than the `default` style may be chosen. This is accomplished by setting the `style` property of the `NoteHead` object to `baroque`, `neomensural`, `mensural` or `petrucci`. The `baroque` style differs from the `default` style only in using a square shape for `\breve` note heads. The `neomensural` style differs from the `baroque` style in that it uses rhomboidal heads for whole notes and all smaller durations. Stems are centered on the note heads. This style is particularly useful when transcribing mensural music, e.g., for the incipit. The `mensural` style produces note heads that mimic the look of note heads in historic printings of the 16th century. Finally, the `petrucci` style also mimicks historic printings, but uses bigger note heads.

The following example demonstrates the `neomensural` style

```
\set Score.skipBars = ##t
\override NoteHead #'style = #'neomensural
a'\longa a'\breve a'1 a'2 a'4 a'8 a'16
```



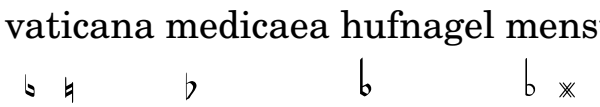
When typesetting a piece in Gregorian Chant notation, the `Gregorian_ligature_engraver` will automatically select the proper note heads, so there is no need to explicitly set the note head style. Still, the note head style can be set, e.g., to `vaticana_punctum` to produce punctum neumes. Similarly, a `Mensural_ligature_engraver` is used to automatically assemble mensural ligatures. See Section 7.7.10 [Ligatures], page 155 for how ligature engravers work.

See also

Examples: '`input/regression/note-head-style.ly`' gives an overview over all available note head styles.

7.7.2 Ancient accidentals

Use the `style` property of grob `Accidental` to select ancient accidentals. Supported styles are `mensural`, `vaticana`, `hufnagel`, and `medicaea`.



As shown, not all accidentals are supported by each style. When trying to access an unsupported accidental, LilyPond will switch to a different style, as demonstrated in '`input/test/ancient-accidentals.ly`'.

Similarly to local accidentals, the style of the key signature can be controlled by the `style` property of the `KeySignature` grob.

See also

In this manual: Section 6.1 [Pitches], page 59, Section 6.1.3 [Cautionary accidentals], page 61 and Section 9.1.1 [Automatic accidentals], page 213 give a general introduction of the use of accidentals. Section 6.4.2 [Key signature], page 77 gives a general introduction of the use of key signatures.

Program reference: `KeySignature`.

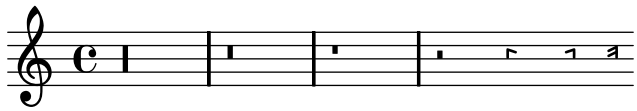
Examples: '`input/test/ancient-accidentals.ly`'.

7.7.3 Ancient rests

Use the `style` property of grob `Rest` to select ancient rests. Supported styles are `classical`, `neomensural`, and `mensural`. `classical` differs from the `default` style only in that the quarter rest looks like a horizontally mirrored 8th rest. The `neomensural` style suits well for, e.g., the incipit of a transcribed mensural piece of music. The `mensural` style finally mimics the appearance of rests as in historic prints of the 16th century.

The following example demonstrates the `neomensural` style

```
\set Score.skipBars = ##t
\override Rest #'style = #'neomensural
r\longa r\breve r1 r2 r4 r8 r16
```



There are no 32th and 64th rests specifically for the mensural or neo-mensural style. Instead, the rests from the default style will be taken. See '`input/test/rests.ly`' for a chart of all rests.

There are no rests in Gregorian Chant notation; instead, it uses Section 7.7.9 [Divisiones], page 155.

See also

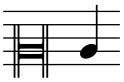
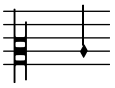
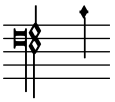
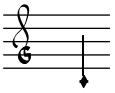
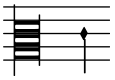
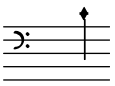
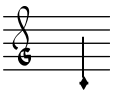
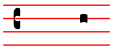
In this manual: Section 6.1.9 [Rests], page 64 gives a general introduction into the use of rests.

7.7.4 Ancient clefs

LilyPond supports a variety of clefs, many of them ancient.

The following table shows all ancient clefs that are supported via the `\clef` command. Some of the clefs use the same glyph, but differ only with respect to the line they are printed on. In such cases, a trailing number in the name is used to enumerate these clefs. Still, you can manually force a clef glyph to be typeset on an arbitrary line, as described in Section 6.4.1 [Clef], page 76. The note printed to the right side of each clef in the example column denotes the `c'` with respect to that clef.

Description	Supported Clefs	Example

modern style mensural C clef	`neomensural-c1`, `neomensural-c2`, `neomensural-c3`, `neomensural-c4`	
petrucci style mensural C clefs, for use on different staff lines (the examples show the 2nd staff line C clef)	`petrucci-c1`, `petrucci-c2`, `petrucci-c3`, `petrucci-c4`, `petrucci-c5`	
petrucci style mensural F clef	`petrucci-f`	
petrucci style mensural G clef	`petrucci-g`	
historic style mensural C clef	`mensural-c1`, `mensural-c2`, `mensural-c3`, `mensural-c4`	
historic style mensural F clef	`mensural-f`	
historic style mensural G clef	`mensural-g`	
Editio Vaticana style do clef	`vaticana-do1`, `vaticana-do2`, `vaticana-do3`	

Editio Vaticana style fa clef	`vaticana-fa1`, `vaticana-fa2`
Editio Medicaea style do clef	`medicaea-do1`, `medicaea-do2`, `medicaea-do3`
Editio Medicaea style fa clef	`medicaea-fa1`, `medicaea-fa2`
historic style hufnagel do clef	`hufnagel-do1`, `hufnagel-do2`, `hufnagel-do3`
historic style hufnagel fa clef	`hufnagel-fa1`, `hufnagel-fa2`
historic style hufnagel combined do/fa clef	`hufnagel-do-fa`

Modern style means "as is typeset in contemporary editions of transcribed mensural music".

Petrucci style means "inspired by printings published by the famous engraver Petrucci (1466-1539)".

Historic style means "as was typeset or written in historic editions (other than those of Petrucci)".

Editio XXX style means "as is/was printed in Editio XXX".

Petrucci used C clefs with differently balanced left-side vertical beams, depending on which staff line it is printed.

See also

In this manual: see Section 6.4.1 [Clef], page 76.

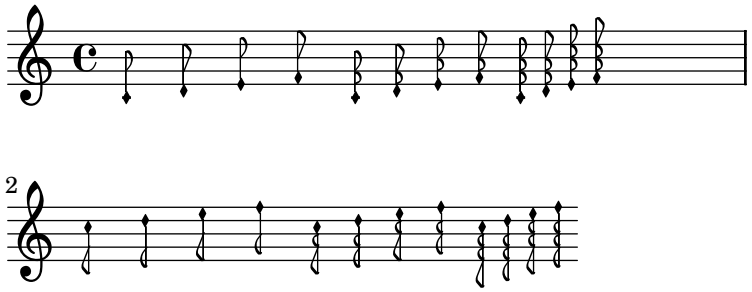
Bugs

The mensural g clef is mapped to the Petrucci g clef.

7.7.5 Ancient flags

Use the `flag-style` property of grob `Stem` to select ancient flags. Besides the `default` flag style, only the `mensural` style is supported

```
\override Stem #'flag-style = #'mensural
\override Stem #'thickness = #1.0
\override NoteHead #'style = #'mensural
\autoBeamOff
c'8 d'8 e'8 f'8 c'16 d'16 e'16 f'16 c'32 d'32 e'32 f'32 s8
c''8 d''8 e''8 f''8 c''16 d''16 e''16 f''16 c''32 d''32 e''32 f''32
```



Note that the innermost flare of each mensural flag always is vertically aligned with a staff line.

There is no particular flag style for neo-mensural notation. Hence, when typesetting the incipit of a transcribed piece of mensural music, the default flag style should be used. There are no flags in Gregorian Chant notation.

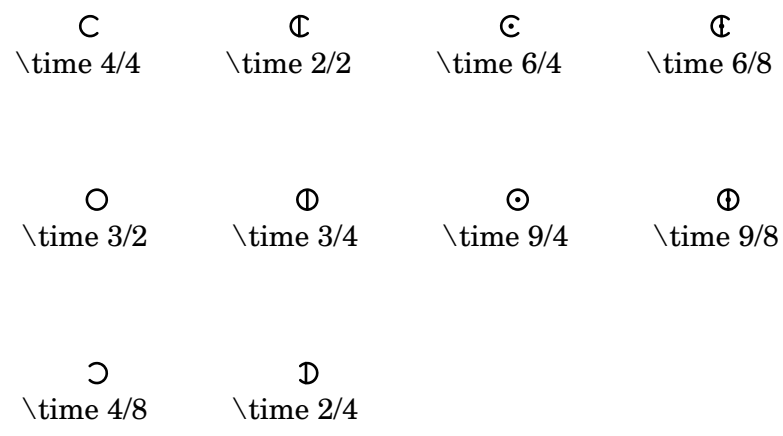
Bugs

The attachment of ancient flags to stems is slightly off due to a change in early 2.3.x.

Vertically aligning each flag with a staff line assumes that stems always end either exactly on or exactly in the middle between two staff lines. This may not always be true when using advanced layout features of classical notation (which however are typically out of scope for mensural notation).

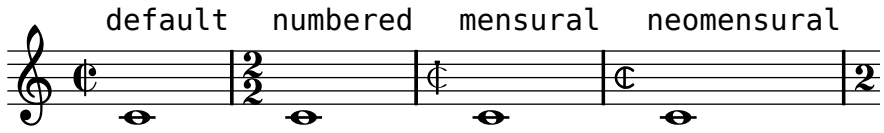
7.7.6 Ancient time signatures

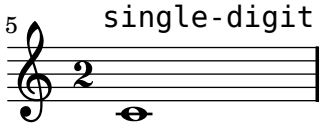
There is limited support for mensural time signatures. The glyphs are hard-wired to particular time fractions. In other words, to get a particular mensural signature glyph with the `\time n/m` command, `n` and `m` have to be chosen according to the following table



Use the `style` property of grob `TimeSignature` to select ancient time signatures. Supported styles are `neomensural` and `mensural`. The above table uses the `neomensural` style. This style is appropriate for the incipit of transcriptions of mensural pieces. The `mensural` style mimics the look of historical printings of the 16th century.

The following examples show the differences in style,





See also

This manual: Section 6.4.3 [Time signature], page 78 gives a general introduction to the use of time signatures.

Bugs

Ratios of note durations do not change with the time signature. For example, the ratio of 1 brevis = 3 semibrevis (tempus perfectum) must be made by hand, by setting

```
breveTP = #(ly:make-duration -1 0 3 2)
...
{ c\breveTP f1 }
```

This sets `breveTP` to 3/2 times 2 = 3 times a whole note.

The `old6/8alt` symbol (an alternate symbol for 6/8) is not addressable with `\time`. Use a `\markup` instead

7.7.7 Ancient articulations

In addition to the standard articulation signs described in section Section 6.6.1 [Articulations], page 94, articulation signs for ancient notation are provided. These are specifically designed for use with notation in Editio Vaticana style.

```
\include "gregorian-init.ly"
\score {
  \new VaticanaVoice {
    \override TextScript #'font-family = #'typewriter
    \override TextScript #'font-shape = #'upright
    \override Script #'padding = #-0.1
    a\ictus_"ictus" \break
    a\circulus_"circulus" \break
    a\semicirculus_"semicirculus" \break
    a\accentus_"accentus" \break
    \[ a_"episem" \episemInitium \pes b \flexa a b \episemFinis \flexa a \]
  }
}
```

Bugs

Some articulations are vertically placed too closely to the correpsonding note heads.

The episem line is not displayed in many cases. If it is displayed, the right end of the episem line is often too far to the right.

7.7.8 Custodes

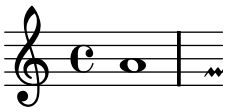
A *custos* (plural: *custodes*; Latin word for 'guard') is a symbol that appears at the end of a staff. It anticipates the pitch of the first note(s) of the following line thus helping the performer to manage line breaks during performance.

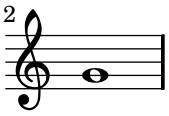
Custodes were frequently used in music notation until the 17th century. Nowadays, they have survived only in a few particular forms of musical notation such as contemporary editions of Gregorian chant like the *editio vaticana*. There are different custos glyphs used in different flavors of notational style.

For typesetting custodes, just put a `Custos_engraver` into the `Staff` context when declaring the `\layout` block, as shown in the following example

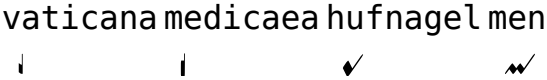
```
\layout {
  \context {
    \Staff
    \consists Custos_engraver
    Custos \override #'style = #'mensural
  }
}
```

The result looks like this





The custos glyph is selected by the `style` property. The styles supported are `vaticana`, `medicaea`, `hufnagel`, and `mensural`. They are demonstrated in the following fragment



See also

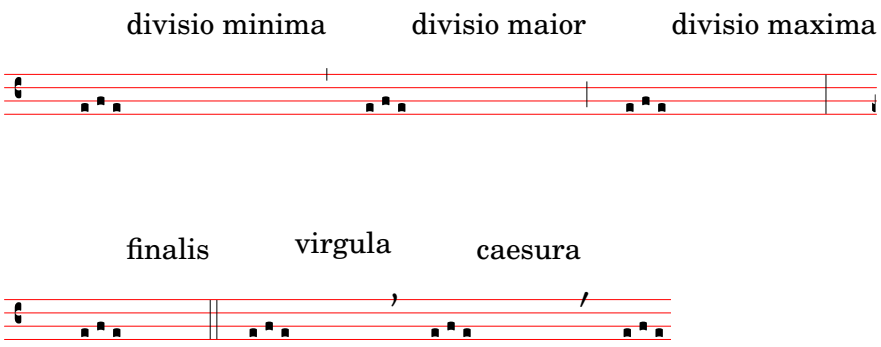
Program reference: `Custos`.

Examples: '`input/regression/custos.ly`'.

7.7.9 Divisiones

A *divisio* (plural: *divisiones*; Latin word for 'division') is a staff context symbol that is used to structure Gregorian music into phrases and sections. The musical meaning of *divisio minima*, *divisio maior*, and *divisio maxima* can be characterized as short, medium, and long pause, somewhat like the breathmarks from Section 6.6.4 [Breath marks], page 100. The *finalis* sign not only marks the end of a chant, but is also frequently used within a single antiphonal/responsorial chant to mark the end of each section.

To use divisiones, include the file '`gregorian-init.ly`'. It contains definitions that you can apply by just inserting `\divisioMinima`, `\divisioMaior`, `\divisioMaxima`, and `\finalis` at proper places in the input. Some editions use *virgula* or *caesura* instead of divisio minima. Therefore, '`gregorian-init.ly`' also defines `\virgula` and `\caesura`



Predefined commands

`\virgula`, `\caesura`, `\divisioMinima`, `\divisioMaior`, `\divisioMaxima`, `\finalis`.

See also

In this manual: Section 6.6.4 [Breath marks], page 100.

Program reference: `BreathingSign`.

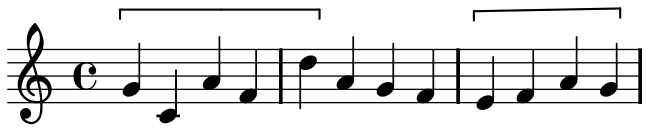
Examples: '`input/test/divisiones.ly`'.

7.7.10 Ligatures

A ligature is a graphical symbol that represents at least two distinct notes. Ligatures originally appeared in the manuscripts of Gregorian chant notation to denote ascending or descending sequences of notes.

Ligatures are entered by enclosing them in `\[` and `\]`. Some ligature styles may need additional input syntax specific for this particular type of ligature. By default, the `LigatureBracket` engraver just puts a square bracket above the ligature

```
\transpose c c' {
  \[ g c a f d' \]
  a g f
  \[ e f a g \]
}
```



To select a specific style of ligatures, a proper ligature engraver has to be added to the `Voice` context, as explained in the following subsections. Only white mensural ligatures are supported with certain limitations.

Bugs

Ligatures need special spacing that has not yet been implemented. As a result, there is too much space between ligatures most of the time, and line breaking often is unsatisfactory. Also, lyrics do not correctly align with ligatures.

Accidentals must not be printed within a ligature, but instead need to be collected and printed in front of it.

The syntax still uses the deprecated infix style `\[ music expr \]`. For consistency reasons, it will eventually be changed to postfix style `note\[ ... note\]`. Alternatively, the file '`gregorian-init.ly`' can be included; it provides a scheme function

```
\ligature music expr
```

with the same effect and is believed to be stable.

7.7.10.1 White mensural ligatures

There is limited support for white mensural ligatures.

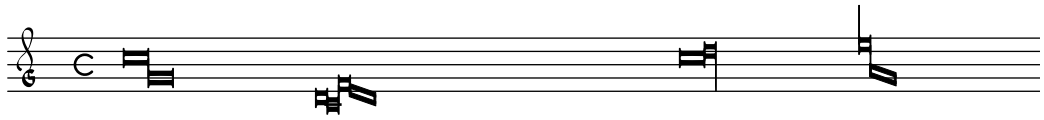
To engrave white mensural ligatures, in the layout block put the `Mensural_ligature_engraver` into the `Voice` context, and remove the `Ligature_bracket_engraver`, like this

```
\layout {
  \context {
    \Voice
    \remove Ligature_bracket_engraver
    \consists Mensural_ligature_engraver
  }
}
```

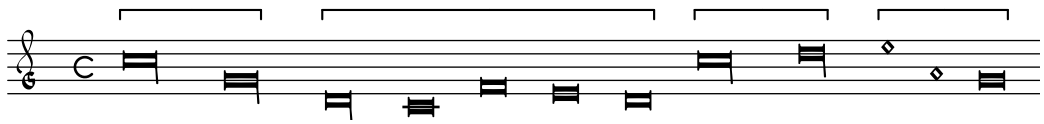
There is no additional input language to describe the shape of a white mensural ligature. The shape is rather determined solely from the pitch and duration of the enclosed notes. While this approach may take a new user a while to get accustomed to, it has the great advantage that the full musical information of the ligature is known internally. This is not only required for correct MIDI output, but also allows for automatic transcription of the ligatures.

For example,

```
\set Score.timing = ##f
\set Score.defaultBarType = "empty"
\override NoteHead #'style = #'neomensural
\override Staff.TimeSignature #'style = #'neomensural
\clef "petrucci-g"
\[ c'\maxima g \]
\[ d\longa c\breve f e d \]
\[ c'\maxima d'\longa \]
\[ e'1 a g\breve \]
```



Without replacing `Ligature_bracket_engraver` with `Mensural_ligature_engraver`, the same music transcribes to the following



Bugs

Horizontal spacing is poor.

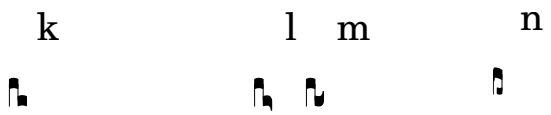
7.7.10.2 Gregorian square neumes ligatures

There is limited support for Gregorian square neumes notation (following the style of the Editio Vaticana). Core ligatures can already be typeset, but essential issues for serious typesetting are still lacking, such as (among others) horizontal alignment of multiple ligatures, lyrics alignment and proper handling of accidentals.

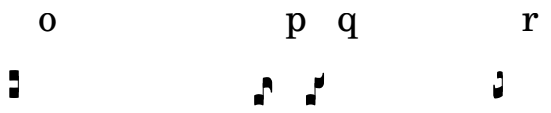
The following table contains the extended neumes table of the 2nd volume of the Antiphonale Romanum (*Liber Hymnarius*), published 1983 by the monks of Solesmes.

Neuma aut Neumarum Elementa	**Figurae Rectae**	**Figurae Liquescentes Auctae**	**Figurae Liquescentes Deminutae**
`1. Punctum`	a b	c d e	f
`2. Virga`	g		
`3. Apostropha vel Stropha`	h	i	
`4. Oriscus`	j		

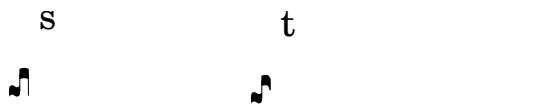
5. Clivis vel Flexa



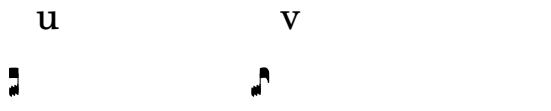
6. Podatus vel Pes



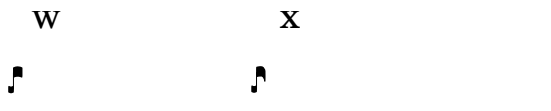
7. Pes Quassus



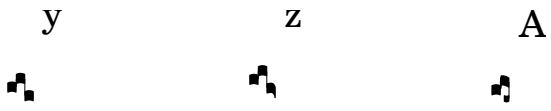
8. Quilisma Pes



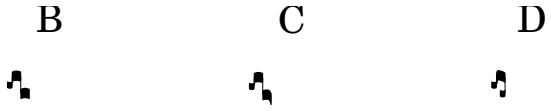
9. Podatus Initio Debilis



10. Torculus



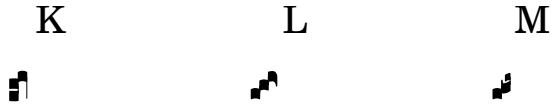
11. Torculus Initio Debilis



12. Porrectus

13. Climacus

14. Scandicus



15. Salicus

16. Trigonus

Unlike most other neumes notation systems, the input language for neumes does not reflect the typographical appearance, but is designed to focus on musical meaning. For example, `\[ a \pes b \flexa g \]` produces a Torculus consisting of three Punctum heads, while `\[ a \flexa g \pes b \]` produces a Porrectus with a curved flexa shape and only a single Punctum head. There is no command to explicitly typeset the curved flexa shape; the decision of when to typeset a curved flexa shape is based on the musical input. The idea of this approach is to separate the musical aspects of the input from the notation style of the output. This way, the same input can be reused to typeset the same music in a different style of Gregorian chant notation.

The following table shows the code fragments that produce the ligatures in the above neumes table. The letter in the first column in each line of the below table indicates to which ligature in the above table it refers. The second column gives the name of the ligature. The third column shows the code fragment that produces this ligature, using `g`, `a`, and `b` as example pitches.

#	**Name**	**Input Language**
a	Punctum	`\[ b \]`
b	Punctum Inclinatum	`\[ \inclinatum b \]`
c	Punctum Auctum Ascendens	`\[ \auctum \ascendens b \]`
d	Punctum Auctum Descendens	`\[ \auctum \descendens b \]`
e	Punctum Inclinatum Auctum	`\[ \inclinatum \auctum b \]`
f	Punctum Inclinatum Parvum	`\[ \inclinatum \deminutum b \]`

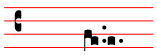
g	Virga	`\[ \virga b \]`
h	Stropha	`\[ \stropha b \]`
i	Stropha Aucta	`\[ \stropha \auctum b \]`
j	Oriscus	`\[ \oriscus b \]`
k	Clivis vel Flexa	`\[ b \flexa g \]`
l	Clivis Aucta Descendens	`\[ b \flexa \auctum \descendens g \]`
m	Clivis Aucta Ascendens	`\[ b \flexa \auctum \ascendens g \]`
n	Cephalicus	`\[ b \flexa \deminutum g \]`
o	Podatus vel Pes	`\[ g \pes b \]`
p	Pes Auctus Descendens	`\[ g \pes \auctum \descendens b \]`
q	Pes Auctus Ascendens	`\[ g \pes \auctum \ascendens b \]`
r	Epiphonus	`\[ g \pes \deminutum b \]`
s	Pes Quassus	`\[ \oriscus g \pes \virga b \]`
t	Pes Quassus Auctus Descendens	`\[ \oriscus g \pes \auctum \descendens b \]`
u	Quilisma Pes	`\[ \quilisma g \pes b \]`
v	Quilisma Pes Auctus Descendens	`\[ \quilisma g \pes \auctum \descendens b \]`
w	Pes Initio Debilis	`\[ \deminutum g \pes b \]`
x	Pes Auctus Descendens Initio Debilis	`\[ \deminutum g \pes \auctum \descendens b \]`
y	Torculus	`\[ a \pes b \flexa g \]`
z	Torculus Auctus Descendens	`\[ a \pes b \flexa \auctum \descendens g \]`
A	Torculus Deminutus	`\[ a \pes b \flexa \deminutum g \]`
B	Torculus Initio Debilis	`\[ \deminutum a \pes b \flexa g \]`
C	Torculus Auctus Descendens Initio Debilis	`\[ \deminutum a \pes b \flexa \auctum \descendens g \]`
D	Torculus Deminutus Initio Debilis	`\[ \deminutum a \pes b \flexa \deminutum g \]`
E	Porrectus	`\[ a \flexa g \pes b \]`

F	Porrectus Auctus Descendens	`\[ a \flexa g \pes \auctum \descendens b \]`
G	Porrectus Deminutus	`\[ a \flexa g \pes \deminutum b \]`
H	Climacus	`\[ \virga b \inclinatum a \inclinatum g \]`
I	Climacus Auctus	`\[ \virga b \inclinatum a \inclinatum \auctum g \]`
J	Climacus Deminutus	`\[ \virga b \inclinatum a \inclinatum \deminutum g \]`
K	Scandicus	`\[ g \pes a \virga b \]`
L	Scandicus Auctus Descendens	`\[ g \pes a \pes \auctum \descendens b \]`
M	Scandicus Deminutus	`\[ g \pes a \pes \deminutum b \]`
N	Salicus	`\[ g \oriscus a \pes \virga b \]`
O	Salicus Auctus Descendens	`\[ g \oriscus a \pes \auctum \descendens b \]`
P	Trigonus	`\[ \stropha b \stropha b \stropha a \]`

The ligatures listed above mainly serve as a limited, but still representative pool of Gregorian ligature examples. Virtually, within the ligature delimiters `\[` and `\]`, any number of heads may be accumulated to form a single ligature, and head prefixes like `\pes`, `\flexa`, `\virga`, `\inclinatum`, etc. may be mixed in as desired. The use of the set of rules that underlies the construction of the ligatures in the above table is accordingly extrapolated. This way, infinitely many different ligatures can be created.

Augmentum dots, also called *morae*, are added with the music function `\augmentum`. Note that `\augmentum` is implemented as a unary music function rather than as head prefix. It applies to the immediately following music expression only. That is, `\augmentum \virga c` will have no visible effect. Instead, say `\virga \augmentum c` or `\augmentum {\virga c}`. Also note that you can say `\augmentum {a g}` as a shortcut for `\augmentum a \augmentum g`.

```
\include "gregorian-init.ly"
\score {
  \new VaticanaVoice {
    \[ \augmentum a \flexa \augmentum g \]
    \augmentum g
  }
}
```



Predefined commands

The following head prefixes are supported

`\virga`, `\stropha`, `\inclinatum`, `\auctum`, `\descendens`, `\ascendens`, `\oriscus`, `\quilisma`, `\deminutum`, `\cavum`, `\linea`.

Head prefixes can be accumulated, though restrictions apply. For example, either `\descendens` or `\ascendens` can be applied to a head, but not both to the same head.

Two adjacent heads can be tied together with the `\pes` and `\flexa` infix commands for a rising and falling line of melody, respectively.

Use the unary music function `\augmentum` to add augmentum dots.

Bugs

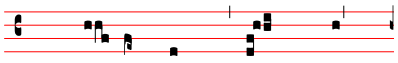
When an `\augmentum` dot appears at the end of the last staff within a ligature, it is sometimes vertically placed wrong. As a workaround, add an additional skip note (e.g. `s8`) as last note of the staff.

`\augmentum` should be implemented as a head prefix rather than a unary music function, such that `\augmentum` can be intermixed with head prefixes in arbitrary order.

7.7.11 Gregorian Chant contexts

The predefined `VaticanaVoiceContext` and `VaticanaStaffContext` can be used to engrave a piece of Gregorian Chant in the style of the Editio Vaticana. These contexts initialize all relevant context properties and grob properties to proper values, so you can immediately go ahead entering the chant, as the following excerpt demonstrates

```
\include "gregorian-init.ly"
\score {
  <<
    \new VaticanaVoice = "cantus" {
      \[ c'\melisma c' \flexa a \]
      \[ a \flexa \deminutum g\melismaEnd \]
      f \divisioMinima
      \[ f\melisma \pes a c' c' \pes d'\melismaEnd \]
      c' \divisioMinima \break
      \[ c'\melisma c' \flexa a \]
      \[ a \flexa \deminutum g\melismaEnd \] f \divisioMinima
    }
    \new Lyrics \lyricsto "cantus" {
      San- ctus, San- ctus, San- ctus
    }
  >>
}
```



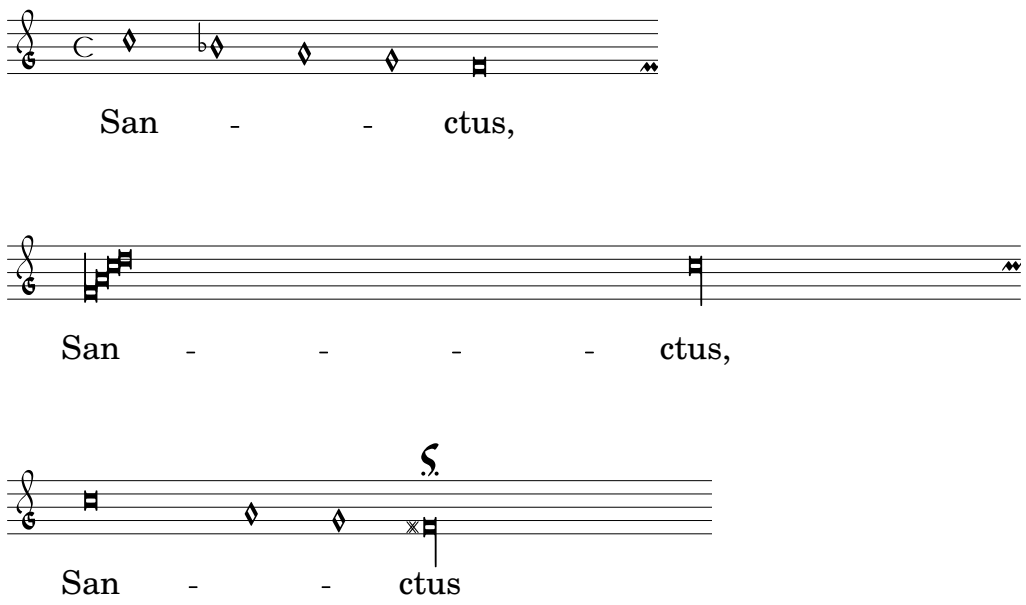
San- ctus, San- ctus,

San- ctus

7.7.12 Mensural contexts

The predefined `MensuralVoiceContext` and `MensuralStaffContext` can be used to engrave a piece in mensural style. These contexts initialize all relevant context properties and grob properties to proper values, so you can immediately go ahead entering the chant, as the following excerpt demonstrates

```
\score {
  <<
    \new MensuralVoice = "discantus" \transpose c c' {
      \override Score.BarNumber #'transparent = ##t {
        c'1\melisma bes a g\melismaEnd
        f\breve
        \[ f1\melisma a c'\breve d'\melismaEnd \]
        c'\longa
        c'\breve\melisma a1 g1\melismaEnd
        fis\longa^\signumcongruentiae
      }
    }
    \new Lyrics \lyricsto "discantus" {
      San -- ctus, San -- ctus, San -- ctus
    }
  >>
}
```

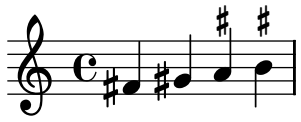


7.7.13 Musica ficta accidentals

In European music from before about 1600, singers were often expected to chromatically alter notes at their own initiative. This is called "Musica Ficta". In modern transcriptions, these accidentals are usually printed over the note.

Support for such suggested accidentals is included, and can be switched on by setting `suggestAccidentals` to true.

```
fis gis
\set suggestAccidentals = ##t
ais bis
```



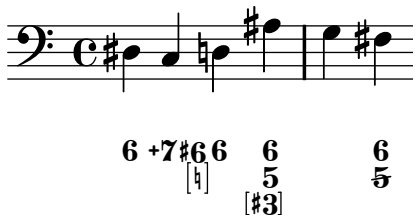
See also

Program reference: `Accidental_engraver` engraver and the `AccidentalSuggestion` object.

7.7.14 Figured bass

LilyPond has support for figured bass

```
<<
  \new Voice { \clef bass dis4 c d ais g fis}
  \new FiguredBass \figuremode {
    < 6 >4 < 7\+ >8 < 6+ [_!] >
    < 6 >4 <6 5 [3+] >
    < _ >4 < 6 5/>4
  }
>>
```



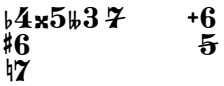
The support for figured bass consists of two parts: there is an input mode, introduced by `\figuremode`, where you can enter bass figures as numbers, and there is a context called `FiguredBass` that takes care of making `BassFigure` objects.

In figures input mode, a group of bass figures is delimited by `<` and `>`. The duration is entered after the `>`

```
<4 6>
```

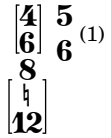
Accidentals are added when you append `-`, `!`, and `+` to the numbers. A plus sign is added when you append `\+`, and diminished fifths and sevenths can be obtained with `5/` and `7/`.

```
<4- 6+ 7!> <5++> <3--> <7/> r <6\+ 5/>
```



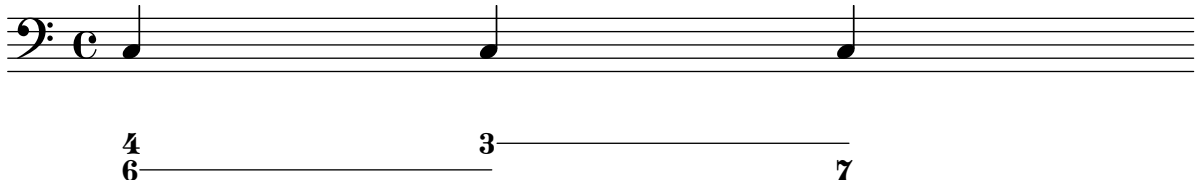
Spaces may be inserted by using `_`. Brackets are introduced with `[` and `]`. You can also include text strings and text markups, see Section 8.1.6 [Overview of text markup commands], page 174.

```
< [4 6] 8 [_! 12] > < 5 \markup { \number 6 \super (1) } >
```



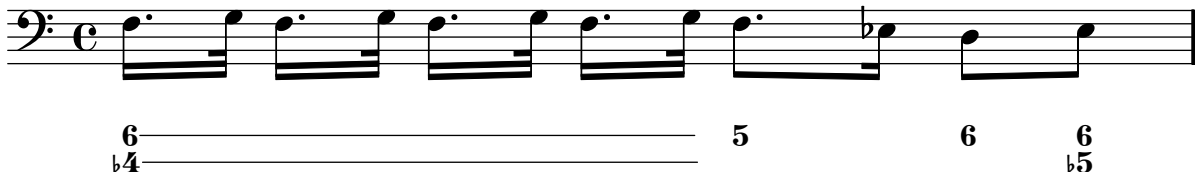
It is also possible to use continuation lines for repeated figures,

```
<<
  \new Staff {
    \clef bass
    c4 c c
  }
  \figures {
    \set useBassFigureExtenders = ##t
    <4 6> <3 6> <3 7>
  }
>>
```



In this case, the extender lines always replace existing figures.

The `FiguredBass` context doesn't pay attention to the actual bass line. As a consequence, you may have to insert extra figures to get extender lines below all notes, and you may have to add `\!` to avoid getting an extender line, e.g.



When using continuation lines, common figures are always put in the same vertical position. When this is unwanted, you can insert a rest with `r`. The rest will clear any previous alignment. For example, you can write

```
<4 6>8 r8
```

instead of

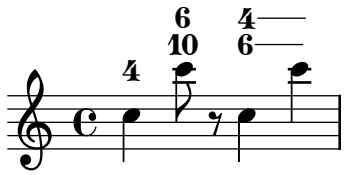
```
<4 6>4
```

Accidentals and plus signs can appear before or after the numbers, depending on the `figuredBassAlterationDirection` and `figuredBassPlusDirection` properties

Although the support for figured bass may superficially resemble chord support, it is much simpler. The `\figuremode` mode simply stores the numbers and `FiguredBass` context prints them as entered. There is no conversion to pitches and no realizations of the bass are played in the MIDI file.

Internally, the code produces markup texts. You can use any of the markup text properties to override formatting. For example, the vertical spacing of the figures may be set with `baseline-skip`.

Figured bass can also be added to `Staff` contexts directly. In this case, their vertical position is adjusted automatically.



Bugs

When using figured bass above the staff with extender lines and `implicitBassFigures` the lines may become swapped around. Maintaining order consistently will be impossible when multiple figures have overlapping extender lines. To avoid this problem, plese use `stacking-dir` on `BassFigureAlignment`.

See also

Program reference: `NewBassFigure`, `BassFigureAlignment`, `BassFigureLine`, `BassFigureBracket`, and `BassFigureContinuation` objects and `FiguredBass` context.

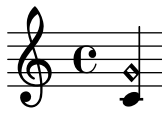
7.8 Other instrument specific notation

This section includes extra information for writing for instruments.

7.8.1 Artificial harmonics (strings)

Artificial harmonics are notated with a different notehead style. They are entered by marking the harmonic pitch with `\harmonic`.

```
<c g'\harmonic>4
```



8 Advanced notation

This chapter deals with rarely-used and advanced notation.

8.1 Text

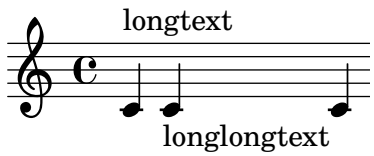
This section explains how to include text (with various formatting) in your scores.

To write accented and special text (such as characters from other languages), simply insert the characters directly into the lilypond file. The file must be saved as UTF-8. For more information, see Section 10.1.7 [Text encoding], page 238.

8.1.1 Text scripts

It is possible to place arbitrary strings of text or Section 8.1.4 [Text markup], page 170 above or below notes by using a string `c^"text"`. By default, these indications do not influence the note spacing, but by using the command `\fatText`, the widths will be taken into account

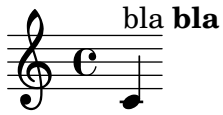
```
c4^"longtext" \fatText c4_"longlongtext" c4
```



To prevent text from influencing spacing, use `\emptyText`.

More complex formatting may also be added to a note by using the markup command,

```
c'4^\markup { bla \bold bla }
```



The `\markup` is described in more detail in Section 8.1.4 [Text markup], page 170.

Predefined commands

`\fatText`, `\emptyText`.

Commonly tweaked properties

Checking to make sure that text scripts and lyrics are within the margins is a relatively large computational task. To speed up processing, lilypond does not perform such calculations by default; to enable it, use

```
\override Score.PaperColumn #'keep-inside-line = ##t
```

See also

In this manual: Section 8.1.4 [Text markup], page 170.

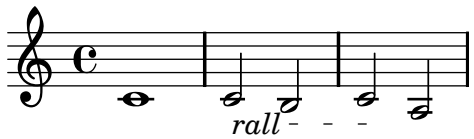
Program reference: `TextScript`.

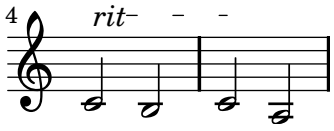
8.1.2 Text spanners

Some performance indications, e.g., *rallentando* or *accelerando*, are written as text and are extended over many measures with dotted lines. Such texts are created using text spanners; attach `\startTextSpan` and `\stopTextSpan` to the first and last notes of the spanner.

The string to be printed, as well as the style, is set through object properties

```
c1
\textSpannerDown
\override TextSpanner #'edge-text = #'("rall " . "")
c2\startTextSpan b c\stopTextSpan a
\break
\textSpannerUp
\override TextSpanner #'edge-text = #(cons (markup #:italic "rit" ) "")
c2\startTextSpan b c\stopTextSpan a
```





Predefined commands

`\textSpannerUp`, `\textSpannerDown`, `\textSpannerNeutral`.

Commonly tweaked properties

To print a solid line, use

```
\override TextSpanner #'dash-fraction = #'()
```

See also

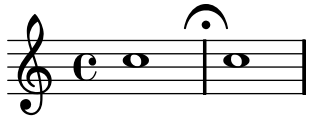
Program reference: `TextSpanner`.

Examples: ‘`input/regression/text-spanner.ly`’.

8.1.3 Text marks

The `\mark` command is primarily used for Section 8.2.3 [Rehearsal marks], page 187, but it can also be used to put signs like coda, segno, and fermata on a bar line. Use `\markup` to access the appropriate symbol (symbols are listed in Section C.4 [The Feta font], page 316)

```
c1 \mark \markup { \musicglyph #"scripts.ufermata" }
c1
```



`\mark` is only typeset above the top stave of the score. If you specify the `\mark` command at a bar line, the resulting mark is placed above the bar line. If you specify it in the middle of a bar, the resulting mark is positioned between notes. If it is specified before the beginning of a score line, it is placed before the first note of the line. Finally, if the mark occurs at a line break, the mark will be printed at the beginning of the next line. If there is no next line, then the mark will not be printed at all.

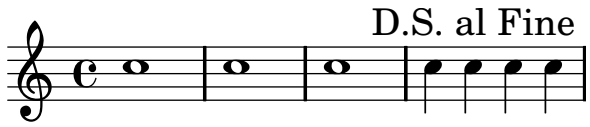
Commonly tweaked properties

To print the mark at the end of the current line, use

```
\override Score.RehearsalMark
  #'break-visibility = #begin-of-line-invisible
```

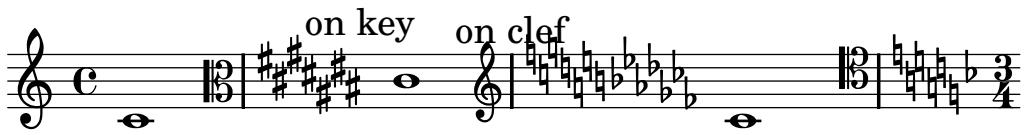
`\mark` is often useful for adding text to the end of bar. In such cases, changing the `#'self-alignment` is very useful

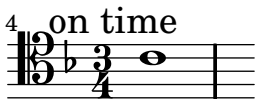
```
\override Score.RehearsalMark
  #'break-visibility = #begin-of-line-invisible
c1 c c c4 c c c
\once \override Score.RehearsalMark #'self-alignment-X = #right
\mark "D.S. al Fine "
```



Text marks may be aligned with notation objects other than bar lines,

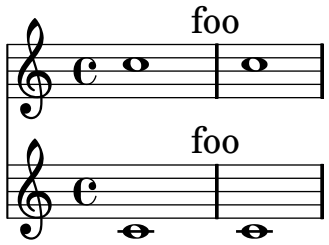
```
\relative {
  c1
  \key cis \major
  \clef alto
  \override Score.RehearsalMark #'break-align-symbol = #'key-signature
  \mark "on key"
  cis
  \key ces \major
  \override Score.RehearsalMark #'break-align-symbol = #'clef
  \clef treble
  \mark "on clef"
  ces
  \override Score.RehearsalMark #'break-align-symbol = #'time-signature
  \key d \minor
  \clef tenor
  \time 3/4
  \mark "on time"
  c
}
```





Although text marks are normally only printed above the topmost staff, you may alter this to print them on every staff,

```
{
  \new Score \with {
    \remove "Mark_engraver"
  }
  <<
    \new Staff \with {
      \consists "Mark_engraver"
    }
    { c''1 \mark "foo" c'' }
    \new Staff \with {
      \consists "Mark_engraver"
    }
    { c'1 \mark "foo" c' }
  >>
}
```



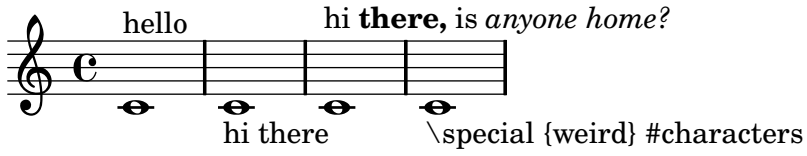
See also

Program reference: `RehearsalMark`.

8.1.4 Text markup

Use `\markup` to typeset text. Commands are entered with the backslash `\`. To enter `\` and `#`, use double quotation marks.

```
c1^\markup { hello }
c1_\markup { hi there }
c1^\markup { hi \bold there, is \italic {anyone home?} }
c1_\markup { "\special {weird} #characters" }
```

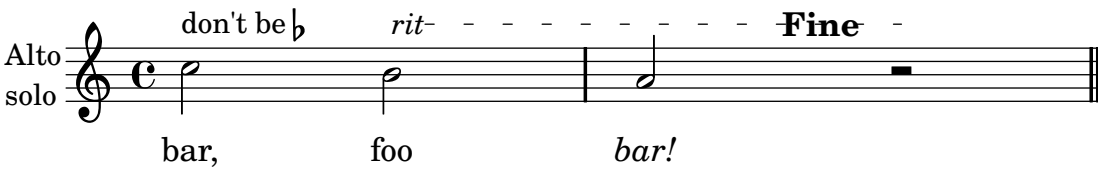


See Section 8.1.6 [Overview of text markup commands], page 174 for a list of all commands.

`\markup` is primarily used for `TextScripts`, but it can also be used anywhere text is called in lilypond

```
\header{ title = \markup{ \bold { foo \italic { bar! } } } }
\score{
  \relative c'' {
    \override Score.RehearsalMark
      #'break-visibility = #begin-of-line-invisible
    \override Score.RehearsalMark #'self-alignment-X = #right

    \set Staff.instrumentName = \markup{ \column{ Alto solo } }
    c2^\markup{ don't be \flat }
    \override TextSpanner #'edge-text = #(cons (markup #:italic "rit" ) "")
    b2\startTextSpan
    a2\mark \markup{ \large \bold Fine }
    r2\stopTextSpan
    \bar "||"
  }
  \addlyrics { bar, foo \markup{ \italic bar! } }
}
```

foo *bar!*



A `\markup` command can also be placed on its own, away from any `\score` block, see Section 10.1.4 [Multiple scores in a book], page 236.

```
\markup{ Here is some text. }
```

Here is some text.

The markup in the example demonstrates font switching commands. The command `\bold` and `\italic` apply to the first following word only; to apply a command to more than one word, enclose the words with braces,

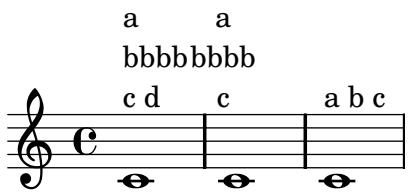
```
\markup { \bold { hi there } }
```

For clarity, you can also do this for single arguments, e.g.,

```
\markup { is \italic { anyone } home }
```

In markup mode you can compose expressions, similar to mathematical expressions, XML documents, and music expressions. You can stack expressions grouped vertically with the command `\column`. Similarly, `\center-align` aligns texts by their center lines:

```
c1^\markup { \column { a bbbb \line { c d } } }
c1^\markup { \center-align { a bbbb c } }
c1^\markup { \line { a b c } }
```



Lists with no previous command are not kept distinct. The expression

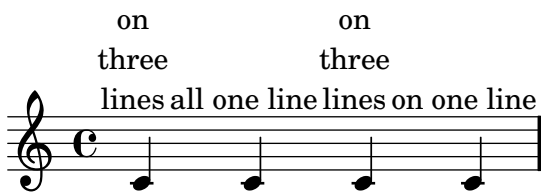
```
\center-align { { a b } { c d } }
```

is equivalent to

```
\center-align { a b c d }
```

To keep lists of words distinct, please use quotes `"` or the `\line` command

```
\fatText
c4^\markup{ \center-align { on three lines } }
c4^\markup{ \center-align { "all one line" } }
c4^\markup{ \center-align { { on three lines } } }
c4^\markup{ \center-align { \line { on one line } } }
```



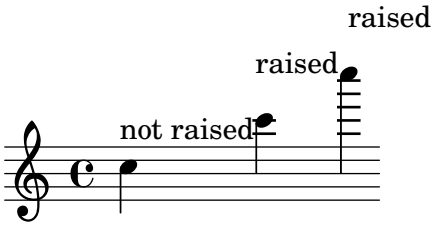
Markups can be stored in variables and these variables may be attached to notes, like

```
allegro = \markup { \bold \large { Allegro } }
 { a^\allegro b c d }
```

Some objects have alignment procedures of their own, which cancel out any effects of alignments applied to their markup arguments as a whole. For example, the `RehearsalMark` is horizontally centered, so using `\mark \markup { \left-align .. }` has no effect.

In addition, vertical placement is performed after creating the text markup object. If you wish to move an entire piece of markup, you need to use the #'padding property or create an "anchor" point inside the markup (generally with `\hspace #0`).

```
\fatText
c'4^\markup{ \raise #5 "not raised" }
\once \override TextScript #'padding = #3
c'4^\markup{ raised }
c'4^\markup{ \hspace #0 \raise #1.5 raised }
```



Some situations (such as dynamic marks) have preset font-related properties. If you are creating text in such situations, it is advisable to cancel those properties with `normal-text`. See Section 8.1.6 [Overview of text markup commands], page 174 for more details.

See also

This manual: Section 8.1.6 [Overview of text markup commands], page 174.

Program reference: `TextScript`.

Init files: '`scm/new-markup.scm`'.

Bugs

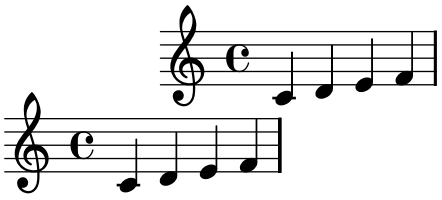
Kerning or generation of ligatures is only done when the TEX backend is used. In this case, LilyPond does not account for them so texts will be spaced slightly too wide.

Syntax errors for markup mode are confusing.

8.1.5 Nested scores

It is possible to nest music inside markups, by adding a `\score` block to a markup expression. Such a score must contain a `\layout` block.

```
\relative {
  c4 d^\markup {
    \score {
      \relative { c4 d e f }
      \layout { }
    }
  }
  e f
}
```



8.1.6 Overview of text markup commands

The following commands can all be used inside `\markup { }`.

`\arrow-head` *axis* (integer) *direction* (direction) *filled* (boolean)
: produce an arrow head in specified direction and axis. Use the filled head if *filled* is specified.

`\beam` *width* (number) *slope* (number) *thickness* (number)
: Create a beam with the specified parameters.

`\bigger` *arg* (markup)
: Increase the font size relative to current setting

`\bold` *arg* (markup)
: Switch to bold font-series

`\box` *arg* (markup)
: Draw a box round *arg*. Looks at `thickness`, `box-padding` and `font-size` properties to determine line thickness and padding around the markup.

`\bracket` *arg* (markup)
: Draw vertical brackets around *arg*.

`\bracketed-y-column` *indices* (list) *args* (list of markups)
: Make a column of the markups in *args*, putting brackets around the elements marked in *indices*, which is a list of numbers.

`\caps` *arg* (markup)
`\center-align` *args* (list of markups)
: Put `args` in a centered column.

`\char` *num* (integer)
: Produce a single character, e.g. `\char #65` produces the letter 'A'.

`\circle` *arg* (markup)
: Draw a circle around *arg*. Use `thickness`, `circle-padding` and `font-size` properties to determine line thickness and padding around the markup.

`\column` *args* (list of markups)
: Stack the markups in *args* vertically. The property `baseline-skip` determines the space between each markup in *args*.

`\combine` *m1* (markup) *m2* (markup)
: Print two markups on top of each other.

`\dir-column` *args* (list of markups)
: Make a column of args, going up or down, depending on the setting of the `#'direction` layout property.

`\doubleflat`
: Draw a double flat symbol.

`\doublesharp`
: Draw a double sharp symbol.

`\draw-circle` *radius* (number) *thickness* (number) *fill* (boolean)
: A circle of radius *radius*, thickness *thickness* and optionally filled.

`\dynamic` *arg* (markup)
: Use the dynamic font. This font only contains **s**, **f**, **m**, **z**, **p**, and **r**. When producing phrases, like "più **f**", the normal words (like "più") should be done in a different font. The recommend font for this is bold and italic

`\epsfile` *axis* (number) *size* (number) *file-name* (string)
: Inline an EPS image. The image is scaled along *axis* to *size*.

`\fill-line` *markups* (list of markups)
: Put *markups* in a horizontal line of width *line-width*. The markups are spaced/flushed to fill the entire line. If there are no arguments, return an empty stencil.

`\filled-box` *xext* (pair of numbers) *yext* (pair of numbers) *blot* (number)
: Draw a box with rounded corners of dimensions *xext* and *yext*. For example,

```
\filled-box #'(-.3 . 1.8) #'(-.3 . 1.8) #0
```

: create a box extending horizontally from -0.3 to 1.8 and vertically from -0.3 up to 1.8, with corners formed from a circle of diameter 0 (ie sharp corners).

`\finger` *arg* (markup)
: Set the argument as small numbers.

`\flat`
: Draw a flat symbol.

`\fontCaps` *arg* (markup)
: Set `font-shape` to `caps`.

`\fontsize` *increment* (number) *arg* (markup)
: Add *increment* to the font-size. Adjust baseline skip accordingly.

`\fraction` *arg1* (markup) *arg2* (markup)
: Make a fraction of two markups.

`\fret-diagram` *definition-string* (string)
: Example

```
\markup \fret-diagram #"s:0.75;6-x;5-x;4-o;3-2;2-3;1-2;"
```

: for fret spacing 3/4 of staff space, D chord diagram

: Syntax rules for *definition-string*:

  - Diagram items are separated by semicolons.
  - Possible items:
    - s:number – set the fret spacing of the diagram (in staff spaces). Default 1
    - t:number – set the line thickness (in staff spaces). Default 0.05
    - h:number – set the height of the diagram in frets. Default 4
    - w:number – set the width of the diagram in strings. Default 6
    - f:number – set fingering label type (0 = none, 1 = in circle on string, 2 = below string) Default 0
    - d:number – set radius of dot, in terms of fret spacing. Default 0.25
    - p:number – set the position of the dot in the fret space. 0.5 is centered; 1 is on lower fret bar, 0 is on upper fret bar. Default 0.6
    - c:string1-string2-fret – include a barre mark from string1 to string2 on fret
    - string-fret – place a dot on string at fret. If fret is o, string is identified as open. If fret is x, string is identified as muted.
    - string-fret-fingering – place a dot on string at fret, and label with fingering as defined by f: code.
  - Note: There is no limit to the number of fret indications per string.

`\fret-diagram-terse` *definition-string* (string)
: Make a fret diagram markup using terse string-based syntax.

: Example

```
\markup \fret-diagram-terse #"x;x;o;2;3;2;"
```

for a D chord diagram.

Syntax rules for *definition-string*:

- Strings are terminated by semicolons; the number of semicolons is the number of strings in the diagram.
- Mute strings are indicated by `"x"`.
- Open strings are indicated by `"o"`.
- A number indicates a fret indication at that fret.
- If there are multiple fret indicators desired on a string, they should be separated by spaces.
- Fingerings are given by following the fret number with a `"-"`, followed by the finger indicator, e.g. 3-2 for playing the third fret with the second finger.
- Where a barre indicator is desired, follow the fret (or fingering) symbol with `"-("` to start a barre and `"-)"` to end the barre.

`\fret-diagram-verbose` *marking-list* (list)

Make a fret diagram containing the symbols indicated in *marking-list*

For example,

```
\markup \fret-diagram-verbose #'((mute 6) (mute 5) (open 4)
    (place-fret 3 2) (place-fret 2 3) (place-fret 1 2))
```

will produce a standard D chord diagram without fingering indications.

Possible elements in *marking-list*:

(mute string-number)
: Place a small 'x' at the top of string *string-number*

(open string-number)
: Place a small 'o' at the top of string *string-number*

(barre start-string end-string fret-number)
: Place a barre indicator (much like a tie) from string *start-string* to string *end-string* at fret *fret-number*

(place-fret string-number fret-number finger-value)
: Place a fret playing indication on string *string-number* at fret *fret-number* with an optional fingering label *finger-value*. By default, the fret playing indicator is a solid dot. This can be changed by setting the value of the variable *dot-color*. If the *finger* part of the place-fret element is present, *finger-value* will be displayed according to the setting of the variable *finger-code*. There is no limit to the number of fret indications per string.

`\fromproperty` *symbol* (symbol)

Read the *symbol* from property settings, and produce a stencil from the markup contained within. If *symbol* is not defined, it returns an empty markup

`\general-align` *axis* (integer) *dir* (number) *arg* (markup)

Align *arg* in *axis* direction to the *dir* side.

`\halign` *dir* (number) *arg* (markup)

Set horizontal alignment. If *dir* is `-1`, then it is left-aligned, while `+1` is right. Values in between interpolate alignment accordingly.

`\hbracket` *arg* (markup)

Draw horizontal brackets around *arg*.

`\hcenter-in` *length* (number) *arg* (markup)
: Center *arg* horizontally within a box of extending *length*/2 to the left and right.

`\hcenter` *arg* (markup)
: Align `arg` to its X center.

`\hspace` *amount* (number)
: This produces a invisible object taking horizontal space.

```
\markup { A \hspace #2.0 B }
```

: will put extra space between A and B, on top of the space that is normally inserted before elements on a line.

`\huge` *arg* (markup)
: Set font size to +2.

`\italic` *arg* (markup)
: Use italic `font-shape` for *arg*.

`\justify-field` *symbol* (symbol)
`\justify` *args* (list of markups)
: Like wordwrap, but with lines stretched to justify the margins. Use `\override #'(line-width . X)` to set line-width, where X is the number of staff spaces.

`\justify-string` *arg* (string)
: Justify a string. Paragraphs may be separated with double newlines

`\large` *arg* (markup)
: Set font size to +1.

`\left-align` *arg* (markup)
: Align *arg* on its left edge.

`\line` *args* (list of markups)
: Put *args* in a horizontal line. The property `word-space` determines the space between each markup in *args*.

`\lookup` *glyph-name* (string)
: Lookup a glyph by name.

`\lower` *amount* (number) *arg* (markup)
: Lower *arg*, by the distance *amount*. A negative *amount* indicates raising, see also `\raise`.

`\magnify` *sz* (number) *arg* (markup)
: Set the font magnification for the its argument. In the following example, the middle A will be 10% larger:

```
A \magnify #1.1 { A } A
```

: Note: magnification only works if a font-name is explicitly selected. Use `\fontsize` otherwise.

`\markalphabet` *num* (integer)
: Make a markup letter for *num*. The letters start with A to Z and continues with double letters.

`\markletter` *num* (integer)
: Make a markup letter for *num*. The letters start with A to Z (skipping I), and continues with double letters.

`\medium` *arg* (markup)
: Switch to medium font-series (in contrast to bold).

`\musicglyph` *glyph-name* (string)
: This is converted to a musical symbol, e.g. `\musicglyph #"accidentals.0"` will select the natural sign from the music font. See user manual, `The Feta font` for a complete listing of the possible glyphs.

`\natural`
: Draw a natural symbol.

`\normal-size-sub` *arg* (markup)
: Set *arg* in subscript, in a normal font size.

`\normal-size-super` *arg* (markup)
: Set *arg* in superscript with a normal font size.

`\normal-text` *arg* (markup)
: Set all font related properties (except the size) to get the default normal text font, no matter what font was used earlier.

`\normalsize` *arg* (markup)
: Set font size to default.

`\note-by-number` *log* (number) *dot-count* (number) *dir* (number)
: Construct a note symbol, with stem. By using fractional values for *dir*, you can obtain longer or shorter stems.

`\note` *duration* (string) *dir* (number)
: This produces a note with a stem pointing in *dir* direction, with the *duration* for the note head type and augmentation dots. For example, `\note #"4." #-0.75` creates a dotted quarter note, with a shortened down stem.

`\null`
: An empty markup with extents of a single point

`\number` *arg* (markup)
: Set font family to `number`, which yields the font used for time signatures and fingerings. This font only contains numbers and some punctuation. It doesn't have any letters.

`\on-the-fly` *procedure* (symbol) *arg* (markup)
: Apply the *procedure* markup command to *arg*. *procedure* should take a single argument.

`\override` *new-prop* (pair) *arg* (markup)
: Add the first argument in to the property list. Properties may be any sort of property supported by `font-interface` and `text-interface`, for example

```
\override #'(font-family . married) "bla"
```

`\pad-around` *amount* (number) *arg* (markup)
: Add padding *amount* all around *arg*.

`\pad-markup` *padding* (number) *arg* (markup)
: Add space around a markup object.

`\pad-to-box` *x-ext* (pair of numbers) *y-ext* (pair of numbers) *arg* (markup)
: Make *arg* take at least *x-ext*, *y-ext* space

`\pad-x` *amount* (number) *arg* (markup)
: Add padding *amount* around *arg* in the X-direction.

`\postscript` *str* (string)
: This inserts *str* directly into the output as a PostScript command string. Due to technicalities of the output backends, different scales should be used for the TEX and PostScript backend, selected with `-f`.

  For the TeX backend, the following string prints a rotated text

  ```
  0 0 moveto /ecrm10 findfont
  1.75 scalefont setfont 90 rotate (hello) show
  ```

  The magical constant 1.75 scales from LilyPond units (staff spaces) to TeX dimensions.

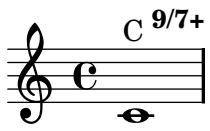
  For the postscript backend, use the following

  ```
  gsave /ecrm10 findfont
   10.0 output-scale div
   scalefont setfont 90 rotate (hello) show grestore
  ```

`\put-adjacent` *arg1* (markup) *axis* (integer) *dir* (direction) *arg2* (markup)
: Put *arg2* next to *arg1*, without moving *arg1*.

`\raise` *amount* (number) *arg* (markup)
: Raise *arg*, by the distance *amount*. A negative *amount* indicates lowering, see also `\lower`.

  ```
  c1^\markup { C \small \raise #1.0 \bold { "9/7+" }}
  ```



  The argument to `\raise` is the vertical displacement amount, measured in (global) staff spaces. `\raise` and `\super` raise objects in relation to their surrounding markups.

  If the text object itself is positioned above or below the staff, then `\raise` cannot be used to move it, since the mechanism that positions it next to the staff cancels any shift made with `\raise`. For vertical positioning, use the `padding` and/or `extra-offset` properties.

`\right-align` *arg* (markup)
: Align *arg* on its right edge.

`\roman` *arg* (markup)
: Set font family to `roman`.

`\rotate` *ang* (number) *arg* (markup)
: Rotate object with *ang* degrees around its center.

`\sans` *arg* (markup)
: Switch to the sans serif family

`\score` *score* (unknown)
: Inline an image of music.

`\semiflat`
: Draw a semiflat.

`\semisharp`
: Draw a semi sharp symbol.

`\sesquiflat`
: Draw a 3/2 flat symbol.

`\sesquisharp`
: Draw a 3/2 sharp symbol.

`\sharp`
: Draw a sharp symbol.

`\simple` *str* (string)
: A simple text string; `\markup { foo }` is equivalent with `\markup { \simple #"foo" }`.

`\slashed-digit` *num* (integer)
: A feta number, with slash. This is for use in the context of figured bass notation

`\small` *arg* (markup)
: Set font size to -1.

`\smallCaps` *text* (markup)
: Turn `text`, which should be a string, to small caps.

```
\markup \smallCaps "Text between double quotes"
```

`\smaller` *arg* (markup)
: Decrease the font size relative to current setting

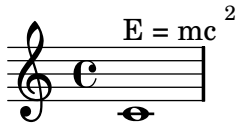
`\stencil` *stil* (unknown)
: Stencil as markup

`\strut`
: Create a box of the same height as the space in the current font.

`\sub` *arg* (markup)
: Set *arg* in subscript.

`\super` *arg* (markup)
: Raising and lowering texts can be done with `\super` and `\sub`:

```
c1^\markup { E "=" mc \super "2" }
```



`\teeny` *arg* (markup)
: Set font size to -3.

`\text` *arg* (markup)
: Use a text font instead of music symbol or music alphabet font.

`\tied-lyric` *str* (string)
: Like simple-markup, but use tie characters for ~ tilde symbols.

`\tiny` *arg* (markup)
: Set font size to -2.

`\translate` *offset* (pair of numbers) *arg* (markup)
: This translates an object. Its first argument is a cons of numbers

```
A \translate #(cons 2 -3) { B C } D
```

This moves 'B C' 2 spaces to the right, and 3 down, relative to its surroundings. This command cannot be used to move isolated scripts vertically, for the same reason that `\raise` cannot be used for that.

`\translate-scaled` *offset* (pair of numbers) *arg* (markup)
: Translate *arg* by *offset*, scaling the offset by the `font-size`.

`\transparent` *arg* (markup)
: Make the argument transparent

`\triangle` *filled* (boolean)
: A triangle, filled or not

`\typewriter` *arg* (markup)
: Use `font-family` typewriter for *arg*.

`\upright` *arg* (markup)
: Set font shape to `upright`. This is the opposite of `italic`.

`\vcenter` *arg* (markup)
: Align `arg` to its Y center.

`\verbatim-file` *name* (string)
: Read the contents of a file, and include verbatimly

`\whiteout` *arg* (markup)
: Provide a white underground for *arg*

`\with-color` *color* (list) *arg* (markup)
: Draw *arg* in color specified by *color*

`\with-dimensions` *x* (pair of numbers) *y* (pair of numbers) *arg* (markup)
: Set the dimensions of *arg* to *x* and *y*.

`\with-url` *url* (string) *arg* (markup)
: Add a link to URL *url* around *arg*. This only works in the PDF backend.

`\wordwrap-field` *symbol* (symbol)
`\wordwrap` *args* (list of markups)
: Simple wordwrap. Use `\override #'(line-width . X)` to set line-width, where X is the number of staff spaces.

`\wordwrap-string` *arg* (string)
: Wordwrap a string. Paragraphs may be separated with double newlines

8.1.7 Font selection

By setting the object properties described below, you can select a font from the preconfigured font families. LilyPond has default support for the feta music fonts. Text fonts are selected through Pango/FontConfig. The serif font defaults to New Century Schoolbook, the sans and typewriter to whatever the Pango installation defaults to.

- `font-encoding` is a symbol that sets layout of the glyphs. This should only be set to select different types of non-text fonts, e.g.

  `fetaBraces` for piano staff braces, `fetaMusic` the standard music font, including ancient glyphs, `fetaDynamic` for dynamic signs and `fetaNumber` for the number font.

- `font-family` is a symbol indicating the general class of the typeface. Supported are `roman` (Computer Modern), `sans`, and `typewriter`.

- `font-shape` is a symbol indicating the shape of the font. There are typically several font shapes available for each font family. Choices are `italic`, `caps`, and `upright`.
- `font-series` is a symbol indicating the series of the font. There are typically several font series for each font family and shape. Choices are `medium` and `bold`.

Fonts selected in the way sketched above come from a predefined style sheet. If you want to use a font from outside the style sheet, then set the `font-name` property,

```
{
  \override Staff.TimeSignature #'font-name = #"Charter"
  \override Staff.TimeSignature #'font-size = #2
  \time 3/4
  c'1_\markup {
    \override #'(font-name . "Vera Bold")
      { This text is in Vera Bold }
  }
}
```

Any font can be used, as long as it is available to Pango/FontConfig. To get a full list of all available fonts, run the command

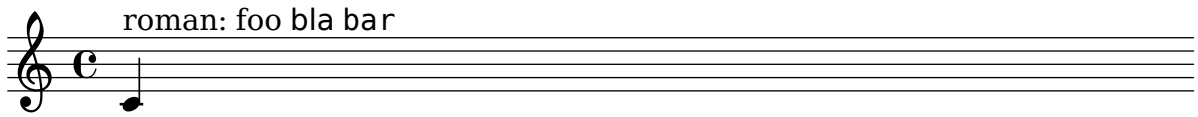
```
lilypond -dshow-available-fonts blabla
```

(the last argument of the command can be anything, but has to be present).

The size of the font may be set with the `font-size` property. The resulting size is taken relative to the `text-font-size` as defined in the `\paper` block.

It is also possible to change the default font family for the entire document. This is done by calling the `make-pango-font-tree` from within the `\paper` block. The function takes names for the font families to use for roman, sans serif and monospaced text. For example,

```
\paper  {
  myStaffSize = #20

  #(define fonts
    (make-pango-font-tree "Times New Roman"
                          "Nimbus Sans"
                          "Luxi Mono"
                           (/ myStaffSize 20)))
}

{
  c'^\markup { roman: foo \sans bla \typewriter bar }
}
```



See also

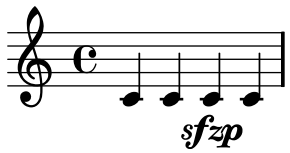
Examples: ‘input/regression/font-family-override.ly’.

8.1.8 New dynamic marks

It is possible to print new dynamic marks or text that should be aligned with dynamics. Use `make-dynamic-script` to create these marks. Note that the dynamic font only contains the characters `f`,`m`,`p`,`r`,`s` and `z`.

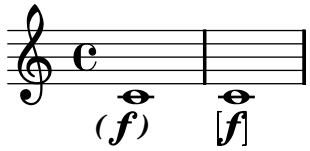
Some situations (such as dynamic marks) have preset font-related properties. If you are creating text in such situations, it is advisable to cancel those properties with `normal-text`. See Section 8.1.6 [Overview of text markup commands], page 174 for more details.

```
sfzp = #(make-dynamic-script "sfzp")
\relative c' {
  c4 c c\sfzp c
}
```



It is also possible to print dynamics in round parenthesis or square brackets. These are often used for adding editorial dynamics.

```
rndf = \markup{ \center-align {\line { \bold{\italic (}
  \dynamic f \bold{\italic )} }} }
boxf = \markup{ \bracket { \dynamic f } }
{ c'1_\rndf c'1_\boxf }
```



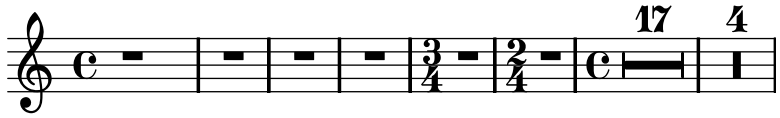
8.2 Preparing parts

This section describes various notation that are useful for preparing individual parts.

8.2.1 Multi measure rests

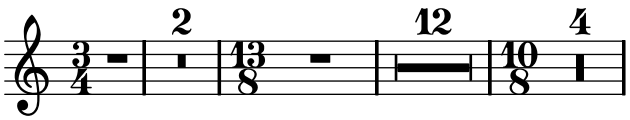
Rests for one full measure (or many bars) are entered using ‘R’. It is specifically meant for full bar rests and for entering parts: the rest can expand to fill a score with rests, or it can be printed as a single multi-measure rest. This expansion is controlled by the property `Score.skipBars`. If this is set to true, empty measures will not be expanded, and the appropriate number is added automatically

```
\time 4/4 r1 | R1 | R1*2 \time 3/4 R2. \time 2/4 R2 \time 4/4
\set Score.skipBars = ##t R1*17 R1*4
```



The `1` in `R1` is similar to the duration notation used for notes. Hence, for time signatures other than 4/4, you must enter other durations. This can be done with augmentation dots or fractions

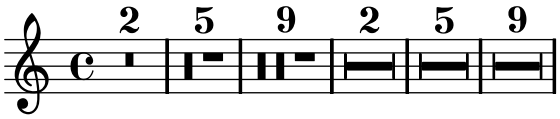
```
\set Score.skipBars = ##t
\time 3/4
R2. | R2.*2
\time 13/8
R1*13/8
R1*13/8*12 |
\time 10/8 R4*5*4 |
```



An `R` spanning a single measure is printed as either a whole rest or a breve, centered in the measure regardless of the time signature.

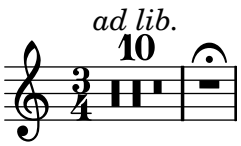
If there are only a few measures of rest, LilyPond prints "church rests" (a series of rectangles) in the staff. To replace that with a simple rest, use `MultiMeasureRest.expand-limit`.

```
\set Score.skipBars = ##t
R1*2 | R1*5 | R1*9
\override MultiMeasureRest #'expand-limit = 1
R1*2 | R1*5 | R1*9
```



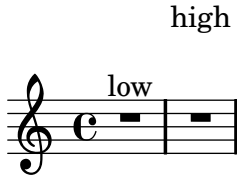
Texts can be added to multi-measure rests by using the *note*-`markup` syntax Section 8.1.4 [Text markup], page 170. A variable (`\fermataMarkup`) is provided for adding fermatas

```
\set Score.skipBars = ##t
\time 3/4
R2.*10^\markup { \italic "ad lib." }
R2.^\fermataMarkup
```



Warning! This text is created by `MultiMeasureRestText`, not `TextScript`.

```
\override TextScript #'padding = #5
R1^"low"
\override MultiMeasureRestText #'padding = #5
R1^"high"
```



If you want to have text on the left end of a multi-measure rest, attach the text to a zero-length skip note, i.e.,

```
s1*0^"Allegro"
R1*4
```

See also

Program reference: `MultiMeasureRestMusicGroup`, `MultiMeasureRest`.

The layout object `MultiMeasureRestNumber` is for the default number, and `MultiMeasureRestText` for user specified texts.

Bugs

It is not possible to use fingerings (e.g., `R1-4`) to put numbers over multi-measure rests. And the pitch of multi-measure rests (or staff-centered rests) can not be influenced.

There is no way to automatically condense multiple rests into a single multi-measure rest. Multi-measure rests do not take part in rest collisions.

Be careful when entering multi-measure rests followed by whole notes. The following will enter two notes lasting four measures each

```
R1*4 cis cis
```

When `skipBars` is set, the result will look OK, but the bar numbering will be off.

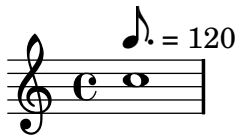
8.2.2 Metronome marks

Metronome settings can be entered as follows

```
\tempo duration = per-minute
```

In the MIDI output, they are interpreted as a tempo change. In the layout output, a metronome marking is printed

```
\tempo 8.=120 c''1
```



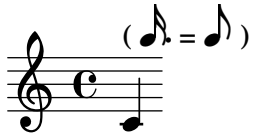
Commonly tweaked properties

To change the tempo in the MIDI output without printing anything, make the metronome marking invisible

```
\once \override Score.MetronomeMark #'transparent = ##t
```

To print other metronome markings, use these markup commands

```
c4^\markup {
  (
  \smaller \general-align #Y #DOWN \note #"16." #1
  =
  \smaller \general-align #Y #DOWN \note #"8" #1
  ) }
```



See Section 8.1.4 [Text markup], page 170 for more details.

See also

Program reference: `MetronomeMark`.

Bugs

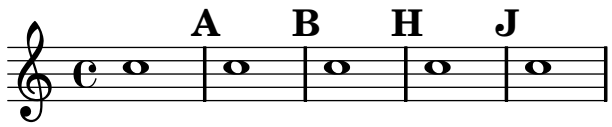
Collisions are not checked. If you have notes above the top line of the staff (or notes with articulations, slurs, text, etc), then the metronome marking may be printed on top of musical symbols. If this occurs, increase the padding of the metronome mark to place it further away from the staff.

```
\override Score.MetronomeMark #'padding = #2.5
```

8.2.3 Rehearsal marks

To print a rehearsal mark, use the `\mark` command

```
c1 \mark \default
c1 \mark \default
c1 \mark #8
c1 \mark \default
c1 \mark \default
```



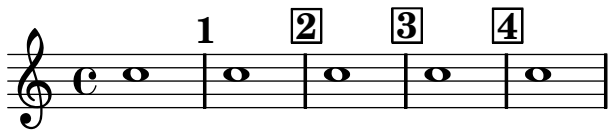
The letter 'I' is skipped in accordance with engraving traditions. If you wish to include the letter 'I', then use

```
\set Score.markFormatter = #format-mark-alphabet
```

The mark is incremented automatically if you use `\mark \default`, but you can also use an integer argument to set the mark manually. The value to use is stored in the property `rehearsalMark`.

The style is defined by the property `markFormatter`. It is a function taking the current mark (an integer) and the current context as argument. It should return a markup object. In the following example, `markFormatter` is set to a canned procedure. After a few measures, it is set to function that produces a boxed number.

```
\set Score.markFormatter = #format-mark-numbers
c1 \mark \default
c1 \mark \default
\set Score.markFormatter = #format-mark-box-numbers
c1 \mark \default
c1 \mark \default
c1
```



The file 'scm/translation-functions.scm' contains the definitions of format-mark-numbers (the default format), format-mark-box-numbers, format-mark-letters and format-mark-box-letters. These can be used as inspiration for other formatting functions.

You may use format-mark-barnumbers, format-mark-box-barnumbers, and format-mark-circle-barnumbers to get bar numbers instead of incremented numbers or letters.

Other styles of rehearsal mark can be specified manually

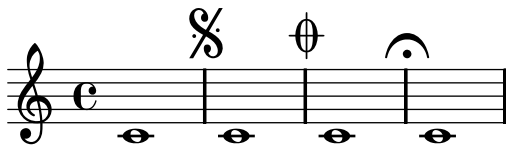
```
\mark "A1"
```

Score.markFormatter does not affect marks specified in this manner. However, it is possible to apply a \markup to the string.

```
\mark \markup{ \box A1 }
```

Music glyphs (such as the segno sign) may be printed inside a \mark

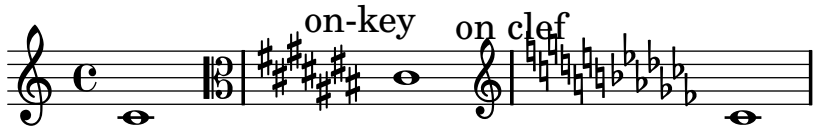
```
c1 \mark \markup { \musicglyph #"scripts.segno" }
c1 \mark \markup { \musicglyph #"scripts.coda" }
c1 \mark \markup { \musicglyph #"scripts.ufermata" }
c1
```



See Section C.4 [The Feta font], page 316 for a list of symbols which may be printed with \musicglyph.

The horizontal location of rehearsal marks can be adjusted by setting break-align-symbol

```
c1
\key cis \major
\clef alto
\override Score.RehearsalMark #'break-align-symbol = #'key-signature
\mark "on-key"
cis
\key ces \major
\override Score.RehearsalMark #'break-align-symbol = #'clef
\mark "on clef"
ces
```



break-align-symbol may also accept the following values: ambitus, breathing-sign, clef, custos, staff-bar, left-edge, key-cancellation, key-signature, and time-signature. Setting break-align-symbol will only have an effect if the symbol appears at that point in the music.

See also

This manual: Section 8.1.3 [Text marks], page 168.

Program reference: `RehearsalMark`.

Init files: '`scm/translation-functions.scm`' contains the definition of `format-mark-numbers` and `format-mark-letters`. They can be used as inspiration for other formatting functions.

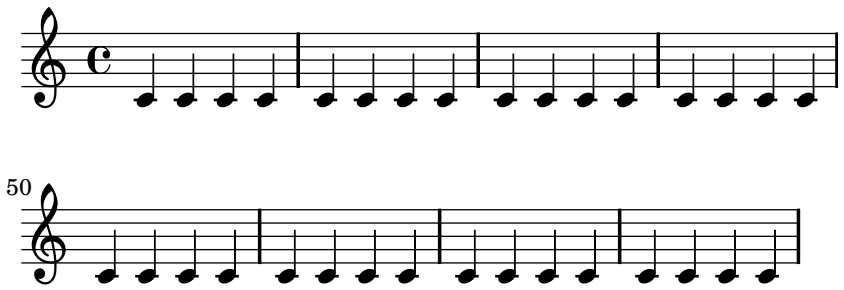
Examples: '`input/regression/rehearsal-mark-letter.ly`',

'`input/regression/rehearsal-mark-number.ly`'.

8.2.4 Bar numbers

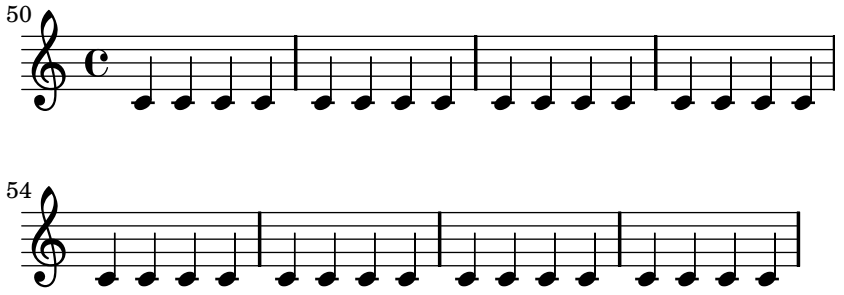
Bar numbers are printed by default at the start of the line. The number itself is stored in the `currentBarNumber` property, which is normally updated automatically for every measure.

```
\repeat unfold 4 {c4 c c c} \break
\set Score.currentBarNumber = #50
\repeat unfold 4 {c4 c c c}
```

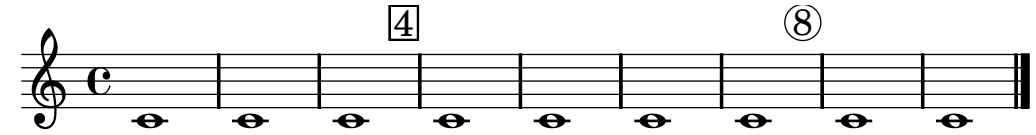


Bar numbers may only be printed at bar lines; to print a bar number at the beginning of a piece, an empty bar line must be added

```
\set Score.currentBarNumber = #50
\bar ""
\repeat unfold 4 {c4 c c c} \break
\repeat unfold 4 {c4 c c c}
```

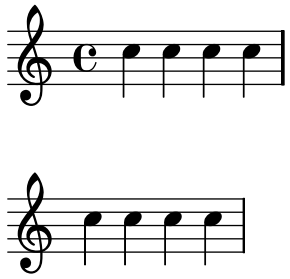


Bar numbers can be typeset at regular intervals instead of at the beginning of each line. This is illustrated in the following example, whose source is available as '`input/test/bar-number-regular-interval.ly`'



Bar numbers can be removed entirely by removing the Bar number engraver from the score.

```
\layout {
  \context {
    \Score
    \remove "Bar_number_engraver"
  }
}
\relative c''{
c4 c c c \break
c4 c c c
}
```



See also

Program reference: `BarNumber`.

Examples: ‘`input/test/bar-number-every-five-reset.ly`’, and ‘`input/test/bar-number-regular-interval.ly`’.

Bugs

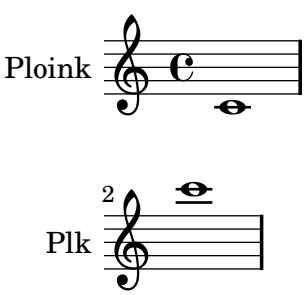
Bar numbers can collide with the `StaffGroup` bracket, if there is one at the top. To solve this, the `padding` property of `BarNumber` can be used to position the number correctly.

8.2.5 Instrument names

In an orchestral score, instrument names are printed at the left side of the staves.

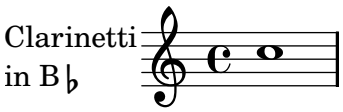
This can be achieved by setting `Staff.instrumentName` and `Staff.shortInstrumentName`, or `PianoStaff.instrumentName` and `PianoStaff.shortInstrumentName`. This will print text before the start of the staff. For the first staff, `instrumentName` is used, for the following ones, `shortInstrumentName` is used.

```
\set Staff.instrumentName = "Ploink "
\set Staff.shortInstrumentName = "Plk "
c1
\break
c''
```



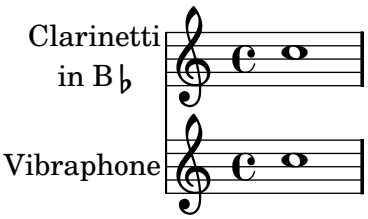
You can also use markup texts to construct more complicated instrument names, for example

```
\set Staff.instrumentName = \markup {
  \column { "Clarinetti"
            \line { "in B" \smaller \flat } } }
c''1
```



If you wish to center the instrument names, you must center all of them

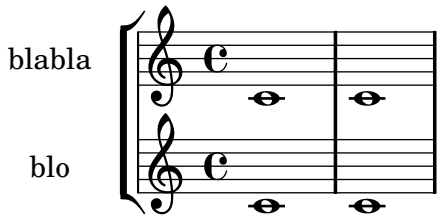
```
{ <<
\new Staff {
  \set Staff.instrumentName = \markup {
    \center-align { "Clarinetti"
      \line { "in B" \smaller \flat } } }
  c''1
}
\new Staff {
  \set Staff.instrumentName = \markup{ \center-align { Vibraphone }}
  c''1
}
>>
}
```



For longer instrument names, it may be useful to increase the `indent` setting in the `\layout` block.

To center instrument names while leaving extra space to the right,

```
\new StaffGroup \relative
<<
  \new Staff {
    \set Staff.instrumentName = \markup { \hcenter-in #10 "blabla" }
    c1 c1
  }
  \new Staff {
    \set Staff.instrumentName = \markup { \hcenter-in #10 "blo" }
    c1 c1
  }
>>
```



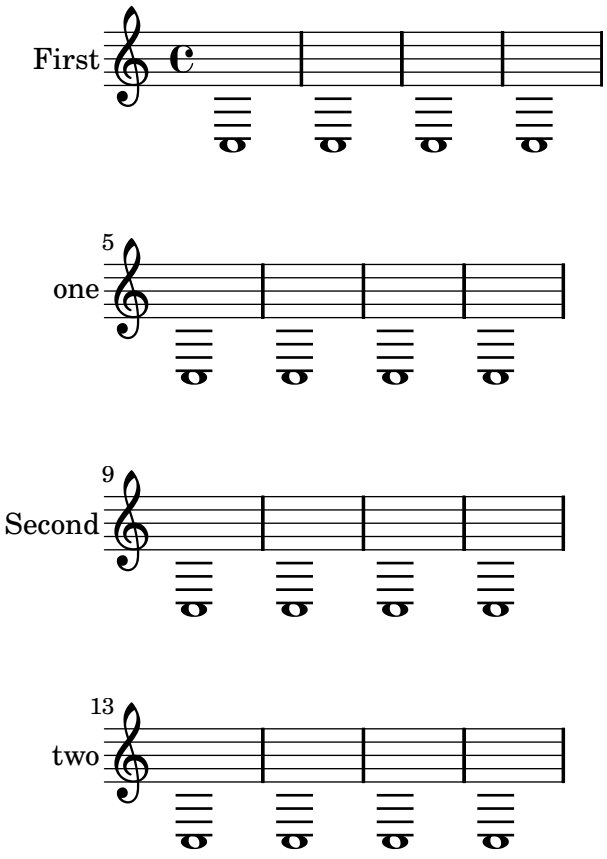
To add instrument names to other contexts (such as `GrandStaff`, `ChoirStaff`, or `StaffGroup`), the engraver must be added to that context.

```
\layout{
  \context {\GrandStaff \consists "Instrument_name_engraver"}
}
```

More information about adding and removing engravers can be found in Section 9.2.4 [Modifying context plug-ins], page 222.

Instrument names may be changed in the middle of a piece,

```
\set Staff.instrumentName = "First"
\set Staff.shortInstrumentName = "one"
c1 c c c \break
c1 c c c \break
\set Staff.instrumentName = "Second"
\set Staff.shortInstrumentName = "two"
c1 c c c \break
c1 c c c \break
```



See also

Program reference: `InstrumentName`.

8.2.6 Instrument transpositions

The key of a transposing instrument can also be specified. This applies to many wind instruments, for example, clarinets (B-flat, A, and E-flat), horn (F) and trumpet (B-flat, C, D, and E-flat).

The transposition is entered after the keyword `\transposition`

```
\transposition bes   %% B-flat clarinet
```

This command sets the property `instrumentTransposition`. The value of this property is used for MIDI output and quotations. It does not affect how notes are printed in the current staff. To change the printed output, see Section 6.1.8 [Transpose], page 63.

The pitch to use for `\transposition` should correspond to the real sound heard when a `c'` written on the staff is played by the transposing instrument. For example, when entering a score in concert pitch, typically all voices are entered in C, so they should be entered as

```
clarinet = {
  \transposition c'
  ...
}
saxophone = {
  \transposition c'
  ...
}
```

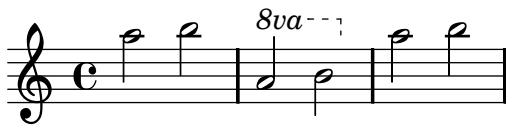
The command `\transposition` should be used when the music is entered from a (transposed) orchestral part. For example, in classical horn parts, the tuning of the instrument is often changed during a piece. When copying the notes from the part, use `\transposition`, e.g.,

```
\transposition d'
c'4^"in D"
...
\transposition g'
c'4^"in G"
...
```

8.2.7 Ottava brackets

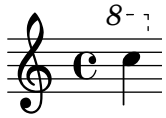
'Ottava' brackets introduce an extra transposition of an octave for the staff. They are created by invoking the function `set-octavation`

```
\relative c''' {
  a2 b
  #(set-octavation 1)
  a b
  #(set-octavation 0)
  a b
}
```



The `set-octavation` function also takes -1 (for 8va bassa), 2 (for 15ma), and -2 (for 15ma bassa) as arguments. Internally the function sets the properties `ottavation` (e.g., to `"8va"` or `"8vb"`) and `centralCPosition`. For overriding the text of the bracket, set `ottavation` after invoking `set-octavation`, i.e.,

```
{
  #(set-octavation 1)
  \set Staff.ottavation = #"8"
  c'''
}
```



See also

Program reference: `OttavaBracket`.

Examples: ‘`input/regression/ottava.ly`’, ‘`input/regression/ottava-broken.ly`’.

Bugs

`set-octavation` will get confused when clef changes happen during an octavation bracket.

8.2.8 Different editions from one source

The `\tag` command marks music expressions with a name. These tagged expressions can be filtered out later. With this mechanism it is possible to make different versions of the same music source.

In the following example, we see two versions of a piece of music, one for the full score, and one with cue notes for the instrumental part

```
c1
<<
  \tag #'part <<
    R1 \\
    {
      \set fontSize = #-1
      c4_"cue" f2 g4 }
  >>
  \tag #'score R1
>>
c1
```

The same can be applied to articulations, texts, etc.: they are made by prepending

```
-\tag #your-tag
```

to an articulation, for example,

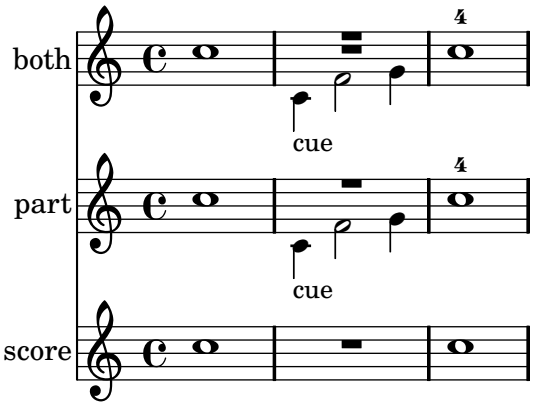
```
c1-\tag #'part ^4
```

This defines a note with a conditional fingering indication.

By applying the `\keepWithTag` and `\removeWithTag` commands, tagged expressions can be filtered. For example,

```
<<
  the music
  \keepWithTag #'score the music
  \keepWithTag #'part the music
>>
```

would yield



The arguments of the `\tag` command should be a symbol (such as `#'score` or `#'part`), followed by a music expression. It is possible to put multiple tags on a piece of music with multiple `\tag` entries,

```
\tag #'original-part \tag #'transposed-part ...
```

See also

Examples: '`input/regression/tag-filter.ly`'.

Bugs

Multiple rests are not merged if you create the score with both tagged sections.

8.3 Orchestral music

Orchestral music involves some special notation, both in the full score and the individual parts. This section explains how to tackle some common problems in orchestral music.

8.3.1 Automatic part combining

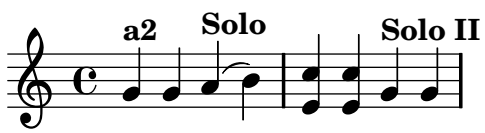
Automatic part combining is used to merge two parts of music onto a staff. It is aimed at typesetting orchestral scores. When the two parts are identical for a period of time, only one is shown. In places where the two parts differ, they are typeset as separate voices, and stem directions are set automatically. Also, solo and *a due* parts are identified and can be marked.

The syntax for part combining is

`\partcombine` *musicexpr1 musicexpr2*

The following example demonstrates the basic functionality of the part combiner: putting parts on one staff, and setting stem directions and polyphony

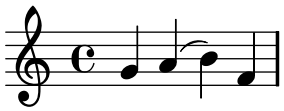
```
\new Staff \partcombine
  \relative g' { g g a( b) c c r r }
  \relative g' { g g r4 r e e g g }
```



The first `g` appears only once, although it was specified twice (once in each part). Stem, slur, and tie directions are set automatically, depending whether there is a solo or unisono. The first part (with context called `one`) always gets up stems, and 'Solo', while the second (called `two`) always gets down stems and 'Solo II'.

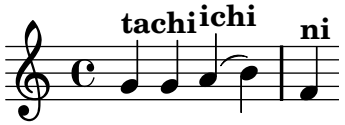
If you just want the merging parts, and not the textual markings, you may set the property `printPartCombineTexts` to false

```
\new Staff <<
  \set Staff.printPartCombineTexts = ##f
  \partcombine
    \relative g' { g a( b) r }
    \relative g' { g r4 r f }
>>
```



To change the text that is printed for solos or merging, you may set the `soloText`, `soloIIText`, and `aDueText` properties.

```
\new Staff <<
  \set Score.soloText = #"ichi"
  \set Score.soloIIText = #"ni"
  \set Score.aDueText = #"tachi"
  \partcombine
    \relative g' { g4 g a( b) r }
    \relative g' { g4 g r r f }
>>
```



Both arguments to `\partcombine` will be interpreted as `Voice` contexts. If using relative octaves, `\relative` should be specified for both music expressions, i.e.,

```
\partcombine
  \relative ... musicexpr1
  \relative ... musicexpr2
```

A `\relative` section that is outside of `\partcombine` has no effect on the pitches of *musicexpr1* and *musicexpr2*.

See also

Program reference: `PartCombineMusic`.

Bugs

When `printPartCombineTexts` is set, when the two voices play the same notes on and off, the part combiner may typeset `a2` more than once in a measure.

`\partcombine` cannot be inside `\times`.

`\partcombine` cannot be inside `\relative`.

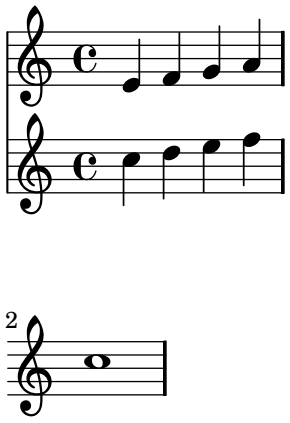
Internally, the `\partcombine` interprets both arguments as `Voices` named `one` and `two`, and then decides when the parts can be combined. Consequently, if the arguments switch to differently named `Voice` contexts, the events in those will be ignored.

8.3.2 Hiding staves

In orchestral scores, staff lines that only have rests are usually removed; this saves some space. This style is called 'French Score'. For `Lyrics`, `ChordNames` and `FiguredBass`, this is switched on by default. When the lines of these contexts turn out empty after the line-breaking process, they are removed.

For normal staves, a specialized `Staff` context is available, which does the same: staves containing nothing (or only multi-measure rests) are removed. The context definition is stored in `\RemoveEmptyStaffContext` variable. Observe how the second staff in this example disappears in the second line

```
\layout {
  \context { \RemoveEmptyStaffContext }
}

{
  \relative c' <<
    \new Staff { e4 f g a \break c1 }
    \new Staff { c4 d e f \break R1 }
  >>
}
```



The first system shows all staves in full. If empty staves should be removed from the first system too, set `remove-first` to true in `VerticalAxisGroup`.

```
\override Score.VerticalAxisGroup #'remove-first = ##t
```

To remove other types of contexts, use `\AncientRemoveEmptyStaffContext` or `\RemoveEmptyRhythmicStaffContext`.

Another application is making ossia sections, i.e., alternative melodies on a separate piece of staff, with help of a Frenched staff.

8.3.3 Quoting other voices

With quotations, fragments of other parts can be inserted into a part directly. Before a part can be quoted, it must be marked especially as quotable. This is done with the `\addquote` command.

```
\addquote name music
```

Here, *name* is an identifying string. The *music* is any kind of music. Here is an example of `\addquote`

```
\addquote clarinet \relative c' {
  f4 fis g gis
}
```

This command must be entered at toplevel, i.e., outside any music blocks.

After calling `\addquote`, the quotation may then be done with `\quoteDuring` or `\cueDuring`,

```
\quoteDuring #name music
```

During a part, a piece of music can be quoted with the `\quoteDuring` command.

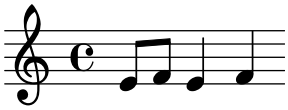
```
\quoteDuring #"clarinet" { s2. }
```

This would cite three quarter notes (the duration of `s2.`) of the previously added `clarinet` voice.

More precisely, it takes the current time-step of the part being printed, and extracts the notes at the corresponding point of the `\addquote`d voice. Therefore, the argument to `\addquote` should be the entire part of the voice to be quoted, including any rests at the beginning.

Quotations take into account the transposition of both source and target instruments, if they are specified using the `\transposition` command.

```
\addquote clarinet \relative c' {
  \transposition bes
  f4 fis g gis
}

{
  e'8 f'8 \quoteDuring #"clarinet" { s2 }
}
```



The type of events that are present in cue notes can be trimmed with the `quotedEventTypes` property. The default value is `(note-event rest-event)`, which means that only notes and rests of the cued voice end up in the `\quoteDuring`. Setting

```
\set Staff.quotedEventTypes =
       #'(note-event articulation-event dynamic-event)
```

will quote notes (but no rests), together with scripts and dynamics.

Bugs

Only the contents of the first `Voice` occurring in an `\addquote` command will be considered for quotation, so *music* can not contain `\new` and `\context Voice` statements that would switch to a different Voice.

Quoting grace notes is broken and can even cause LilyPond to crash.

Quoting nested triplets may result in poor notation.

See also

In this manual: Section 8.2.6 [Instrument transpositions], page 193.

Examples: ‘`input/regression/quote.ly`’ ‘`input/regression/quote-transposition.ly`’

Program reference: `QuoteMusic`.

8.3.4 Formatting cue notes

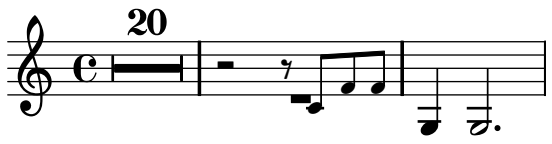
The previous section deals with inserting notes from another voice. There is a more advanced music function called `\cueDuring`, which makes formatting cue notes easier.

The syntax is

```
\cueDuring #name #updown music
```

This will insert notes from the part *name* into a `Voice` called `cue`. This happens simultaneously with *music*, which usually is a rest. When the cue notes start, the staff in effect becomes polyphonic for a moment. The argument *updown* determines whether the cue notes should be notated as a first or second voice.

```
smaller = {
  \set fontSize = #-2
  \override Stem #'length-fraction = #0.8
  \override Beam #'thickness = #0.384
  \override Beam #'length-fraction = #0.8
}

\addquote clarinet \relative {
  R1*20
  r2 r8 c f f
}

\new Staff \relative  <<

  % setup a context for  cue  notes.
  \new Voice = "cue" { \smaller \skip 1*21 }

  \set Score.skipBars = ##t

  \new Voice {
    R1*20
    \cueDuring #"clarinet" #1 {
      R1
    }
    g4 g2.
  }
>>
```



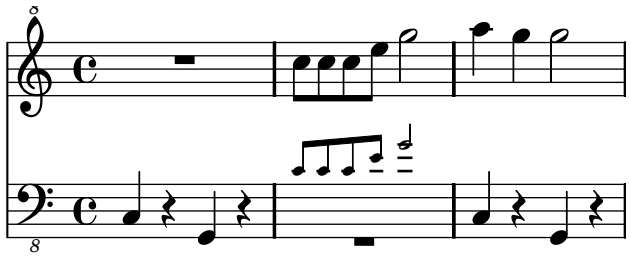
Here are a couple of hints for successful cue notes

- Cue notes have smaller font sizes.
- the cued part is marked with the instrument playing the cue.
- when the original part takes over again, this should be marked with the name of the original instrument.

  Any other changes introduced by the cued part should also be undone. For example, if the cued instrument plays in a different clef, the original clef should be stated once again.

The macro `\transposedCueDuring` is useful to add cues to instruments which use a completely different octave range (for example, having a cue of a piccolo flute within a contra bassoon part).

```
picc = \relative c''' {
  \clef "treble^8"
  R1 |
  c8 c c e g2 |
  a4 g g2 |
}
\addquote "picc" { \picc }

cbsn = \relative c, {
  \clef "bass_8"
  c4 r g r
  \transposedCueDuring #"picc" #UP c,, { R1 } |
  c4 r g r |
}

<<
  \context Staff = "picc" \picc
  \context Staff = "cbsn" \cbsn
>>
```

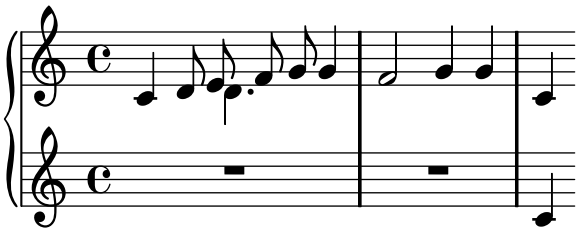


8.3.5 Aligning to cadenzas

In an orchestral context, cadenzas present a special problem: when constructing a score that includes a cadenza, all other instruments should skip just as many notes as the length of the cadenza, otherwise they will start too soon or too late.

A solution to this problem are the functions `mmrest-of-length` and `skip-of-length`. These Scheme functions take a piece of music as argument, and generate a `\skip` or multi-rest, exactly as long as the piece. The use of `mmrest-of-length` is demonstrated in the following example.

```
cadenza = \relative c' {
  c4 d8 << { e f g } \\ { d4. } >>
  g4 f2 g4 g
}

\new GrandStaff <<
  \new Staff { \cadenza c'4 }
  \new Staff {
    #(ly:export (mmrest-of-length cadenza))
    c'4
  }
>>
```



8.4 Contemporary notation

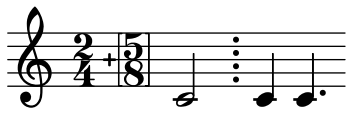
In the 20th century, composers have greatly expanded the musical vocabulary. With this expansion, many innovations in musical notation have been tried. The book "Music Notation in the 20th century" by Kurt Stone gives a comprehensive overview (see Appendix A [Literature list], page 309).

This section describes notation that does not fit into traditional notation categories, such as pitches, tuplet beams, and articulation. For contemporary notation that fits into traditional notation categories, such as microtones, nested tuplet beams, and unusual fermatas, please see those sections of the documentation.

8.4.1 Polymetric notation

Double time signatures are not supported explicitly, but they can be faked. In the next example, the markup for the time signature is created with a markup text. This markup text is inserted in the `TimeSignature` grob. See also '`input/test/compound-time.ly`').

```
% create 2/4 + 5/8
tsMarkup =\markup {
  \override #'(baseline-skip . 2) \number {
    \column { "2" "4" }
    \vcenter "+"
    \bracket \column { "5" "8" }
  }
}

{
  \override Staff.TimeSignature #'stencil = #ly:text-interface::print
  \override Staff.TimeSignature #'text = #tsMarkup
  \time 3/2
  c'2 \bar ":" c'4 c'4.
}
```



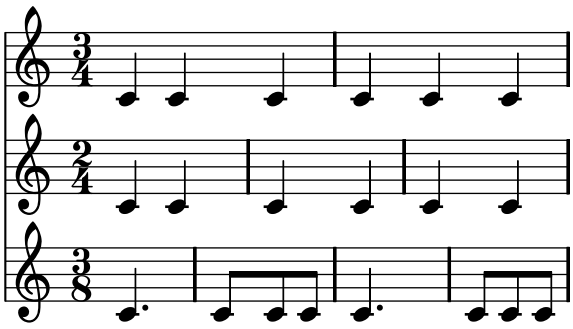
Each staff can also have its own time signature. This is done by moving the `Timing_translator` to the `Staff` context.

```
    \layout {
      \context { \Score
         \remove "Timing_translator"
         \remove "Default_bar_line_engraver"
      }
      \context {
        \Staff
        \consists "Timing_translator"
        \consists "Default_bar_line_engraver"
```

```
  }

}
```

Now, each staff has its own time signature.

```
<<
  \new Staff {
    \time 3/4
    c4 c c | c c c |
  }
  \new Staff {
    \time 2/4
    c4 c | c c | c c
  }
  \new Staff {
    \time 3/8
    c4. c8 c c c4. c8 c c
  }
>>
```



A different form of polymetric notation is where note lengths have different values across staves.

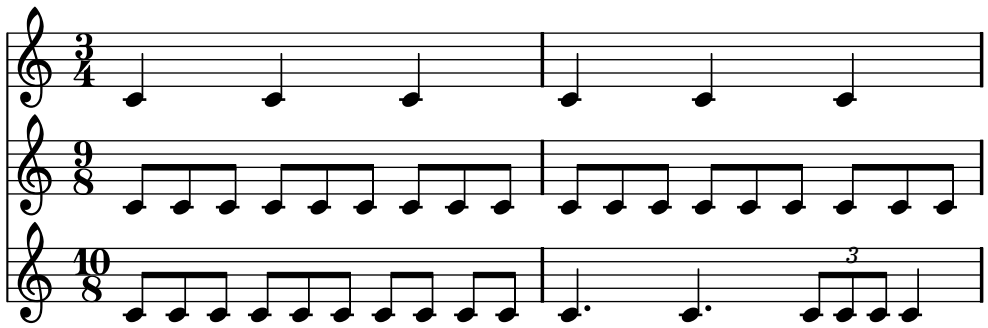
This notation can be created by setting a common time signature for each staff but replacing it manually using `timeSignatureFraction` to the desired fraction. Then the printed durations in each staff are scaled to the common time signature. The latter is done with `\compressMusic`, which is used similar to `\times`, but does not create a tuplet bracket. The syntax is

```
\compressMusic #'(numerator . denominator) musicexpr
```

In this example, music with the time signatures of 3/4, 9/8, and 10/8 are used in parallel. In the second staff, shown durations are multiplied by 2/3, so that 2/3 * 9/8 = 3/4, and in the third staff, shown durations are multiplied by 3/5, so that 3/5 * 10/8 = 3/4.

```
\relative c' { <<
  \new Staff {
    \time 3/4
    c4 c c | c c c |
  }
  \new Staff {
    \time 3/4
    \set Staff.timeSignatureFraction = #'(9 . 8)
    \compressMusic #'(2 . 3)
      \repeat unfold 6 { c8[ c c] }
  }
```

```
    \new Staff {
      \time 3/4
      \set Staff.timeSignatureFraction = #'(10 . 8)
      \compressMusic #'(3 . 5) {
        \repeat unfold 2 { c8[ c c] }
        \repeat unfold 2 { c8[ c] }
        | c4. c4. \times 2/3 { c8 c c } c4
      }
    }
  >> }
```



Bugs

When using different time signatures in parallel, the spacing is aligned vertically, but bar lines distort the regular spacing.

8.4.2 Time administration

Time is administered by the `Time_signature_engraver`, which usually lives in the `Score` context. The bookkeeping deals with the following variables

`currentBarNumber`
: The measure number.

`measureLength`
: The length of the measures in the current time signature. For a 4/4 time this is 1, and for 6/8 it is 3/4.

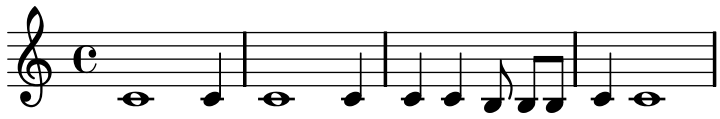
`measurePosition`
: The point within the measure where we currently are. This quantity is reset to 0 whenever it exceeds `measureLength`. When that happens, `currentBarNumber` is incremented.

`timing`
: If set to true, the above variables are updated for every time step. When set to false, the engraver stays in the current measure indefinitely.

Timing can be changed by setting any of these variables explicitly. In the next example, the 4/4 time signature is printed, but `measureLength` is set to 5/4. After a while, the measure is shortened by 1/8, by setting `measurePosition` to 7/8 at 2/4 in the measure, so the next bar line will fall at 2/4 + 3/8. The 3/8 arises because 5/4 normally has 10/8, but we have manually set the measure position to be 7/8 and 10/8 - 7/8 = 3/8.

```
\set Score.measureLength = #(ly:make-moment 5 4)
c1 c4
c1 c4
c4 c4
```

```
\set Score.measurePosition = #(ly:make-moment 7 8)
b8 b b
c4 c1
```

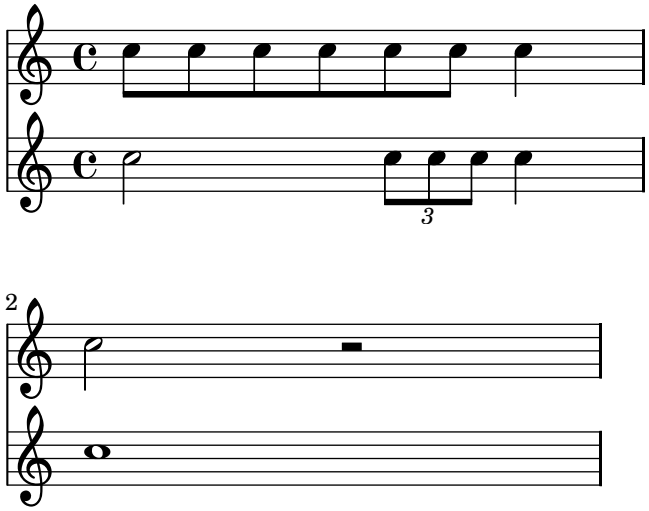


As the example illustrates, `ly:make-moment n m` constructs a duration of n/m of a whole note. For example, `ly:make-moment 1 8` is an eighth note duration and `ly:make-moment 7 16` is the duration of seven sixteenths notes.

8.4.3 Proportional notation

Notes can be spaced proportionally to their time-difference by assigning a duration to `proportionalNotationDuration`

```
<<
  \set Score.proportionalNotationDuration = #(ly:make-moment 1 16)
  \new Staff { c8[ c c c c c]  c4 c2 r2 }
  \new Staff { c2  \times 2/3 { c8 c c } c4 c1 }
>>
```



Setting this property only affects the ideal spacing between consecutive notes. For true proportional notation, the following settings are also required.

- True proportional notation requires that symbols are allowed to overstrike each other. That is achieved by removing the `Separating_line_group_engraver` from `Staff` context.
- Spacing influence of prefatory matter (clefs, bar lines, etc.) is removed by setting the `strict-note-spacing` property to `#t` in `SpacingSpanner` grob.
- Optical spacing tweaks are switched by setting `uniform-stretching` in `SpacingSpanner` to true.

See also

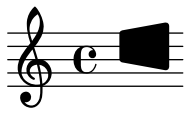
‘`input/regression/spacing-proportional/.ly`’ ‘`input/regression/spacing-strict-notespacing/.ly`’ ‘`input/regression/spacing-strict-spacing-grace/.ly`’

An example of strict proportional notation is in the example file ‘`input/proportional.ly`’.

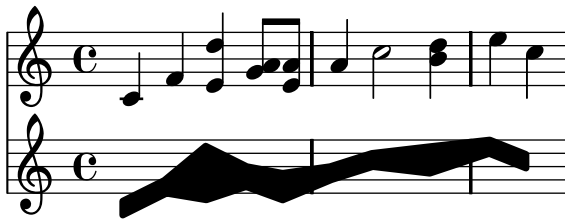
8.4.4 Clusters

A cluster indicates a continuous range of pitches to be played. They can be denoted as the envelope of a set of notes. They are entered by applying the function `makeClusters` to a sequence of chords, e.g.,

```
\makeClusters { <c e > <b f'> }
```



The following example (from '`input/regression/cluster.ly`') shows what the result looks like



Ordinary notes and clusters can be put together in the same staff, even simultaneously. In such a case no attempt is made to automatically avoid collisions between ordinary notes and clusters.

See also

Program reference: `ClusterSpanner`, `ClusterSpannerBeacon`, `Cluster_spanner_engraver`.

Examples: '`input/regression/cluster.ly`'.

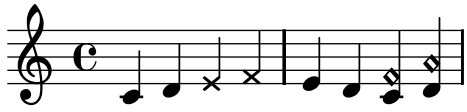
Bugs

Music expressions like `<< { g8 e8 } a4 >>` are not printed accurately. Use `<g a>8 <e a>8` instead.

8.4.5 Special noteheads

Different noteheads are used by various instruments for various meanings – crosses are used for "parlato" with vocalists, stopped notes on guitar; diamonds are used for harmonics on string instruments, etc. There is a shorthand (`\harmonic`) for diamond shapes; the other notehead styles are produced by tweaking the property

```
c4 d
\override NoteHead #'style = #'cross
e f
\revert NoteHead #'style
e d <c f\harmonic> <d a'\harmonic>
```



To see all notehead styles, please see '`input/regression/note-head-style.ly`'.

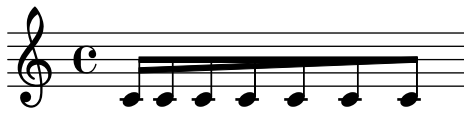
See also

Program reference: `NoteHead`.

8.4.6 Feathered beams

Feathered beams are printed by setting the `grow-direction` property of a `Beam`. The `\featherDurations` function can be used to adjust note durations.

```
\featherDurations #(ly:make-moment 5 4)
{
  \override Beam #'grow-direction = #LEFT
  c16[ c c c c c c]
}
```



Bugs

The `\featherDuration` command only works with very short music snippets.

8.4.7 Improvisation

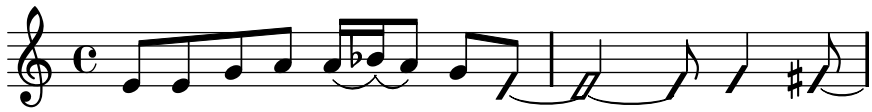
Improvisation is sometimes denoted with slashed note heads. Such note heads can be created by adding a `Pitch_squash_engraver` to the `Staff` or `Voice` context. Then, the following command

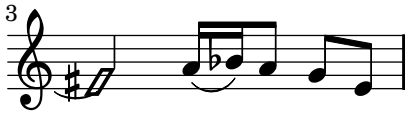
```
\set squashedPosition = #0
\override NoteHead #'style = #'slash
```

switches on the slashes.

There are shortcuts `\improvisationOn` (and an accompanying `\improvisationOff`) for this command sequence. They are used in the following example

```
\new Staff \with {
  \consists Pitch_squash_engraver
} \transpose c c' {
  e8 e g a a16(bes)(a8) g \improvisationOn
  e8
  ~e2~e8 f4 fis8
  ~fis2 \improvisationOff a16(bes) a8 g e
}
```





8.4.8 Selecting notation font size

The easiest method of setting the font size of any context is by setting the `fontSize` property.

```
c8
\set fontSize = #-4
c f
\set fontSize = #3
g
```

It does not change the size of variable symbols, such as beams or slurs.

Internally, the `fontSize` context property will cause the `font-size` property to be set in all layout objects. The value of `font-size` is a number indicating the size relative to the standard size for the current staff height. Each step up is an increase of approximately 12% of the font size. Six steps is exactly a factor two. The Scheme function `magstep` converts a `font-size` number to a scaling factor. The `font-size` property can also be set directly, so that only certain layout objects are affected.

```
c8
\override NoteHead #'font-size = #-4
c f
\override NoteHead #'font-size = #3
g
```

Font size changes are achieved by scaling the design size that is closest to the desired size. The standard font size (for `font-size` equals 0), depends on the standard staff height. For a 20pt staff, a 10pt font is selected.

The `font-size` property can only be set on layout objects that use fonts. These are the ones supporting the `font-interface` layout interface.

Predefined commands

The following commands set `fontSize` for the current voice:

`\tiny`, `\small`, `\normalsize`.

8.5 Educational use

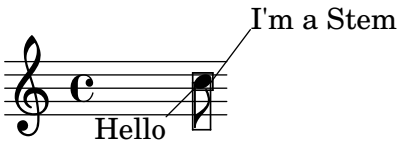
With the amount of control that LilyPond offers, one can make great teaching tools in addition to great musical scores.

8.5.1 Balloon help

Elements of notation can be marked and named with the help of a square balloon. The primary purpose of this feature is to explain notation.

The following example demonstrates its use.

```
\new Voice \with { \consists "Balloon_engraver" }
{
  \balloonGrobText #'Stem #'(3 . 4) \markup { "I'm a Stem" }
  <c-\balloonText #'(-2 . -2) \markup { Hello }  >8
}
```



There are two music functions, `balloonText` and `balloonGrobText`. The latter takes the name of the grob to adorn, while the former may be used as an articulation on a note. The other arguments are the offset and the text of the label.

See also

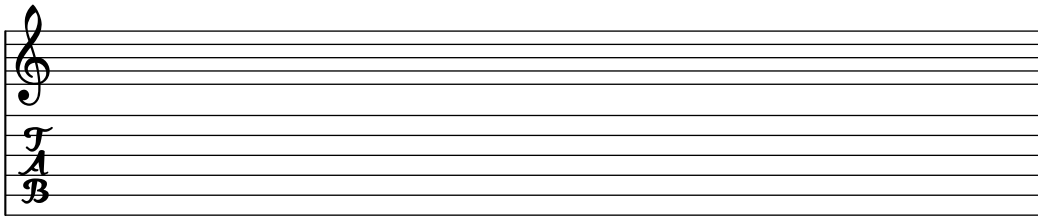
Program reference: `text-balloon-interface`.

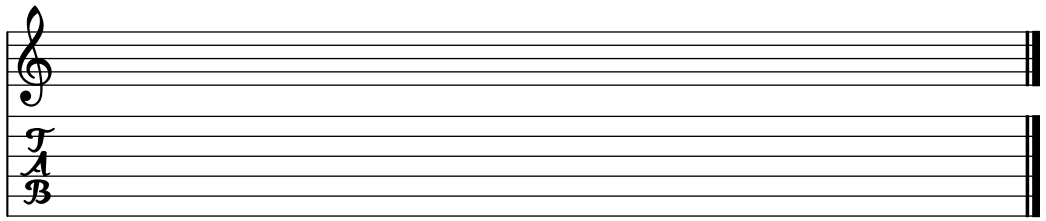
Examples: '`input/regression/balloon.ly`'.

8.5.2 Blank music sheet

A blank music sheet can be produced also by using invisible notes, and removing `Bar_number_engraver`.

```
\layout{ indent = #0 }
emptymusic = {
  \repeat unfold 2 % Change this for more lines.
  { s1\break }
  \bar "|."
}
\new Score \with {
  \override TimeSignature #'transparent = ##t
% un-comment this line if desired
%  \override Clef #'transparent = ##t
  defaultBarType = #""
  \remove Bar_number_engraver
} <<

% modify these to get the staves you want
  \new Staff \emptymusic
  \new TabStaff \emptymusic
>>
```

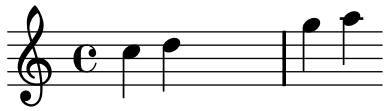




8.5.3 Hidden notes

Hidden (or invisible or transparent) notes can be useful in preparing theory or composition exercises.

```
c4 d4
\hideNotes
e4 f4
\unHideNotes
g4 a
```

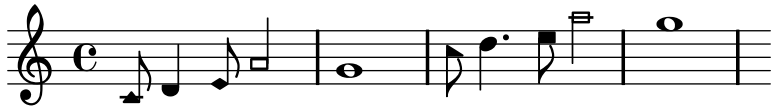


8.5.4 Shape note heads

In shape note head notation, the shape of the note head corresponds to the harmonic function of a note in the scale. This notation was popular in the 19th century American song books.

Shape note heads can be produced by setting `\aikenHeads` or `\sacredHarpHeads`, depending on the style desired.

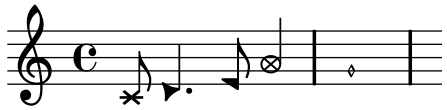
```
\aikenHeads
c8 d4 e8 a2 g1
\sacredHarpHeads
c8 d4. e8 a2 g1
```



Shapes are determined on the step in the scale, where the base of the scale is determined by the `\key` command

Shape note heads are implemented through the `shapeNoteStyles` property. Its value is a vector of symbols. The k-th element indicates the style to use for the k-th step of the scale. Arbitrary combinations are possible, e.g.

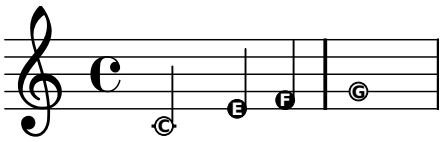
```
\set shapeNoteStyles  = ##(cross triangle fa #f mensural xcircle diamond)
c8 d4. e8 a2 g1
```



8.5.5 Easy Notation note heads

The ‘easy play’ note head includes a note name inside the head. It is used in music for beginners

```
\setEasyHeads
c'2 e'4 f' | g'1
```



The command `\setEasyHeads` overrides settings for the `NoteHead` object. To make the letters readable, it has to be printed in a large font size. To print with a larger font, see Section 11.2.1 [Setting the staff size], page 250.

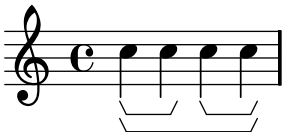
Predefined commands

`\setEasyHeads`

8.5.6 Analysis brackets

Brackets are used in musical analysis to indicate structure in musical pieces. LilyPond supports a simple form of nested horizontal brackets. To use this, add the `Horizontal_bracket_engraver` to `Staff` context. A bracket is started with `\startGroup` and closed with `\stopGroup`

```
\score {
  \relative c'' {
    c4\startGroup\startGroup
    c4\stopGroup
    c4\startGroup
    c4\stopGroup\stopGroup
  }
  \layout {
    \context {
      \Staff \consists "Horizontal_bracket_engraver"
}}}
```



See also

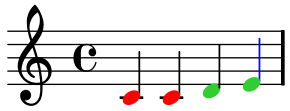
Program reference: `HorizontalBracket`.

Examples: ‘`input/regression/note-group-bracket.ly`’.

8.5.7 Coloring objects

Individual objects may be assigned colors. You may use the color names listed in the Section C.3 [List of colors], page 314.

```
\override NoteHead #'color = #red
c4 c
\override NoteHead #'color = #(x11-color 'LimeGreen)
d
\override Stem #'color = #blue
e
```



The full range of colors defined for X11 can be accessed by using the Scheme function x11-color. The function takes one argument that can be a symbol

```
\override Beam #'color = #(x11-color 'MediumTurquoise)
```

or a string

```
\override Beam #'color = #(x11-color "MediumTurquoise")
```

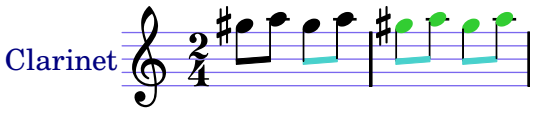
The first form is quicker to write and is more efficient. However, using the second form it is possible to access X11 colors by the multi-word form of its name

```
\override Beam #'color = #(x11-color "medium turquoise")
```

If x11-color cannot make sense of the parameter then the color returned defaults to black. It should be obvious from the final score that something is wrong.

This example illustrates the use of x11-color. Notice that the stem color remains black after being set to (x11-color 'Boggle), which is deliberate nonsense.

```
{
  \override Staff.StaffSymbol #'color = #(x11-color 'SlateBlue2)
  \set Staff.instrumentName = \markup {
    \with-color #(x11-color 'navy) "Clarinet"
  }
  \time 2/4
  gis''8 a''
  \override Beam #'color = #(x11-color "medium turquoise")
  gis'' a''
  \override NoteHead #'color = #(x11-color "LimeGreen")
  gis'' a''
  \override Stem #'color = #(x11-color 'Boggle)
  gis'' a''
}
```



See also

Appendix: Section C.3 [List of colors], page 314.

Bugs

Not all x11 colors are distinguishable in a web browser. For web use normal colors are recommended.

An x11 color is not necessarily exactly the same shade as a similarly named normal color.

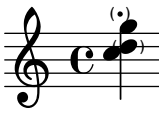
Notes in a chord cannot be colored with `\override`; use `\tweak` instead. See Section 9.3.5 [Objects connected to the input], page 231 for details.

8.5.8 Parentheses

Objects may be parenthesized by prefixing `\parenthesize` to the music event,

```
<
  c
```

```
  \parenthesize d
  g
>4-\parenthesize -.
```



This only functions inside chords, even for single notes

```
< \parenthesize NOTE>
```

8.5.9 Grid lines

Vertical lines can be drawn between staves synchronized with the notes.

Examples: ‘`input/regression/grid-lines.ly`’.

9 Changing defaults

The purpose of LilyPond's design is to provide the finest output quality as a default. Nevertheless, it may happen that you need to change this default layout. The layout is controlled through a large number of proverbial "knobs and switches." This chapter does not list each and every knob. Rather, it outlines what groups of controls are available and explains how to lookup which knob to use for a particular effect.

The controls available for tuning are described in a separate document, the Program reference manual. That manual lists all different variables, functions and options available in LilyPond. It is written as a HTML document, which is available on-line, but is also included with the LilyPond documentation package.

There are four areas where the default settings may be changed:

- Automatic notation: changing the automatic creation of notation elements. For example, changing the beaming rules.
- Output: changing the appearance of individual objects. For example, changing stem directions or the location of subscripts.
- Context: changing aspects of the translation from music events to notation. For example, giving each staff a separate time signature.
- Page layout: changing the appearance of the spacing, line breaks, and page dimensions. These modifications are discussed in Chapter 10 [Non-musical notation], page 234 and Chapter 11 [Spacing issues], page 246.

Internally, LilyPond uses Scheme (a LISP dialect) to provide infrastructure. Overriding layout decisions in effect accesses the program internals, which requires Scheme input. Scheme elements are introduced in a `.ly` file with the hash mark `#`.[1]

9.1 Automatic notation

This section describes how to change the way that accidentals and beams are automatically displayed.

9.1.1 Automatic accidentals

Common rules for typesetting accidentals have been placed in a function. This function is called as follows

```
#(set-accidental-style 'STYLE #('CONTEXT#))
```

The function can take two arguments: the name of the accidental style, and an optional argument that denotes the context that should be changed. If no context name is supplied, `Staff` is the default, but you may wish to apply the accidental style to a single `Voice` instead.

The following accidental styles are supported

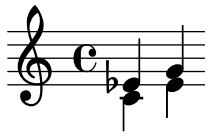
`default` This is the default typesetting behavior. It corresponds to 18th century common practice: Accidentals are remembered to the end of the measure in which they occur and only on their own octave.

`voice` The normal behavior is to remember the accidentals on Staff-level. This variable, however, typesets accidentals individually for each voice. Apart from that, the rule is similar to `default`.

As a result, accidentals from one voice do not get canceled in other voices, which is often an unwanted result

[1] Appendix B [Scheme tutorial], page 310 contains a short tutorial on entering numbers, lists, strings, and symbols in Scheme.

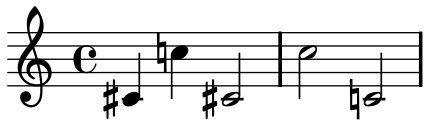
```
\new Staff <<
  #(set-accidental-style 'voice)
  <<
    { es g } \\
    { c, e }
>> >>
```



The `voice` option should be used if the voices are to be read solely by individual musicians. If the staff is to be used by one musician (e.g., a conductor) then `modern` or `modern-cautionary` should be used instead.

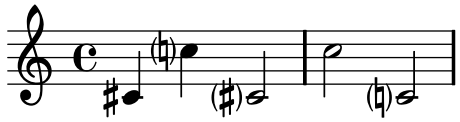
`modern` This rule corresponds to the common practice in the 20th century. This rule prints the same accidentals as `default`, but temporary accidentals also are canceled in other octaves. Furthermore, in the same octave, they also get canceled in the following measure

```
#(set-accidental-style 'modern)
cis' c'' cis'2 | c'' c'
```



`modern-cautionary`
This rule is similar to `modern`, but the "extra" accidentals (the ones not typeset by `default`) are typeset as cautionary accidentals. They are printed in reduced size or with parentheses

```
#(set-accidental-style 'modern-cautionary)
cis' c'' cis'2 | c'' c'
```



`modern-voice`
This rule is used for multivoice accidentals to be read both by musicians playing one voice and musicians playing all voices. Accidentals are typeset for each voice, but they *are* canceled across voices in the same `Staff`.

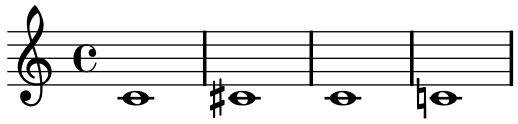
`modern-voice-cautionary`
This rule is the same as `modern-voice`, but with the extra accidentals (the ones not typeset by `voice`) typeset as cautionaries. Even though all accidentals typeset by `default` *are* typeset by this variable, some of them are typeset as cautionaries.

`piano` This rule reflects 20th century practice for piano notation. Very similar to `modern` but accidentals also get canceled across the staves in the same `GrandStaff` or `PianoStaff`.

`piano-cautionary`
: Same as `#(set-accidental-style 'piano)` but with the extra accidentals typeset as cautionaries.

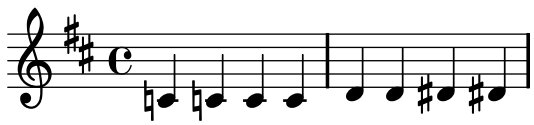
`no-reset`
: This is the same as `default` but with accidentals lasting "forever" and not only until the next measure

```
#(set-accidental-style 'no-reset)
c1 cis cis c
```



`forget`
: This is sort of the opposite of `no-reset`: Accidentals are not remembered at all – and hence all accidentals are typeset relative to the key signature, regardless of what was before in the music

```
#(set-accidental-style 'forget)
\key d\major c4 c cis cis d d dis dis
```



See also

Program reference: `Accidental_engraver`, `Accidental`, and `AccidentalPlacement`.

Bugs

Simultaneous notes are considered to be entered in sequential mode. This means that in a chord the accidentals are typeset as if the notes in the chord happen one at a time, in the order in which they appear in the input file. This is a problem when accidentals in a chord depend on each other, which does not happen for the default accidental style. The problem can be solved by manually inserting `!` and `?` for the problematic notes.

9.1.2 Setting automatic beam behavior

In normal time signatures, automatic beams can start on any note but can only end in a few positions within the measure: beams can end on a beat, or at durations specified by the properties in `autoBeamSettings`. The properties in `autoBeamSettings` consist of a list of rules for where beams can begin and end. The default `autoBeamSettings` rules are defined in '`scm/auto-beam.scm`'.

In order to add a rule to the list, use

```
#(override-auto-beam-setting '(be p q n m) a b [context])
```

- `be` is either "begin" or "end".
- `p/q` is the duration of the note for which you want to add a rule. A beam is considered to have the duration of its shortest note. Set `p` and `q` to `'*'` to have this apply to any beam.
- `n/m` is the time signature to which this rule should apply. Set `n` and `m` to `'*'` to have this apply in any time signature.
- `a/b` is the position in the bar at which the beam should begin/end.

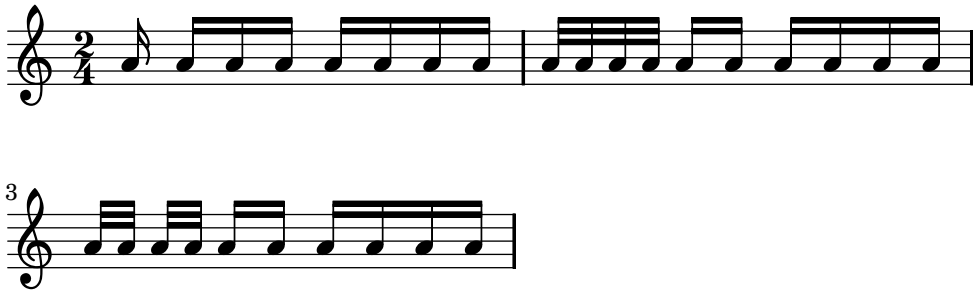
- `context` is optional, and it specifies the context at which the change should be made. The default is `'Voice`. `#(score-override-auto-beam-setting '(A B C D) E F)` is equivalent to `#(override-auto-beam-setting '(A B C D) E F 'Score)`.

For example, if automatic beams should always end on the first quarter note, use

```
#(override-auto-beam-setting '(end * * * *) 1 4)
```

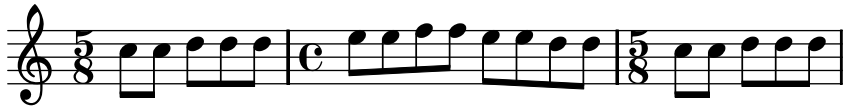
You can force the beam settings to only take effect on beams whose shortest note is a certain duration

```
\time 2/4
#(override-auto-beam-setting '(end 1 16 * *) 1 16)
a16 a a a a a a |
a32 a a a a16 a a a a a |
#(override-auto-beam-setting '(end 1 32 * *) 1 16)
a32 a a a a16 a a a a a |
```



You can force the beam settings to only take effect in certain time signatures

```
\time 5/8
#(override-auto-beam-setting '(end * * 5 8) 2 8)
c8 c d d d
\time 4/4
e8 e f f e e d d
\time 5/8
c8 c d d d
```



You can also remove a previously set beam-ending rule by using

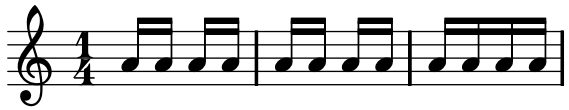
```
#(revert-auto-beam-setting '(be p q n m) a b [context])
```

be, p, q, n, m, a, b and context are the same as above. Note that the default rules are specified in '`scm/auto-beam.scm`', so you can revert rules that you did not explicitly create.

```
\time 4/4
a16 a a a a a a a a a a a a a a a a
#(revert-auto-beam-setting '(end 1 16 4 4) 1 4)
a16 a a a a a a a a a a a a a a a a
```

The rule in a revert-auto-beam-setting statement must exactly match the original rule. That is, no wildcard expansion is taken into account.

```
\time 1/4
#(override-auto-beam-setting '(end 1 16 1 4) 1 8)
a16 a a a
#(revert-auto-beam-setting '(end 1 16 * *) 1 8) % this won't revert it!
a a a a
#(revert-auto-beam-setting '(end 1 16 1 4) 1 8) % this will
a a a a
```



If automatic beams should end on every quarter in 5/4 time, specify all endings

```
#(override-auto-beam-setting '(end * * * *) 1 4 'Staff)
#(override-auto-beam-setting '(end * * * *) 1 2 'Staff)
#(override-auto-beam-setting '(end * * * *) 3 4 'Staff)
#(override-auto-beam-setting '(end * * * *) 5 4 'Staff)
...
```

The same syntax can be used to specify beam starting points. In this example, automatic beams can only end on a dotted quarter note

```
#(override-auto-beam-setting '(end * * * *) 3 8)
#(override-auto-beam-setting '(end * * * *) 1 2)
#(override-auto-beam-setting '(end * * * *) 7 8)
```

In 4/4 time signature, this means that automatic beams could end only on 3/8 and on the fourth beat of the measure (after 3/4, that is 2 times 3/8, has passed within the measure).

If any unexpected beam behaviour occurs, check the default automatic beam settings in '`scm/auto-beam.scm`' for possible interference, because the beam endings defined there will still apply on top of your own overrides. Any unwanted endings in the default vales must be reverted for your time signature(s).

For example, to typeset `(3 4 3 2)`-beam endings in 12/8, begin with

```
%%% revert default values in scm/auto-beam.scm regarding 12/8 time
#(revert-auto-beam-setting '(end * * 12 8) 3 8)
#(revert-auto-beam-setting '(end * * 12 8) 3 4)
#(revert-auto-beam-setting '(end * * 12 8) 9 8)

%%% your new values
#(override-auto-beam-setting '(end 1 8 12 8) 3 8)
#(override-auto-beam-setting '(end 1 8 12 8) 7 8)
#(override-auto-beam-setting '(end 1 8 12 8) 10 8)
```

If beams are used to indicate melismata in songs, then automatic beaming should be switched off with `\autoBeamOff`.

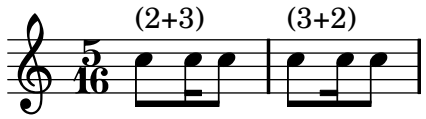
Predefined commands

`\autoBeamOff`, `\autoBeamOn`.

Commonly tweaked properties

Beaming patterns may be altered with the `beatGrouping` property,

```
\time 5/16
\set beatGrouping = #'(2 3)
c8[^"(2+3)" c16 c8]
\set beatGrouping = #'(3 2)
c8[^"(3+2)" c16 c8]
```



Bugs

If a score ends while an automatic beam has not been ended and is still accepting notes, this last beam will not be typeset at all. The same holds polyphonic voices, entered with `<< ... \\ ... >>`. If a polyphonic voice ends while an automatic beam is still accepting notes, it is not typeset.

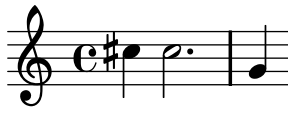
9.2 Interpretation contexts

This section describes what contexts are, and how to modify them.

9.2.1 Contexts explained

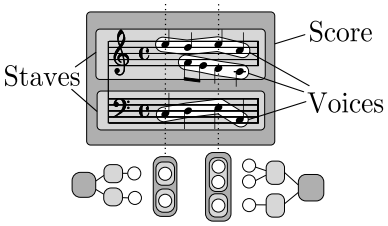
When music is printed, a lot of notational elements must be added to the output. For example, compare the input and output of the following example:

```
cis4 cis2. g4
```



The input is rather sparse, but in the output, bar lines, accidentals, clef, and time signature are added. LilyPond *interprets* the input. During this step, the musical information is inspected in time order, similar to reading a score from left to right. While reading the input, the program remembers where measure boundaries are, and which pitches require explicit accidentals. This information can be presented on several levels. For example, the effect of an accidental is limited to a single staff, while a bar line must be synchronized across the entire score.

Within LilyPond, these rules and bits of information are grouped in *Contexts*. Some examples of contexts are `Voice`, `Staff`, and `Score`. They are hierarchical, for example: a `Staff` can contain many `Voices`, and a `Score` can contain many `Staff` contexts.



Each context has the responsibility for enforcing some notation rules, creating some notation objects and maintaining the associated properties. For example, the `Voice` context may introduce an accidental and then the `Staff` context maintains the rule to show or suppress the accidental for the remainder of the measure. The synchronization of bar lines is handled at `Score` context.

However, in some music we may not want the bar lines to be synchronized – consider a polymetric score in 4/4 and 3/4 time. In such cases, we must modify the default settings of the `Score` and `Staff` contexts.

For very simple scores, contexts are created implicitly, and you need not be aware of them. For larger pieces, such as anything with more than one staff, they must be created explicitly to make sure that you get as many staves as you need, and that they are in the correct order. For typesetting pieces with specialized notation, it can be useful to modify existing or to define new contexts.

A complete description of all available contexts is in the program reference, see Translation ⇒ Context.

9.2.2 Creating contexts

For scores with only one voice and one staff, contexts are created automatically. For more complex scores, it is necessary to create them by hand. There are three commands that do this.

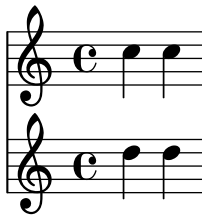
- The easiest command is `\new`, and it also the quickest to type. It is prepended to a music expression, for example

  ```
  \new type music expression
  ```

  where *type* is a context name (like `Staff` or `Voice`). This command creates a new context, and starts interpreting the *music expression* with that.

  A practical application of `\new` is a score with many staves. Each part that should be on its own staff, is preceded with `\new Staff`.

  ```
  <<
    \new Staff { c4 c }
    \new Staff { d4 d }
  >>
  ```

  The `\new` command may also give a name to the context,

  ```
  \new type = id music
  ```

  However, this user specified name is only used if there is no other context already earlier with the same name.

- Like `\new`, the `\context` command also directs a music expression to a context object, but gives the context an explicit name. The syntax is

  ```
  \context type = id music
  ```

  This form will search for an existing context of type *type* called *id*. If that context does not exist yet, a new context with the specified name is created. This is useful if the context is referred to later on. For example, when setting lyrics the melody is in a named context

```
\context Voice = "tenor" music
```

so the texts can be properly aligned to its notes,

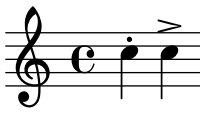
```
\new Lyrics \lyricsto "tenor" lyrics
```

Another possible use of named contexts is funneling two different music expressions into one context. In the following example, articulations and notes are entered separately,

```
music = { c4 c4 }
arts = { s4-. s4-> }
```

They are combined by sending both to the same `Voice` context,

```
<<
  \new Staff \context Voice = "A" \music
  \context Voice = "A" \arts
>>
```



With this mechanism, it is possible to define an Urtext (original edition), with the option to put several distinct articulations on the same notes.

- The third command for creating contexts is

  ```
  \context type music
  ```

  This is similar to `\context` with `= id`, but matches any context of type *type*, regardless of its given name.

  This variant is used with music expressions that can be interpreted at several levels. For example, the `\applyOutput` command (see Section 12.5.2 [Running a function on all layout objects], page 284). Without an explicit `\context`, it is usually applied to `Voice`

  ```
  \applyOutput #'context #function   % apply to Voice
  ```

  To have it interpreted at the `Score` or `Staff` level use these forms

  ```
  \applyOutput #'Score #function
  \applyOutput #'Staff #function
  ```

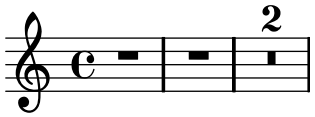
9.2.3 Changing context properties on the fly

Each context can have different *properties*, variables contained in that context. They can be changed during the interpretation step. This is achieved by inserting the `\set` command in the music,

```
\set context.prop = #value
```

For example,

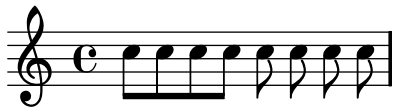
```
R1*2
\set Score.skipBars = ##t
R1*2
```



This command skips measures that have no notes. The result is that multi-rests are condensed. The value assigned is a Scheme object. In this case, it is `#t`, the boolean True value.

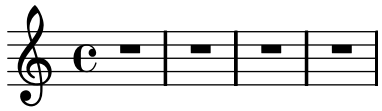
If the *context* argument is left out, then the current bottom-most context (typically `ChordNames`, `Voice`, or `Lyrics`) is used. In this example,

```
c8 c c c
\set autoBeaming = ##f
c8 c c c
```



the *context* argument to `\set` is left out, so automatic beaming is switched off in the current `Voice`. Note that the bottom-most context does not always contain the property that you wish to change – for example, attempting to set the `skipBars` property (of the bottom-most context, in this case `Voice`) will have no effect.

```
R1*2
\set skipBars = ##t
R1*2
```



Contexts are hierarchical, so if a bigger context was specified, for example `Staff`, then the change would also apply to all `Voices` in the current stave. The change is applied 'on-the-fly', during the music, so that the setting only affects the second group of eighth notes.

There is also an `\unset` command,

```
\unset context.prop
```

which removes the definition of *prop*. This command removes the definition only if it is set in *context*, so

```
\set Staff.autoBeaming = ##f
```

introduces a property setting at `Staff` level. The setting also applies to the current `Voice`. However,

```
\unset Voice.autoBeaming
```

does not have any effect. To cancel this setting, the `\unset` must be specified on the same level as the original `\set`. In other words, undoing the effect of `Staff.autoBeaming = ##f` requires

```
\unset Staff.autoBeaming
```

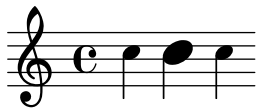
Like `\set`, the *context* argument does not have to be specified for a bottom context, so the two statements

```
\set Voice.autoBeaming = ##t
\set autoBeaming = ##t
```

are equivalent.

Settings that should only apply to a single time-step can be entered with `\once`, for example in

```
c4
\once \set fontSize = #4.7
c4
c4
```



the property `fontSize` is unset automatically after the second note.

A full description of all available context properties is in the program reference, see Translation ⇒ Tunable context properties.

9.2.4 Modifying context plug-ins

Notation contexts (like `Score` and `Staff`) not only store properties, they also contain plug-ins called "engravers" that create notation elements. For example, the `Voice` context contains a `Note_head_engraver` and the `Staff` context contains a `Key_signature_engraver`.

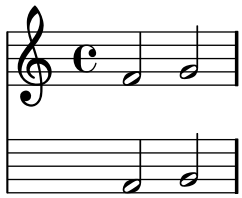
For a full a description of each plug-in, see Program reference ⇒ Translation ⇒ Engravers. Every context described in Program reference ⇒ Translation ⇒ Context. lists the engravers used for that context.

It can be useful to shuffle around these plug-ins. This is done by starting a new context with `\new` or `\context`, and modifying it,

```
\new context \with {
  \consists ...
  \consists ...
  \remove ...
  \remove ...
  etc.
}
{
  ..music..
}
```

where the ... should be the name of an engraver. Here is a simple example which removes `Time_signature_engraver` and `Clef_engraver` from a `Staff` context,

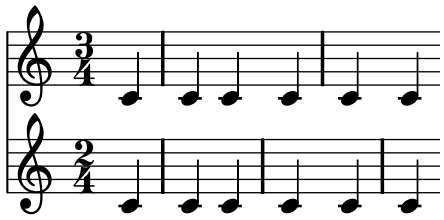
```
<<
  \new Staff {
    f2 g
  }
  \new Staff \with {
     \remove "Time_signature_engraver"
     \remove "Clef_engraver"
  } {
    f2 g2
  }
>>
```



In the second staff there are no time signature or clef symbols. This is a rather crude method of making objects disappear since it will affect the entire staff. This method also influences the spacing, which may or may not be desirable. A more sophisticated method of blanking objects is shown in Section 5.3 [Common tweaks], page 54.

The next example shows a practical application. Bar lines and time signatures are normally synchronized across the score. This is done by the `Timing_translator` and `Default_bar_line_engraver`. This plug-in keeps an administration of time signature, location within the measure, etc. By moving thes engraver from `Score` to `Staff` context, we can have a score where each staff has its own time signature.

```
\new Score \with {
  \remove "Timing_translator"
  \remove "Default_bar_line_engraver"
} <<
  \new Staff \with {
    \consists "Timing_translator"
    \consists "Default_bar_line_engraver"
  } {
      \time 3/4
      c4 c c c c c
  }
  \new Staff \with {
    \consists "Timing_translator"
    \consists "Default_bar_line_engraver"
  } {
       \time 2/4
       c4 c c c c c
  }
>>
```



9.2.5 Layout tunings within contexts

Each context is responsible for creating certain types of graphical objects. The settings used for printing these objects are also stored by context. By changing these settings, the appearance of objects can be altered.

The syntax for this is

```
\override context.name #'property = #value
```

Here *name* is the name of a graphical object, like `Stem` or `NoteHead`, and *property* is an internal variable of the formatting system ('grob property' or 'layout property'). The latter is a symbol, so it must be quoted. The subsection Section 9.3.1 [Constructing a tweak], page 228

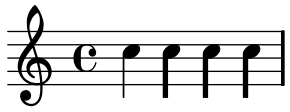
explains what to fill in for *name*, *property*, and *value*. Here we only discuss the functionality of this command.

The command

```
\override Staff.Stem #'thickness = #4.0
```

makes stems thicker (the default is 1.3, with staff line thickness as a unit). Since the command specifies `Staff` as context, it only applies to the current staff. Other staves will keep their normal appearance. Here we see the command in action:

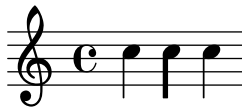
```
c4
\override Staff.Stem #'thickness = #4.0
c4
c4
c4
```



The `\override` command changes the definition of the `Stem` within the current `Staff`. After the command is interpreted all stems are thickened.

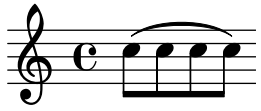
Analogous to `\set`, the *context* argument may be left out, causing the default context `Voice` to be used. Adding `\once` applies the change during one timestep only.

```
c4
\once \override Stem #'thickness = #4.0
c4
c4
```



The `\override` must be done before the object is started. Therefore, when altering *Spanner* objects such as slurs or beams, the `\override` command must be executed at the moment when the object is created. In this example,

```
\override Slur #'thickness = #3.0
c8[( c
\override Beam #'thickness = #0.6
c8 c])
```



the slur is fatter but the beam is not. This is because the command for `Beam` comes after the Beam is started, so it has no effect.

Analogous to `\unset`, the `\revert` command for a context undoes an `\override` command; like with `\unset`, it only affects settings that were made in the same context. In other words, the `\revert` in the next example does not do anything.

```
\override Voice.Stem #'thickness = #4.0
\revert Staff.Stem #'thickness
```

Some tweakable options are called "subproperties" and reside inside properties. To tweak those, use commands of the form

```
\override context.name #'property #'subproperty = #value
```

such as

```
\override Stem #'details #'beamed-lengths = #'(4 4 3)
```

See also

Internals: `OverrideProperty`, `RevertProperty`, `PropertySet`, `Backend`, and `All layout objects`.

Bugs

The back-end is not very strict in type-checking object properties. Cyclic references in Scheme values for properties can cause hangs or crashes, or both.

9.2.6 Changing context default settings

The adjustments of the previous subsections (Section 9.2.3 [Changing context properties on the fly], page 220, Section 9.2.4 [Modifying context plug-ins], page 222, and Section 9.2.5 [Layout tunings within contexts], page 223) can also be entered separately from the music in the `\layout` block,

```
\layout {
  ...
  \context {
    \Staff

    \set fontSize = #-2
    \override Stem #'thickness = #4.0
    \remove "Time_signature_engraver"
  }
}
```

The `\Staff` command brings in the existing definition of the staff context so that it can be modified.

The statements

```
\set fontSize = #-2
\override Stem #'thickness = #4.0
\remove "Time_signature_engraver"
```

affect all staves in the score. Other contexts can be modified analogously.

The `\set` keyword is optional within the `\layout` block, so

```
\context {
  ...
  fontSize = #-2
}
```

will also work.

Bugs

It is not possible to collect context changes in a variable and apply them to a `\context` definition by referring to that variable.

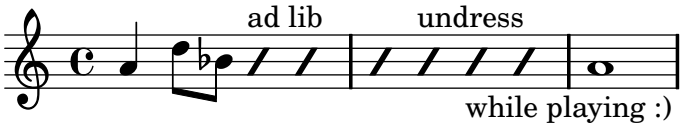
The `\RemoveEmptyStaffContext` will overwrite your current `\Staff` settings. If you wish to change the defaults for a staff which uses `\RemoveEmptyStaffContext`, you must do so after calling `\RemoveemptyStaffContext`, ie

```
\layout {
  \context {
    \RemoveEmptyStaffContext

    \override Stem #'thickness = #4.0
  }
}
```

9.2.7 Defining new contexts

Specific contexts, like `Staff` and `Voice`, are made of simple building blocks. It is possible to create new types of contexts with different combinations of engraver plug-ins.

The next example shows how to build a different type of `Voice` context from scratch. It will be similar to `Voice`, but only prints centered slash noteheads. It can be used to indicate improvisation in jazz pieces,



These settings are defined within a `\context` block inside a `\layout` block,

```
\layout {
  \context {
    ...
  }
}
```

In the following discussion, the example input shown should go in place of the ... in the previous fragment.

First it is necessary to define a name for the new context:

```
\name ImproVoice
```

Since it is similar to the `Voice`, we want commands that work on (existing) `Voices` to remain working. This is achieved by giving the new context an alias `Voice`,

```
\alias Voice
```

The context will print notes and instructive texts, so we need to add the engravers which provide this functionality,

```
\consists Note_heads_engraver
\consists Text_engraver
```

but we only need this on the center line,

```
\consists Pitch_squash_engraver
squashedPosition = #0
```

The `Pitch_squash_engraver` modifies note heads (created by `Note_heads_engraver`) and sets their vertical position to the value of `squashedPosition`, in this case `0`, the center line.

The notes look like a slash, and have no stem,

```
\override NoteHead #'style = #'slash
\override Stem #'transparent = ##t
```

All these plug-ins have to cooperate, and this is achieved with a special plug-in, which must be marked with the keyword `\type`. This should always be `Engraver_group`,

```
\type "Engraver_group"
```

Put together, we get

```
\context {
  \name ImproVoice
  \type "Engraver_group"
  \consists "Note_heads_engraver"
  \consists "Text_engraver"
  \consists Pitch_squash_engraver
  squashedPosition = #0
  \override NoteHead #'style = #'slash
  \override Stem #'transparent = ##t
  \alias Voice
}
```

Contexts form hierarchies. We want to hang the `ImproVoice` under `Staff`, just like normal `Voices`. Therefore, we modify the `Staff` definition with the `\accepts` command,

```
\context {
  \Staff
  \accepts ImproVoice
}
```

The opposite of `\accepts` is `\denies`, which is sometimes needed when reusing existing context definitions.

Putting both into a `\layout` block, like

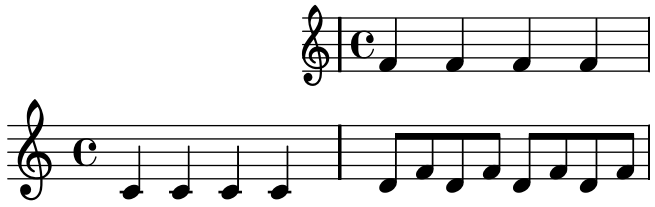
```
\layout {
  \context {
    \name ImproVoice
    ...
  }
  \context {
    \Staff
    \accepts "ImproVoice"
  }
}
```

Then the output at the start of this subsection can be entered as

```
\relative c'' {
  a4 d8 bes8
  \new ImproVoice {
    c4^"ad lib" c
    c4 c^"undress"
    c c_"while playing :)"
  }
  a1
}
```

9.2.8 Aligning contexts

New contexts may be aligned above or below exisiting contexts. This could be useful in setting up a vocal staff (Section D.4 [Vocal ensembles], page 329) and in ossia,



9.3 The \override command

In the previous section, we have already touched on a command that changes layout details: the `\override` command. In this section, we will look in more detail at how to use the command in practice. The general syntax of this command is:

```
\override context.layout_object #'layout_property = #value
```

This will set the *layout_property* of the specified *layout_object*, which is a member of the *context*, to the *value*.

9.3.1 Constructing a tweak

Commands which change output generally look like

```
\override Voice.Stem #'thickness = #3.0
```

To construct this tweak we must determine these bits of information:

- the context: here `Voice`.
- the layout object: here `Stem`.
- the layout property: here `thickness`.
- a sensible value: here `3.0`.

Some tweakable options are called "subproperties" and reside inside properties. To tweak those, use commands in the form

```
\override Stem #'details #'beamed-lengths = #'(4 4 3)
```

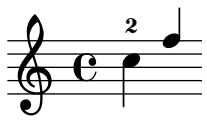
For many properties, regardless of the data type of the property, setting the property to false (`##f`) will result in turning it off, causing Lilypond to ignore that property entirely. This is particularly useful for turning off grob properties which may otherwise be causing problems.

We demonstrate how to glean this information from the notation manual and the program reference.

9.3.2 Navigating the program reference

Suppose we want to move the fingering indication in the fragment below:

```
c-2
\stemUp
f
```



If you visit the documentation on fingering instructions (in Section 6.6.2 [Fingering instructions], page 96), you will notice:

See also

Program reference: `Fingering`.

The programmer's reference is available as an HTML document. It is highly recommended that you read it in HTML form, either online or by downloading the HTML documentation. This section will be much more difficult to understand if you are using the PDF manual.

Follow the link to `Fingering`. At the top of the page, you will see

> Fingering objects are created by: `Fingering_engraver` and `New_fingering_engraver`.

By following related links inside the program reference, we can follow the flow of information within the program:

- `Fingering`: `Fingering` objects are created by: `Fingering_engraver`
- `Fingering_engraver`: Music types accepted: `fingering-event`
- `fingering-event`: Music event type `fingering-event` is in Music expressions named `FingerEvent`

This path goes against the flow of information in the program: it starts from the output, and ends at the input event. You could also start at an input event, and read with the flow of information, eventually ending up at the output object(s).

The program reference can also be browsed like a normal document. It contains chapters on `Music definitions` on `Translation`, and the `Backend`. Every chapter lists all the definitions used and all properties that may be tuned.

9.3.3 Layout interfaces

The HTML page that we found in the previous section describes the layout object called `Fingering`. Such an object is a symbol within the score. It has properties that store numbers (like thicknesses and directions), but also pointers to related objects. A layout object is also called a *Grob*, which is short for Graphical Object. For more details about Grobs, see `grob-interface`.

The page for `Fingering` lists the definitions for the `Fingering` object. For example, the page says

> `padding` (dimension, in staff space):
>
> `0.5`

which means that the number will be kept at a distance of at least 0.5 of the note head.

Each layout object may have several functions as a notational or typographical element. For example, the Fingering object has the following aspects

- Its size is independent of the horizontal spacing, unlike slurs or beams.
- It is a piece of text. Granted, it is usually a very short text.
- That piece of text is typeset with a font, unlike slurs or beams.
- Horizontally, the center of the symbol should be aligned to the center of the notehead.
- Vertically, the symbol is placed next to the note and the staff.
- The vertical position is also coordinated with other superscript and subscript symbols.

Each of these aspects is captured in so-called *interfaces*, which are listed on the `Fingering` page at the bottom

> This object supports the following interfaces: `item-interface`, `self-alignment-interface`, `side-position-interface`, `text-interface`, `text-script-interface`, `font-interface`, `finger-interface`, and `grob-interface`.

Clicking any of the links will take you to the page of the respective object interface. Each interface has a number of properties. Some of them are not user-serviceable ("Internal properties"), but others can be modified.

We have been talking of *the* `Fingering` object, but actually it does not amount to much. The initialization file (see Section 5.4 [Default files], page 55) '`scm/define-grobs.scm`' shows the soul of the 'object',

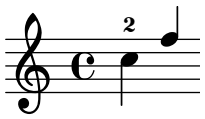
```
(Fingering
  . ((padding . 0.5)
     (avoid-slur . around)
     (slur-padding . 0.2)
     (staff-padding . 0.5)
     (self-alignment-X . 0)
     (self-alignment-Y . 0)
     (script-priority . 100)
     (stencil . ,ly:text-interface::print)
     (direction . ,ly:script-interface::calc-direction)
     (font-encoding . fetaNumber)
     (font-size . -5)  ; don't overlap when next to heads.
     (meta . ((class . Item)
     (interfaces . (finger-interface
                    font-interface
                    text-script-interface
                    text-interface
                    side-position-interface
                    self-alignment-interface
                    item-interface))))))
```

As you can see, the `Fingering` object is nothing more than a bunch of variable settings, and the webpage in the Program Reference is directly generated from this definition.

9.3.4 Determining the grob property

Recall that we wanted to change the position of the **2** in

```
c-2
\stemUp
f
```



Since the **2** is vertically positioned next to its note, we have to meddle with the interface associated with this positioning. This is done using `side-position-interface`. The page for this interface says

`side-position-interface`

Position a victim object (this one) next to other objects (the support). The property `direction` signifies where to put the victim object relative to the support (left or right, up or down?)

Below this description, the variable `padding` is described as

`padding` (dimension, in staff space)

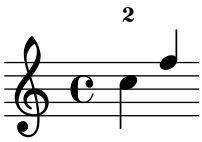
Add this much extra space between objects that are next to each other.

By increasing the value of `padding`, we can move the fingering away from the notehead. The following command inserts 3 staff spaces of white between the note and the fingering:

```
\once \override Voice.Fingering #'padding = #3
```

Inserting this command before the Fingering object is created, i.e., before `c2`, yields the following result:

```
\once \override Voice.Fingering #'padding = #3
c-2
\stemUp
f
```



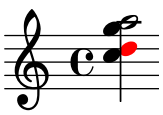
In this case, the context for this tweak is `Voice`. This fact can also be deduced from the program reference, for the page for the `Fingering_engraver` plug-in says

Fingering_engraver is part of contexts: ... `Voice`

9.3.5 Objects connected to the input

In some cases, it is possible to take a short-cut for tuning graphical objects. For objects that result directly from a piece of the input, you can use the `\tweak` function, for example

```
<
  c
  \tweak #'color #red d
  g
  \tweak #'duration-log #1  a
>4-\tweak #'padding #10 -.
          .
```



As you can see, properties are set directly in the objects directly, without mentioning the grob name or context where this should be applied.

This technique only works for objects that are directly connected to an `event` from the input, for example

- note heads, caused by chord-pitch (i.e., notes inside a chord).
- articulation signs, caused by articulation instructions.

It notably does not work for stems and accidentals (these are caused by note heads, not by music events) or clefs (these are not caused by music inputs, but rather by the change of a property value).

There are very few objects which are *directly* connected to output. A normal note (like `c4`) is not directly connected to output, so

```
\tweak #'color #red c4
```

will not change color. See Section 12.3.1 [Displaying music expressions], page 276 for details.

9.3.6 \set vs. \override

We have seen two methods of changing properties: `\set` and `\override`. There are actually two different kinds of properties.

Contexts can have properties, which are usually named in `studlyCaps`. They mostly control the translation from music to notatino, eg. `localKeySignature` (for determining whether to print accidentals), `measurePosition` (for determining when to print a barline). Context properties can change value over time while interpreting a piece of music; `measurePosition` is an obvious example of this. Context properties are modified with `\set`.

There is a special type of context property: the element description. These properties are named in `StudlyCaps` (starting with capital letters). They contain the "default settings" for said graphical object as an association list. See '`scm/define-grobs.scm`' to see what kind of settings there are. Element descriptions may be modified with `\override`.

`\override` is actually a shorthand;

```
\override context.name #'property = #value
```

is more or less equivalent to

```
\set context.name #'property = #(cons (cons 'property value) <previous value of context)
```

The value of `context` (the alist) is used to initalize the properties of individual grobs. Grobs also have properties, named in Scheme style, with `dashed-words`. The values of grob properties change during the formatting process: formatting basically amounts to computing properties using callback functions.

`fontSize` is a special property: it is equivalent to entering `\override ... #'font-size` for all pertinent objects. Since this is a common change, the special property (modified with `\set`) was created.

9.3.7 Difficult tweaks

There are a few classes of difficult adjustments.

- One type of difficult adjustment is the appearance of spanner objects, such as slur and tie. Initially, only one of these objects is created, and they can be adjusted with the normal mechanism. However, in some cases the spanners cross line breaks. If this happens, these objects are cloned. A separate object is created for every system that it is in. These are clones of the original object and inherit all properties, including `\overrides`.

  In other words, an `\override` always affects all pieces of a broken spanner. To change only one part of a spanner at a line break, it is necessary to hook into the formatting process. The `after-line-breaking` callback contains the Scheme procedure that is called after the line breaks have been determined, and layout objects have been split over different systems.

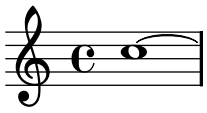
  In the following example, we define a procedure `my-callback`. This procedure

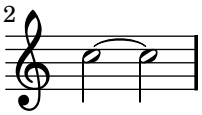
  - determines if we have been split across line breaks
  - if yes, retrieves all the split objects
  - checks if we are the last of the split objects
  - if yes, it sets `extra-offset`.

  This procedure is installed into `Tie`, so the last part of the broken tie is translated up.

```
#(define (my-callback grob)
  (let* (
         ; have we been split?
         (orig (ly:grob-original grob))
```

```
          ; if yes, get the split pieces (our siblings)
          (siblings (if (ly:grob? orig)
                        (ly:spanner-broken-into orig) '() )))

   (if (and (>= (length siblings) 2)
            (eq? (car (last-pair siblings)) grob))
     (ly:grob-set-property! grob 'extra-offset '(-2 . 5)))))

\relative c'' {
  \override Tie #'after-line-breaking =
  #my-callback
  c1 ~ \break c2 ~ c
}
```





When applying this trick, the new `after-line-breaking` callback should also call the old one `after-line-breaking`, if there is one. For example, if using this with `Hairpin`, `ly:hairpin::after-line-breaking` should also be called.

- Some objects cannot be changed with `\override` for technical reasons. Examples of those are `NonMusicalPaperColumn` and `PaperColumn`. They can be changed with the `\outputProperty` function, which works similar to `\once \override`, but uses a different syntax,

```
\outputProperty
#"Score.NonMusicalPaperColumn"  % Grob name
#'line-break-system-details     % Property name
#'((next-padding . 20))         % Value
```

10 Non-musical notation

This section deals with general lilypond issues, rather than specific notation.

10.1 Input files

The main format of input for LilyPond are text files. By convention, these files end with "`.ly`".

10.1.1 File structure (introduction)

A basic example of a lilypond input file is

```
\version "2.10.10"
\score {
  { }     % this is a single music expression;
            % all the music goes in here.
  \header { }
  \layout { }
  \midi { }
}
```

There are many variations of this basic pattern, but this example serves as a useful starting place.

The major part of this manual is concerned with entering various forms of music in LilyPond. However, many music expressions are not valid input on their own, for example, a `.ly` file containing only a note

```
c'4
```

will result in a parsing error. Instead, music should be inside other expressions, which may be put in a file by themselves. Such expressions are called toplevel expressions; see Section 10.1.2 [File structure], page 234 for a list of all such expressions.

10.1.2 File structure

A `.ly` file contains any number of toplevel expressions, where a toplevel expression is one of the following

- An output definition, such as `\paper`, `\midi`, and `\layout`. Such a definition at the toplevel changes the default settings for the block entered.
- A direct scheme expression, such as `#(set-default-paper-size "a7" 'landscape)` or `#(ly:set-option 'point-and-click #f)`.
- A `\header` block. This sets the global header block. This is the block containing the definitions for book-wide settings, like composer, title, etc.
- A `\score` block. This score will be collected with other toplevel scores, and combined as a single `\book`.

  This behavior can be changed by setting the variable `toplevel-score-handler` at toplevel. The default handler is defined in the init file '`scm/lily.scm`'.

  The `\score` must begin with a music expression, and may contain only one music expression.
- A `\book` block logically combines multiple movements (i.e., multiple `\score` blocks) in one document. If there are a number of `\scores`, one output file will be created for each `\book` block, in which all corresponding movements are concatenated. The only reason to explicitly specify `\book` blocks in a `.ly` file is if you wish multiple output files from a single input file. One exception is within lilypond-book documents, where you explicitly have to add a `\book` block if you want more than a single `\score` or `\markup` in the same example.

  This behavior can be changed by setting the variable `toplevel-book-handler` at toplevel. The default handler is defined in the init file '`scm/lily.scm`'.

- A compound music expression, such as

```
{ c'4 d' e'2 }
```

  This will add the piece in a `\score` and format it in a single book together with all other toplevel `\score`s and music expressions. In other words, a file containing only the above music expression will be translated into

```
\book {
  \score {
    \new Staff {
      \new Voice {
        { c'4 d' e'2 }
      }
    }
  }
\layout { }
\header { }
}
```

  This behavior can be changed by setting the variable `toplevel-music-handler` at toplevel. The default handler is defined in the init file '`scm/lily.scm`'.

- A markup text, a verse for example

```
\markup {
   2.  The first line verse two.
}
```

  Markup texts are rendered above, between or below the scores or music expressions, wherever they appear.

- An identifier, such as

```
foo = { c4 d e d }
```

  This can be used later on in the file by entering `\foo`. The name of an identifier should have alphabetic characters only; no numbers, underscores or dashes.

The following example shows three things that may be entered at toplevel

```
\layout {
  % movements are non-justified by default
  ragged-right = ##t
}

\header {
   title = "Do-re-mi"
}

{ c'4 d' e2 }
```

At any point in a file, any of the following lexical instructions can be entered:

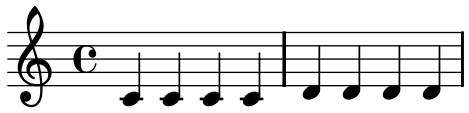
- `\version`
- `\include`
- `\sourcefilename`
- `\sourcefileline`

10.1.3 A single music expression

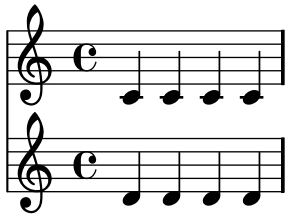
A `\score` must contain a single music expression. However, this music expression may be of any size. Recall that music expressions may be included inside other expressions to form larger

expressions. All of these examples are single music expressions; note the curly braces `{ }` or angle brackets `<< >>` at the beginning and ending of the music.

```
{ c'4 c' c' c' }
{
  { c'4 c' c' c'}
  { d'4 d' d' d'}
}
```



```
<<
  \new Staff { c'4 c' c' c' }
  \new Staff { d'4 d' d' d' }
>>
```



```
{
  \new GrandStaff <<
    \new StaffGroup <<
      \new Staff { \flute }
      \new Staff { \oboe }
    >>
    \new StaffGroup <<
      \new Staff { \violinI }
      \new Staff { \violinII }
    >>
  >>
}
```

10.1.4 Multiple scores in a book

A document may contain multiple pieces of music and texts. Examples of these are an etude book, or an orchestral part with multiple movements. Each movement is entered with a `\score` block,

```
\score {
  ..music..
}
```

and texts are entered with a `\markup` block,

```
\markup {
  ..text..
}
```

All the movements and texts which appear in the same `.ly` file will normally be typeset in the form of a single output file.

```
\score {
  ..
}
\markup {
  ..
}
\score {
  ..
}
```

However, if you want multiple output files from the same `.ly` file, then you can add multiple `\book` blocks, where each such `\book` block will result in a separate output. If you do not specify any `\book` block in the file, LilyPond will implicitly treat the full file as a single `\book` block, see Section 10.1.2 [File structure], page 234. One important exception is within lilypond-book documents, where you explicitly have to add a `\book` block, otherwise only the first `\score` or `\markup` will appear in the output.

The header for each piece of music can be put inside the `\score` block. The `piece` name from the header will be printed before each movement. The title for the entire book can be put inside the `\book`, but if it is not present, the `\header` which is at the top of the file is inserted.

```
\header {
  title = "Eight miniatures"
  composer = "Igor Stravinsky"
}
\score {
  ...
  \header { piece = "Romanze" }
}
\markup {
   ..text of second verse..
}
\markup {
   ..text of third verse..
}
\score {
  ...
  \header { piece = "Menuetto" }
}
```

10.1.5 Extracting fragments of notation

It is possible to quote small fragments of a large score directly from the output. This can be compared to clipping a piece of a paper score with scissors.

This is done by definining the measures that need to be cut out separately. For example, including the following definition

```
\layout {
  clip-regions
  = #(list
      (cons
       (make-rhythmic-location 5 1 2)
       (make-rhythmic-location 7 3 4)))
}
```

will extract a fragment starting halfway the fifth measure, ending in the seventh measure. The meaning of `5 1 2` is: after a 1/2 note in measure 5, and `7 3 4` after 3 quarter notes in measure 7.

More clip regions can be defined by adding more pairs of rhythmic-locations to the list.

In order to use this feature, LilyPond must be invoked with `-dclip-systems`. The clips are output as EPS files, and are converted to PDF and PNG if these formats are switched on as well.

For more information on output formats, see Section 13.1 [Invoking lilypond], page 286.

See also

Examples: '`input/regression//clip-systems.ly`'

10.1.6 Including LilyPond files

A large project may be split up into separate files. To refer to another file, use

```
\include "otherfile.ly"
```

The line `\include "file.ly"` is equivalent to pasting the contents of file.ly into the current file at the place where you have the \include. For example, for a large project you might write separate files for each instrument part and create a "full score" file which brings together the individual instrument files.

The initialization of LilyPond is done in a number of files that are included by default when you start the program, normally transparent to the user. Run lilypond –verbose to see a list of paths and files that Lily finds.

Files placed in directory '`PATH/TO/share/lilypond/VERSION/ly/`' (where VERSION is in the form "2.6.1") are on the path and available to `\include`. Files in the current working directory are available to \include, but a file of the same name in LilyPond's installation takes precedence. Files are available to \include from directories in the search path specified as an option when invoking `lilypond --include=DIR` which adds DIR to the search path.

The `\include` statement can use full path information, but with the Unix convention `"/"` rather than the DOS/Windows `"\"`. For example, if '`stuff.ly`' is located one directory higher than the current working directory, use

```
\include "../stuff.ly"
```

10.1.7 Text encoding

LilyPond uses the Pango library to format multi-lingual texts, and does not perform any input-encoding conversions. This means that any text, be it title, lyric text, or musical instruction containing non-ASCII characters, must be utf-8. The easiest way to enter such text is by using a Unicode-aware editor and saving the file with utf-8 encoding. Most popular modern editors have utf-8 support, for example, vim, Emacs, jEdit, and GEdit do.

To use a Unicode escape sequence, use

```
#(ly:export (ly:wide-char->utf-8 #x2014))
```

See also

'`input/regression/utf-8.ly`'

10.2 Titles and headers

Almost all printed music includes a title and the composer's name; some pieces include a lot more information.

10.2.1 Creating titles

Titles are created for each `\score` block, as well as for the full input file (or `\book` block).

The contents of the titles are taken from the `\header` blocks. The header block for a book supports the following

`dedication`
: The dedicatee of the music, centered at the top of the first page.

`title`
: The title of the music, centered just below the dedication.

`subtitle`
: Subtitle, centered below the title.

`subsubtitle`
: Subsubtitle, centered below the subtitle.

`poet`
: Name of the poet, flush-left below the subtitle.

`composer`
: Name of the composer, flush-right below the subtitle.

`meter`
: Meter string, flush-left below the poet.

`opus`
: Name of the opus, flush-right below the composer.

`arranger`
: Name of the arranger, flush-right below the opus.

`instrument`
: Name of the instrument, centered below the arranger. Also centered at the top of pages (other than the first page).

`piece`
: Name of the piece, flush-left below the instrument.

`breakbefore`
: This forces the title to start on a new page (set to ##t or ##f).

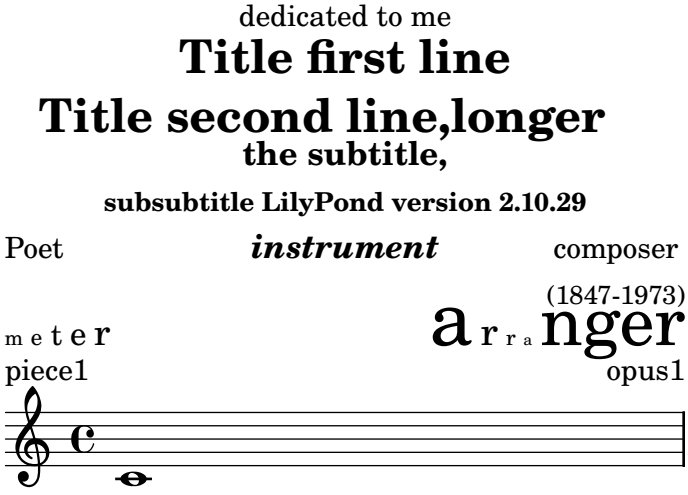
`copyright`
: Copyright notice, centered at the bottom of the first page. To insert the copyright symbol, see Section 10.1.7 [Text encoding], page 238.

`tagline`
: Centered at the bottom of the last page.

Here is a demonstration of the fields available. Note that you may use any Section 8.1.4 [Text markup], page 170 commands in the header.

```
\paper {
  line-width = 9.0\cm
  paper-height = 10.0\cm
}

\book {
  \header {
    dedication = "dedicated to me"
    title = \markup \center-align { "Title first line" "Title second line,
longer" }
    subtitle = "the subtitle,"
    subsubtitle = #(string-append "subsubtitle LilyPond version "
(lilypond-version))
    poet = "Poet"
    composer =  \markup \center-align { "composer" \small "(1847-1973)" }
    texttranslator = "Text Translator"
    meter = \markup { \teeny "m" \tiny "e" \normalsize "t" \large "e" \huge
"r" }
    arranger = \markup { \fontsize #8.5 "a" \fontsize #2.5 "r" \fontsize
#-2.5 "r" \fontsize #-5.3 "a" \fontsize #7.5 "nger" }
    instrument = \markup \bold \italic "instrument"
    piece = "Piece"
```

```
  }

  \score {
    { c'1 }
    \header {
      piece = "piece1"
      opus = "opus1"
    }
  }
  \markup {
      and now...
  }
  \score {
    { c'1 }
    \header {
      piece = "piece2"
      opus = "opus2"
    }
  }
}
```



2 ***instrument***

and now...

piece2 opus2

Music engraving by LilyPond 2.10.29—www.lilypond.org

As demonstrated before, you can use multiple `\header` blocks. When same fields appear in different blocks, the latter is used. Here is a short example.

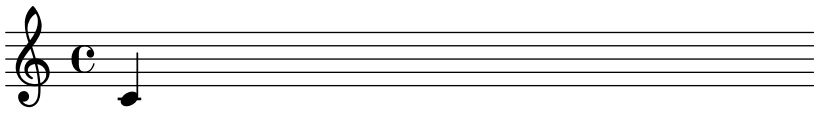
```
\header {
  composer = "Composer"
}
\header {
  piece = "Piece"
}
\score {
  \new Staff { c'4 }
  \header {
    piece = "New piece"  % overwrite previous one
  }
}
```

If you define the `\header` inside the `\score` block, then normally only the `piece` and `opus` headers will be printed. Note that the music expression must come before the `\header`.

```
\score {
  { c'4 }
  \header {
    title = "title"  % not printed
    piece = "piece"
    opus = "opus"
  }
}
```

title

piece opus



You may change this behavior (and print all the headers when defining `\header` inside `\score`) by using

```
\paper{
  printallheaders=##t
}
```

The default footer is empty, except for the first page, where the `copyright` field from `\header` is inserted, and the last page, where `tagline` from `\header` is added. The default tagline is "Music engraving by LilyPond (*version*)".[1]

Headers may be completely removed by setting them to false.

```
\header {
  tagline = ##f
  composer = ##f
}
```

10.2.2 Custom titles

A more advanced option is to change the definitions of the following variables in the `\paper` block. The init file '`ly/titling-init.ly`' lists the default layout.

`bookTitleMarkup`
: This is the title added at the top of the entire output document. Typically, it has the composer and the title of the piece

`scoreTitleMarkup`
: This is the title put over a `\score` block. Typically, it has the name of the movement (`piece` field).

`oddHeaderMarkup`
: This is the page header for odd-numbered pages.

`evenHeaderMarkup`
: This is the page header for even-numbered pages. If unspecified, the odd header is used instead.

  By default, headers are defined such that the page number is on the outside edge, and the instrument is centered.

`oddFooterMarkup`
: This is the page footer for odd-numbered pages.

`evenFooterMarkup`
: This is the page footer for even-numbered pages. If unspecified, the odd header is used instead.

  By default, the footer has the copyright notice on the first, and the tagline on the last page.

The following definition will put the title flush left, and the composer flush right on a single line.

```
\paper {
  bookTitleMarkup = \markup {
   \fill-line {
     \fromproperty #'header:title
     \fromproperty #'header:composer
   }
  }
}
```

[1] Nicely printed parts are good PR for us, so please leave the tagline if you can.

10.3 MIDI output

MIDI (Musical Instrument Digital Interface) is a standard for connecting and controlling digital instruments. A MIDI file is a series of notes in a number of tracks. It is not an actual sound file; you need special software to translate between the series of notes and actual sounds.

Pieces of music can be converted to MIDI files, so you can listen to what was entered. This is convenient for checking the music; octaves that are off or accidentals that were mistyped stand out very much when listening to the MIDI output.

Bugs

Many musically interesting effects, such as swing, articulation, slurring, etc., are not translated to midi.

The midi output allocates a channel for each staff, and one for global settings. Therefore the midi file should not have more than 15 staves (or 14 if you do not use drums). Other staves will remain silent.

Not all midi players correctly handle tempo changes in the midi output. Players that are known to work include timidity.

10.3.1 Creating MIDI files

To create a MIDI from a music piece of music, add a `\midi` block to a score, for example,

```
\score {
  ...music...
   \midi {
     \context {
       \Score
       tempoWholesPerMinute = #(ly:make-moment 72 4)
       }
     }
}
```

The tempo can be specified using the `\tempo` command within the actual music, see Section 8.2.2 [Metronome marks], page 186. An alternative, which does not result in a metronome mark in the printed score, is shown in the example above. In this example the tempo of quarter notes is set to 72 beats per minute. This kind of tempo specification can not take dotted note lengths as an argument. In this case, break the dotted notes into smaller units. For example, a tempo of 90 dotted quarter notes per minute can be specified as 270 eighth notes per minute

```
tempoWholesPerMinute = #(ly:make-moment 270 8)
```

If there is a `\midi` command in a `\score`, only MIDI will be produced. When notation is needed too, a `\layout` block must be added

```
\score {
  ...music...
  \midi { }
  \layout { }
}
```

Ties, dynamics, and tempo changes are interpreted. Dynamic marks, crescendi and decrescendi translate into MIDI volume levels. Dynamic marks translate to a fixed fraction of the available MIDI volume range, crescendi and decrescendi make the volume vary linearly between their two extremes. The fractions can be adjusted by `dynamicAbsoluteVolumeFunction` in `Voice` context. For each type of MIDI instrument, a volume range can be defined. This gives a basic equalizer control, which can enhance the quality of the MIDI output remarkably. The equalizer can be controlled by setting `instrumentEqualizer`, or by setting

```
\set Staff.midiMinimumVolume = #0.2
\set Staff.midiMaximumVolume = #0.8
```

To remove dynamics from the MIDI output, insert the following lines in the `\midi{}` section.

```
\midi {
  ...
  \context {
    \Voice
    \remove "Dynamic_performer"
  }
}
```

Bugs

Unterminated (de)crescendos will not render properly in the midi file, resulting in silent passages of music. The workaround is to explicitly terminate the (de)crescendo. For example,

```
{ a\< b c d\f }
```

will not work properly but

```
{ a\< b c d\!\f }
```

will.

10.3.2 MIDI block

The MIDI block is analogous to the layout block, but it is somewhat simpler. The `\midi` block is similar to `\layout`. It can contain context definitions.

Context definitions follow precisely the same syntax as within the \layout block. Translation modules for sound are called performers. The contexts for MIDI output are defined in '`ly/performer-init.ly`'.

10.3.3 MIDI instrument names

The MIDI instrument name is set by the `Staff.midiInstrument` property. The instrument name should be chosen from the list in Section C.2 [MIDI instruments], page 314.

```
\set Staff.midiInstrument = "glockenspiel"
...notes...
```

If the selected instrument does not exactly match an instrument from the list of MIDI instruments, the Grand Piano (`"acoustic grand"`) instrument is used.

10.4 Displaying LilyPond notation

Displaying a music expression in LilyPond notation can be done using the music function `\displayLilyMusic`. For example,

```
{
  \displayLilyMusic \transpose c a, { c e g a bes }
}
```

will display

```
{ a, cis e fis g }
```

By default, LilyPond will print these messages to the console along with all the other messages. To split up these messages and save the results of `\display{STUFF}`, redirect the output to a file.

```
lilypond file.ly >display.txt
```

10.5 Skipping corrected music

When entering or copying music, usually only the music near the end (where you are adding notes) is interesting to view and correct. To speed up this correction process, it is possible to skip typesetting of all but the last few measures. This is achieved by putting

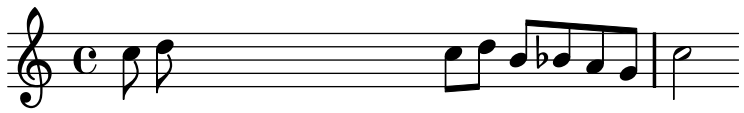
```
showLastLength = R1*5
\score { ... }
```

in your source file. This will render only the last 5 measures (assuming 4/4 time signature) of every `\score` in the input file. For longer pieces, rendering only a small part is often an order of magnitude quicker than rendering it completely

Skipping parts of a score can be controlled in a more fine-grained fashion with the property `Score.skipTypesetting`. When it is set, no typesetting is performed at all.

This property is also used to control output to the MIDI file. Note that it skips all events, including tempo and instrument changes. You have been warned.

```
\relative c'' {
  c8 d
  \set Score.skipTypesetting = ##t
  e e e e e e e e
  \set Score.skipTypesetting = ##f
  c d b bes a g c2 }
```



In polyphonic music, `Score.skipTypesetting` will affect all voices and staves, saving even more time.

11 Spacing issues

The global paper layout is determined by three factors: the page layout, the line breaks, and the spacing. These all influence each other. The choice of spacing determines how densely each system of music is set. This influences where line breaks are chosen, and thus ultimately, how many pages a piece of music takes.

Globally speaking, this procedure happens in four steps: first, flexible distances ("springs") are chosen, based on durations. All possible line breaking combinations are tried, and a "badness" score is calculated for each. Then the height of each possible system is estimated. Finally, a page breaking and line breaking combination is chosen so that neither the horizontal nor the vertical spacing is too cramped or stretched.

11.1 Paper and pages

This section deals with the boundaries that define the area that music can be printed inside.

11.1.1 Paper size

To change the paper size, there are two commands,

```
#(set-default-paper-size "a4")
\paper {
  #(set-paper-size "a4")
}
```

The first command sets the size of all pages. The second command sets the size of the pages that the `\paper` block applies to – if the `\paper` block is at the top of the file, then it will apply to all pages. If the `\paper` block is inside a `\book`, then the paper size will only apply to that book.

Support for the following paper sizes are included by default, `a6`, `a5`, `a4`, `a3`, `legal`, `letter`, `11x17` (also known as tabloid).

Extra sizes may be added by editing the definition for `paper-alist` in the initialization file '`scm/paper.scm`'.

If the symbol `landscape` is supplied as an argument to `set-default-paper-size`, the pages will be rotated by 90 degrees, and wider line widths will be set correspondingly.

```
#(set-default-paper-size "a6" 'landscape)
```

Setting the paper size will adjust a number of `\paper` variables (such as margins). To use a particular paper size with altered `\paper` variables, set the paper size before setting the variables.

11.1.2 Page formatting

LilyPond will do page layout, set margins, and add headers and footers to each page.

The default layout responds to the following settings in the `\paper` block.

`first-page-number`
: The value of the page number of the first page. Default is 1.

`print-first-page-number`
: If set to true, will print the page number in the first page. Default is false.

`print-page-number`
: If set to false, page numbers will not be printed. Default is true.

`paper-width`
: The width of the page. The default is taken from the current paper size, see Section 11.1.1 [Paper size], page 246.

`paper-height`
: The height of the page. The default is taken from the current paper size, see Section 11.1.1 [Paper size], page 246.

`top-margin`
: Margin between header and top of the page. Default is 5mm.

`bottom-margin`
: Margin between footer and bottom of the page. Default is 6mm.

`left-margin`
: Margin between the left side of the page and the beginning of the music. Unset by default, which means that the margins is determined based on the `paper-width` and `line-width` to center the score on the paper.

`line-width`
: The length of the systems. Default is `paper-width` minus 20mm.

`head-separation`
: Distance between the top-most music system and the page header. Default is 4mm.

`foot-separation`
: Distance between the bottom-most music system and the page footer. Default is 4mm.

`page-top-space`
: Distance from the top of the printable area to the center of the first staff. This only works for staves which are vertically small. Big staves are set with the top of their bounding box aligned to the top of the printable area. Default is 12mm.

`ragged-bottom`
: If set to true, systems will not be spread vertically across the page. This does not affect the last page. Default is false.

  This should be set to true for pieces that have only two or three systems per page, for example orchestral scores.

`ragged-last-bottom`
: If set to false, systems will be spread vertically to fill the last page. Default is true.

  Pieces that amply fill two pages or more should have this set to true.

`system-count`
: This variable, if set, specifies into how many lines a score should be broken. Unset by default.

`between-system-space`
: This dimensions determines the distance between systems. It is the ideal distance between the center of the bottom staff of one system and the center of the top staff of the next system. Default is 20mm.

  Increasing this will provide a more even appearance of the page at the cost of using more vertical space.

`between-system-padding`
: This dimension is the minimum amount of white space that will always be present between the bottom-most symbol of one system, and the top-most of the next system. Default is 4mm.

Increasing this will put systems whose bounding boxes almost touch farther apart.

`horizontal-shift`
: All systems (including titles and system separators) are shifted by this amount to the right. Page markup, such as headers and footers are not affected by this. The purpose of this variable is to make space for instrument names at the left. Default is 0.

`after-title-space`
: Amount of space between the title and the first system. Default is 5mm.

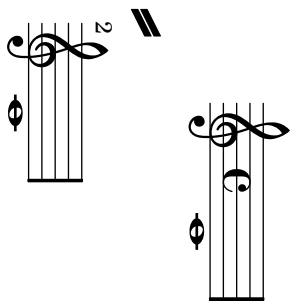
`before-title-space`
: Amount of space between the last system of the previous piece and the title of the next. Default is 10mm.

`between-title-space`
: Amount of space between consecutive titles (e.g., the title of the book and the title of a piece). Default is 2mm.

`printallheaders`
: Setting this to #t will print all headers for each \score in the output. Normally only the piece and opus \headers are printed.

`systemSeparatorMarkup`
: This contains a markup object, which will be inserted between systems. This is often used for orchestral scores. Unset by default.

  The markup command `\slashSeparator` is provided as a sensible default, for example



`blank-page-force`
: The penalty for having a blank page in the middle of a score. This is not used by `ly:optimal-breaking` since it will never consider blank pages in the middle of a score. Default value is 10.

`blank-last-page-force`
: The penalty for ending the score on an odd-numbered page. Default value is 0.

`page-spacing-weight`
: The relative importance of page (vertical) spacing and line (horizontal) spacing. High values will make page spacing more important. Default value is 10.

`auto-first-page-number`
: The page breaking algorithm is affected by the first page number being odd or even. If this variable is set to #t, the page breaking algorithm will decide whether to start with an odd or even number. This will result in the first page number remaining as is or being increased by one.

Example:

```
\paper{
  paper-width = 2\cm
  top-margin = 3\cm
```

```
  bottom-margin = 3\cm
  ragged-last-bottom = ##t
}
```

You can also define these values in Scheme. In that case `mm`, `in`, `pt`, and `cm` are variables defined in '`paper-defaults.ly`' with values in millimeters. That is why the value must be multiplied in the example

```
\paper {
  #(define bottom-margin (* 2 cm))
}
```

The header and footer are created by the functions `make-footer` and `make-header`, defined in `\paper`. The default implementations are in '`ly/paper-defaults.ly`' and '`ly/titling-init.ly`'.

The page layout itself is done by two functions in the `\paper` block, `page-music-height` and `page-make-stencil`. The former tells the line-breaking algorithm how much space can be spent on a page, the latter creates the actual page given the system to put on it.

Bugs

The option right-margin is defined but doesn't set the right margin yet. The value for the right margin has to be defined adjusting the values of `left-margin` and `line-width`.

The default page header puts the page number and the `instrument` field from the `\header` block on a line.

The titles (from the `\header{}` section) are treated as a system, so `ragged-bottom` and `ragged-last-bottom` will add space between the titles and the first system of the score.

11.2 Music layout

11.2.1 Setting the staff size

To set the staff size globally for all scores in a file (or in a `book` block, to be precise), use `set-global-staff-size`.

```
#(set-global-staff-size 14)
```

This sets the global default size to 14pt staff height and scales all fonts accordingly.

To set the staff size individually for each score, use

```
\score{
  ...
  \layout{
  #(layout-set-staff-size 15)
  }
}
```

The Feta font provides musical symbols at eight different sizes. Each font is tuned for a different staff size: at a smaller size the font becomes heavier, to match the relatively heavier staff lines. The recommended font sizes are listed in the following table:

font name	**staff height (pt)**	**staff height (mm)**	**use**
feta11	11.22	3.9	pocket scores
feta13	12.60	4.4	
feta14	14.14	5.0	
feta16	15.87	5.6	

feta18	17.82	6.3	song books
feta20	20	7.0	standard parts
feta23	22.45	7.9	
feta26	25.2	8.9	

These fonts are available in any sizes. The context property `fontSize` and the layout property `staff-space` (in `StaffSymbol`) can be used to tune the size for individual staves. The sizes of individual staves are relative to the global size.

See also

This manual: Section 8.4.8 [Selecting notation font size], page 207.

11.2.2 Score layout

While `\paper` contains settings that relate to the page formatting of the whole document, `\layout` contains settings for score-specific layout.

```
\layout {
  indent = 2.0\cm
  \context { \Staff
    \override VerticalAxisGroup #'minimum-Y-extent = #'(-6 . 6)
  }
  \context { \Voice
    \override TextScript #'padding = #1.0
    \override Glissando #'thickness = #3
  }
}
```

See also

This manual: Section 9.2.6 [Changing context default settings], page 225

11.3 Breaks

11.3.1 Line breaking

Line breaks are normally computed automatically. They are chosen so that lines look neither cramped nor loose, and that consecutive lines have similar density.

Occasionally you might want to override the automatic breaks; you can do this by specifying `\break`. This will force a line break at this point. Line breaks can only occur at places where there are bar lines. If you want to have a line break where there is no bar line, you can force an invisible bar line by entering `\bar ""`. Similarly, `\noBreak` forbids a line break at a point.

For line breaks at regular intervals use `\break` separated by skips and repeated with `\repeat`:

```
<< \repeat unfold 7 {
          s1 \noBreak s1 \noBreak
          s1 \noBreak s1 \break }
   the real music
>>
```

This makes the following 28 measures (assuming 4/4 time) be broken every 4 measures, and only there.

Predefined commands

`\break`, and `\noBreak`.

See also

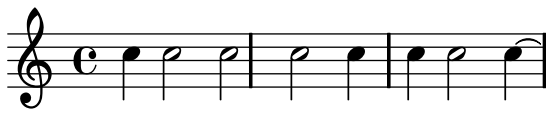
Internals: `LineBreakEvent`.

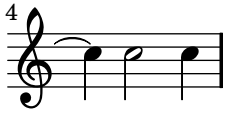
A linebreaking configuration can now be saved as a `.ly` file automatically. This allows vertical alignments to be stretched to fit pages in a second formatting run. This is fairly new and complicated.

Bugs

Line breaks can only occur if there is a 'proper' bar line. A note which is hanging over a bar line is not proper, such as

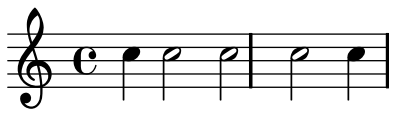
```
c4 c2 c2 \break   % this does nothing
c2 c4 |           % a break here would work
c4 c2 c4 ~ \break % as does this break
c4 c2 c4
```





To allow line breaks on such bar lines, the `Forbid_line_break_engraver` can be removed from `Voice` context, like so

```
\new Voice \with {
  \remove "Forbid_line_break_engraver"
} {
  c4 c2 c2 \break   % now the break is allowed
  c2 c4
}
```



11.3.2 Page breaking

The default page breaking may be overriden by inserting `\pageBreak` or `\noPageBreak` commands. These commands are analogous to `\break` and `\noBreak`. They should be inserted at a bar line. These commands force and forbid a page-break from happening. Of course, the `\pageBreak` command also forces a line break.

Page breaks are computed by the `page-breaking` function. LilyPond provides two algorithms for computing page breaks, `ly:optimal-breaking` and `ly:page-turn-breaking`. The default is `ly:optimal-breaking`, but the value can be changed in the `\paper` block:

```
\paper{
  #(define page-breaking ly:page-turn-breaking)
}
```

The old page breaking algorithm is called `optimal-page-breaks`. If you are having trouble with the new page breakers, you can enable the old one as a workaround.

Predefined commands

`\pageBreak` `\noPageBreak`

11.3.3 Optimal page breaking

The `ly:optimal-breaking` function is LilyPond's default method of determining page breaks. It attempts to find a page breaking that minimizes cramping and stretching, both horizontally and vertically. Unlike `ly:page-turn-breaking`, it has no concept of page turns.

11.3.4 Optimal page turning

Often it is necessary to find a page breaking configuration so that there is a rest at the end of every second page. This way, the musician can turn the page without having to miss notes. The `ly:page-turn-breaking` function attempts to find a page breaking minimizing cramping and stretching, but with the additional restriction that it is only allowed to introduce page turns in specified places.

There are two steps to using this page breaking function. First, you must enable it in the `\paper` block. Then, you must tell the function where you would like to allow page breaks.

There are two ways to achieve the second step. First, you can specify each potential page turn manually, by inserting `\allowPageTurn` into your input file at the appropriate places.

If this is too tedious, you can add a `Page_turn_engraver` to a Staff or Voice context. The `Page_turn_engraver` will scan the context for sections without notes (note that it does not scan for rests; it scans for the absence of notes. This is so that single-staff polyphony with rests in one of the parts does not throw off the `Page_turn_engraver`). When it finds a sufficiently long section without notes, the `Page_turn_engraver` will insert an `\allowPageTurn` at the final barline in that section, unless there is a 'special' barline (such as a double bar), in which case the `\allowPageTurn` will be inserted at the final 'special' barline in the section.

The `Page_turn_engraver` reads the context property `minimumPageTurnLength` to determine how long a note-free section must be before a page turn is considered. The default value for `minimumPageTurnLength` is `#(ly:make-moment 1 1)`. If you want to disable page turns, you can set it to something very large.

```
\new Staff \with { \consists "Page_turn_engraver" }
{
  a4 b c d |
  R1 | % a page turn will be allowed here
  a4 b c d |
  \set Staff.minimumPageTurnLength = #(ly:make-moment 5 2)
  R1 | % a page turn will not be allowed here
  a4 b r2 |
  R1*2 | % a page turn will be allowed here
  a1
}
```

The `Page_turn_engraver` detects volta repeats. It will only allow a page turn during the repeat if there is enough time at the beginning and end of the repeat to turn the page back. The `Page_turn_engraver` can also disable page turns if the repeat is very short. If you set the

context property `minimumRepeatLengthForPageTurn` then the `Page_turn_engraver` will only allow turns in repeats whose duration is longer than this value.

Bugs

There should only be one `Page_turn_engraver` in a score. If there is more than one, they will interfere with each other.

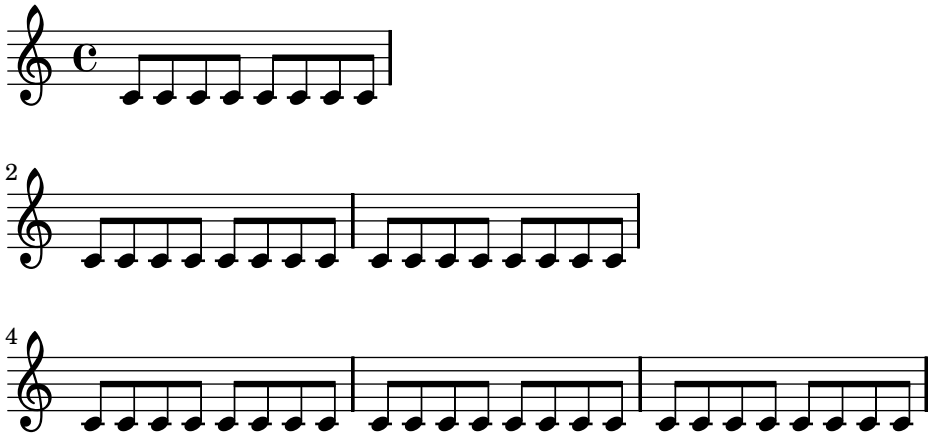
11.3.5 Explicit breaks

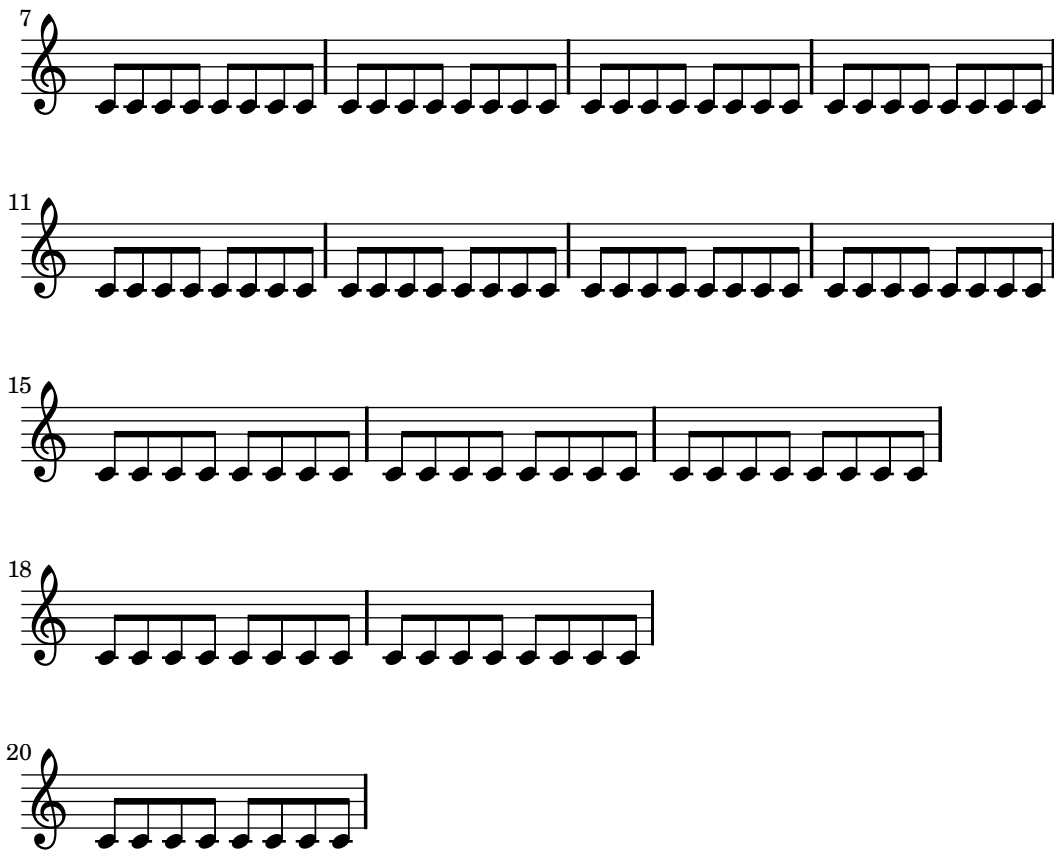
Lily sometimes rejects explicit `\break` and `\pageBreak` commands. There are two commands to override this behavior:

```
\override NonMusicalPaperColumn #'line-break-permission = ##f
\override NonMusicalPaperColumn #'page-break-permission = ##f
```

When `line-break-permission` is overriden to false, Lily will insert line breaks at explicit `\break` commands and nowhere else. When `page-break-permission` is overriden to false, Lily will insert page breaks at explicit `\pageBreak` commands and nowhere else.

```
\paper {
  indent = #0
  ragged-right = ##t
  ragged-bottom = ##t
}

\score {
  \new Score \with {
    \override NonMusicalPaperColumn #'line-break-permission = ##f
    \override NonMusicalPaperColumn #'page-break-permission = ##f
  } {
    \new Staff {
      \repeat unfold 2 { c'8 c'8 c'8 c'8 } \break
      \repeat unfold 4 { c'8 c'8 c'8 c'8 } \break
      \repeat unfold 6 { c'8 c'8 c'8 c'8 } \break
      \repeat unfold 8 { c'8 c'8 c'8 c'8 } \pageBreak
      \repeat unfold 8 { c'8 c'8 c'8 c'8 } \break
      \repeat unfold 6 { c'8 c'8 c'8 c'8 } \break
      \repeat unfold 4 { c'8 c'8 c'8 c'8 } \break
      \repeat unfold 2 { c'8 c'8 c'8 c'8 }
    }
  }
}
```





11.3.6 Using an extra voice for breaks

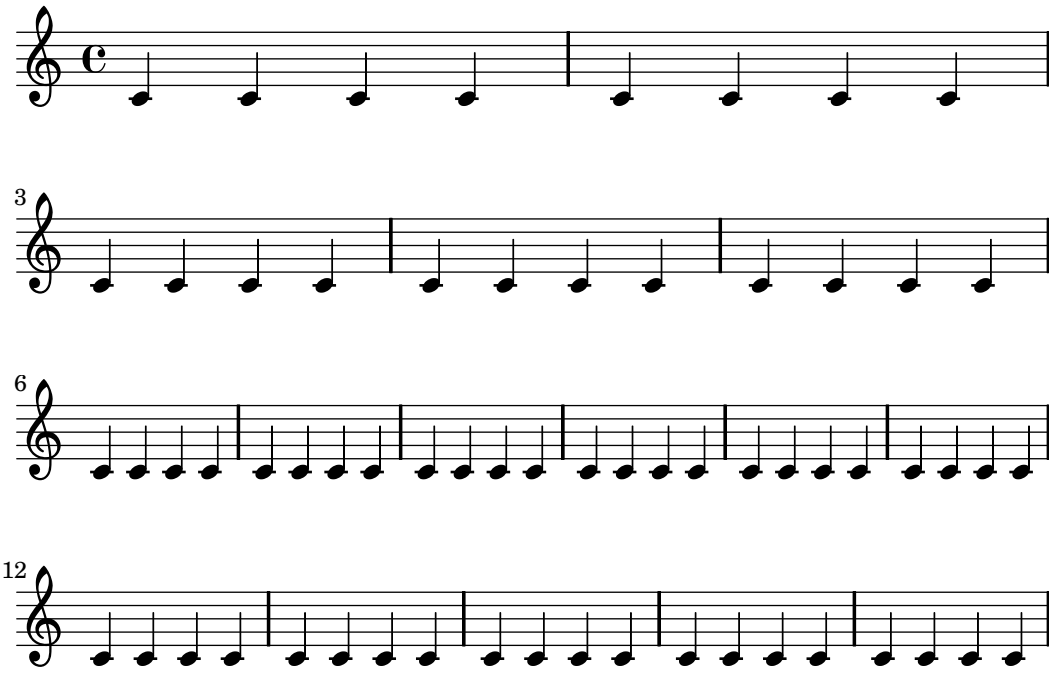
Line- and page-breaking information usually appears within note entry directly.

```
\new Score {
  \new Staff {
    \repeat unfold 2 { c'4 c'4 c'4 c'4 }
    \break
    \repeat unfold 3 { c'4 c'4 c'4 c'4 }
  }
}
```

This makes `\break` and `\pageBreak` commands easy to enter but mixes music entry with information that specifies how music should lay out on the page. You can keep music entry and line- and page-breaking information in two separate places by introducing an extra voice to contain the breaks. This extra voice contains only skips together with `\break`, `pageBreak` and other breaking layout information.

```
\new Score {
  \new Staff <<
     \new Voice {
        s1 * 2 \break
        s1 * 3 \break
        s1 * 6 \break
        s1 * 5 \break
     }
     \new Voice {
        \repeat unfold 2 { c'4 c'4 c'4 c'4 }
        \repeat unfold 3 { c'4 c'4 c'4 c'4 }
        \repeat unfold 6 { c'4 c'4 c'4 c'4 }
        \repeat unfold 5 { c'4 c'4 c'4 c'4 }
     }
```

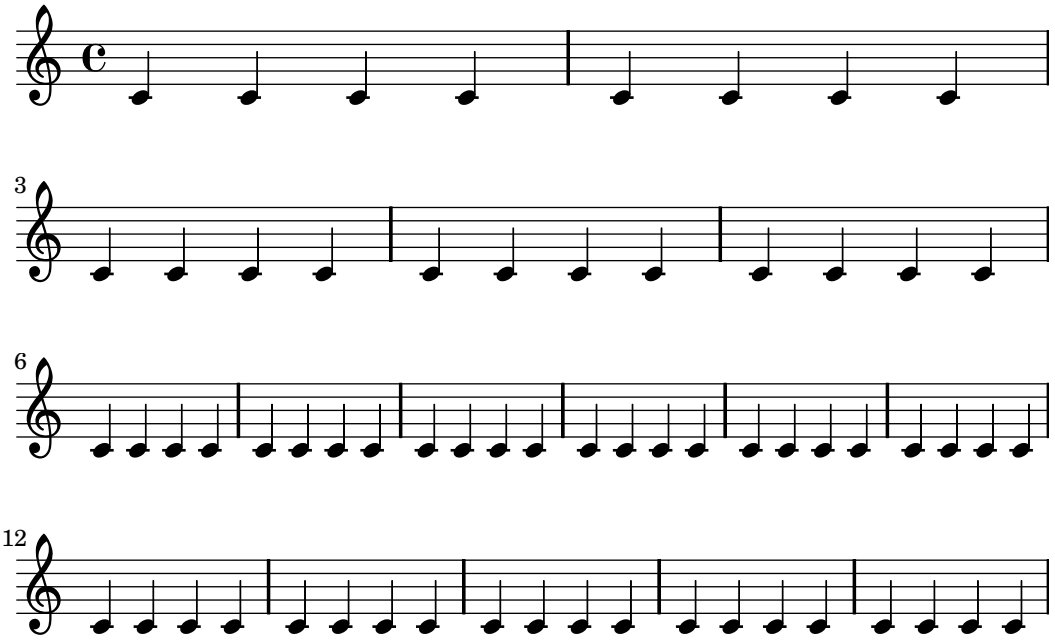
```
  >>
}
```



This pattern becomes especially helpful when overriding `line-break-system-details` and the other useful but long properties of `NonMusicalPaperColumnGrob`, as explained in Section 11.4 [Vertical spacing], page 257.

```
\new Score {
  \new Staff <<
     \new Voice {

        \overrideProperty "Score.NonMusicalPaperColumn"
        #'line-break-system-details #'((Y-offset . 0))
        s1 * 2 \break

        \overrideProperty "Score.NonMusicalPaperColumn"
        #'line-break-system-details #'((Y-offset . 35))
        s1 * 3 \break

        \overrideProperty "Score.NonMusicalPaperColumn"
        #'line-break-system-details #'((Y-offset . 70))
        s1 * 6 \break

        \overrideProperty "Score.NonMusicalPaperColumn"
        #'line-break-system-details #'((Y-offset . 105))
        s1 * 5 \break
     }
     \new Voice {
        \repeat unfold 2 { c'4 c'4 c'4 c'4 }
        \repeat unfold 3 { c'4 c'4 c'4 c'4 }
        \repeat unfold 6 { c'4 c'4 c'4 c'4 }
        \repeat unfold 5 { c'4 c'4 c'4 c'4 }
```

```
        }
    >>
}
```



11.4 Vertical spacing

Vertical spacing is controlled by three things: the amount of space available (i.e., paper size and margins), the amount of space between systems, and the amount of space between staves inside a system.

11.4.1 Vertical spacing inside a system

The height of each system is determined automatically. To prevent staves from bumping into each other, some minimum distances are set. By changing these, you can put staves closer together. This reduces the amount of space each system requires, and may result in having more systems per page.

Normally staves are stacked vertically. To make staves maintain a distance, their vertical size is padded. This is done with the property `minimum-Y-extent`. When applied to a `VerticalAxisGroup`, it controls the size of a horizontal line, such as a staff or a line of lyrics. `minimum-Y-extent` takes a pair of numbers, so if you want to make it smaller than its default `#'(-4 . 4)` then you could set

```
\override Staff.VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
```

This sets the vertical size of the current staff to 3 staff spaces on either side of the center staff line. The value `(-3 . 3)` is interpreted as an interval, where the center line is the 0, so the first number is generally negative. The numbers need not match; for example, the staff can be made larger at the bottom by setting it to `(-6 . 4)`.

See also

Internals: Vertical alignment of staves is handled by the `VerticalAlignment` object. The context parameters specifying the vertical extent are described in connection with the `Axis_group_engraver`.

Example files: ‘`input/regression//page-spacing.ly`’, ‘`input/regression//alignment-vertical-spacing.ly`’.

11.4.2 Vertical spacing of piano staves

The distance between staves of a `PianoStaff` cannot be computed during formatting. Rather, to make cross-staff beaming work correctly, that distance has to be fixed beforehand.

The distance of staves in a `PianoStaff` is set with the `forced-distance` property of the `VerticalAlignment` object, created in `PianoStaff`.

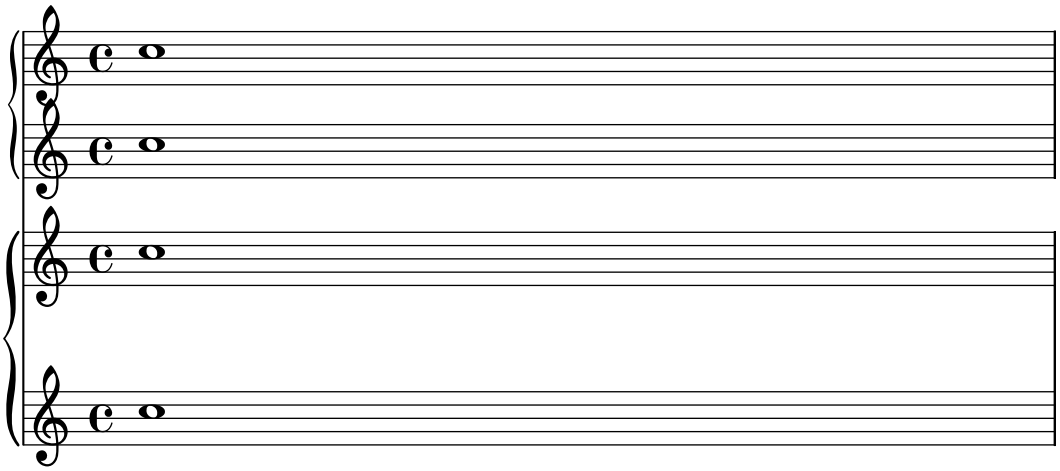
It can be adjusted as follows

```
\new PianoStaff \with {
  \override VerticalAlignment #'forced-distance = #7
} {
  ...
}
```

This would bring the staves together at a distance of 7 staff spaces, measured from the center line of each staff.

The difference is demonstrated in the following example,

```
\relative c'' <<
  \new PianoStaff \with {
    \override VerticalAlignment #'forced-distance = #7
  } <<
    \new Staff { c1 }
    \new Staff { c }
  >>
  \new PianoStaff <<
    \new Staff { c }
    \new Staff { c }
  >>
>>
```



See also

Example files: ‘`input/regression//alignment-vertical-spacing.ly`’.

11.4.3 Vertical spacing between systems

Space between systems are controlled by four `\paper` variables,

```
\paper {
  between-system-space = 1.5\cm
```

```
    between-system-padding = #1
    ragged-bottom=##f
    ragged-last-bottom=##f
}
```

11.4.4 Explicit staff and system positioning

One way to understand the `VerticalAxisGroup` and `\paper` settings explained in the previous two sections is as a collection of different settings that primarily concern the amount of vertical padding different staves and systems running down the page.

It is possible to approach vertical spacing in a different way using `NonMusicalPaperColumn #'line-break-system-details`. Where `VerticalAxisGroup` and `\paper` settings specify vertical padding, `NonMusicalPaperColumn #'line-break-system-details` specifies exact vertical positions on the page.

`NonMusicalPaperColumn #'line-break-system-details` accepts an associative list of five different settings:

- `X-offset`
- `Y-offset`
- `alignment-offsets`
- `alignment-extra-space`
- `fixed-alignment-extra-space`

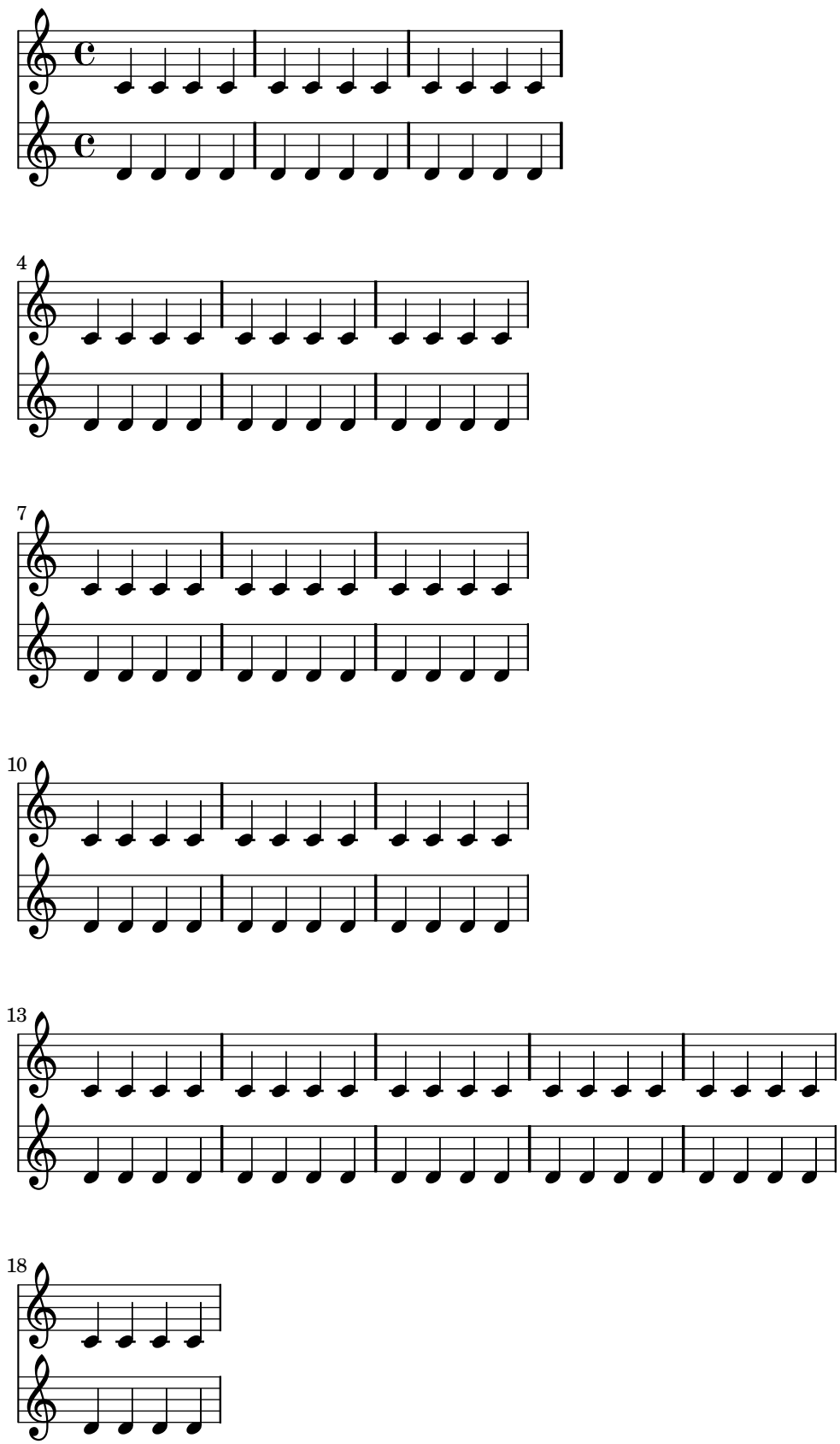
Grob overrides, including the overrides for `NonMusicalPaperColumn` below, can occur in any of three different places in an input file:

- in the middle of note entry directly
- in a `\context` block
- in the `\with` block

When we override `NonMusicalPaperColumn`, we use the usual `\override` command in `\context` blocks and in the `\with` block. On the other hand, when we override `NonMusicalPaperColumn` in the middle of note entry, use the special `\overrideProperty` command. Here are some example `NonMusicalPaperColumn` overrides with the special `\overrideProperty` command:

```
\overrideProperty NonMusicalPaperColumn
  #'line-break-system-details #'((X-offset . 20))

\overrideProperty NonMusicalPaperColumn
  #'line-break-system-details #'((Y-offset . 40))

\overrideProperty NonMusicalPaperColumn
  #'line-break-system-details #'((X-offset . 20) (Y-offset . 40))

\override NonMusicalPaperColumn
  #'line-break-system-details #'((alignment-offsets . (0 -15)))

\override NonMusicalPaperColumn
  #'line-break-system-details #'((X-offset . 20) (Y-offset . 40)
                                 (alignment-offsets . (0 -15)))
```

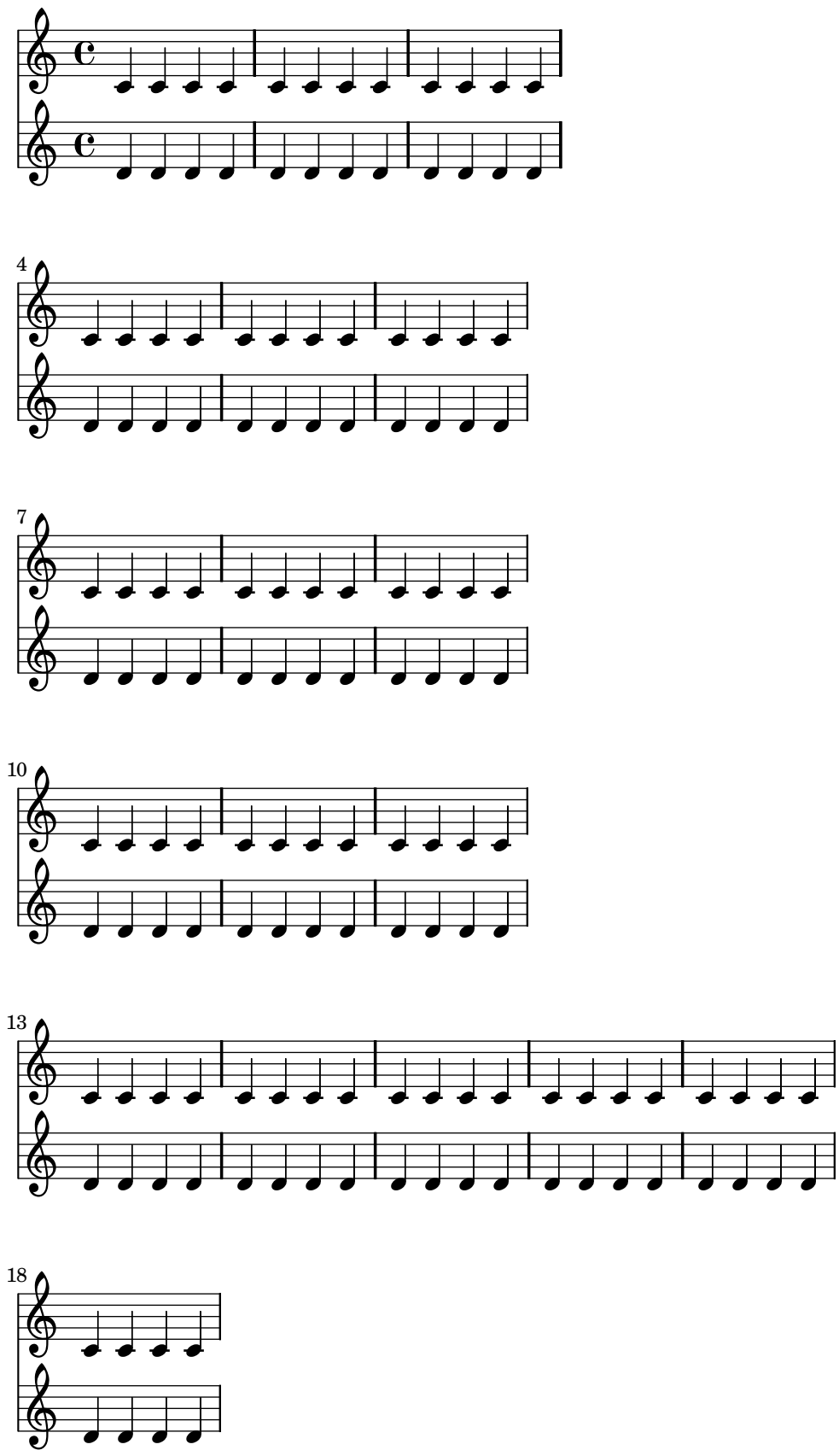
To understand how each of these different settings work, we begin by looking at an example that includes no overrides at all.



This score isolates line- and page-breaking information in a dedicated voice. This technique of creating a breaks voice will help keep layout separate from music entry as our example becomes more complicated. See Section 11.3.6 [Using an extra voice for breaks], page 255.

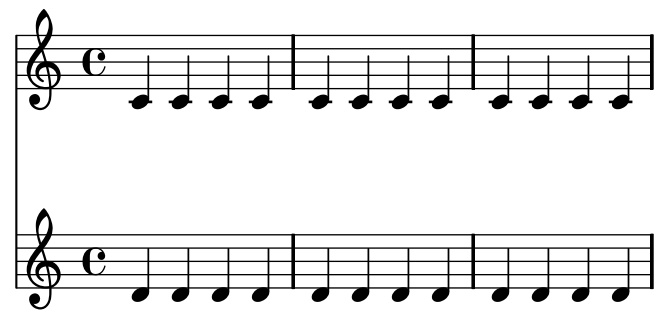
Explicit `\breaks` evenly divide the music into six measures per line. Vertical spacing results from LilyPond's defaults. To set the vertical startpoint of each system explicitly, we can set the

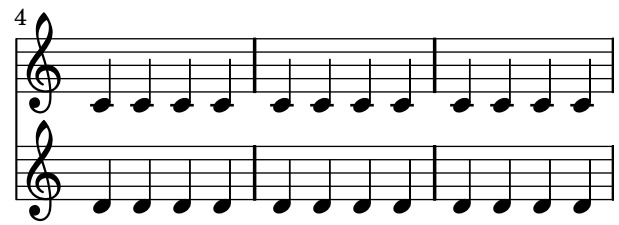
`Y-offset` pair in the `line-break-system-details` attribute of the `NonMusicalPaperColumn` grob:

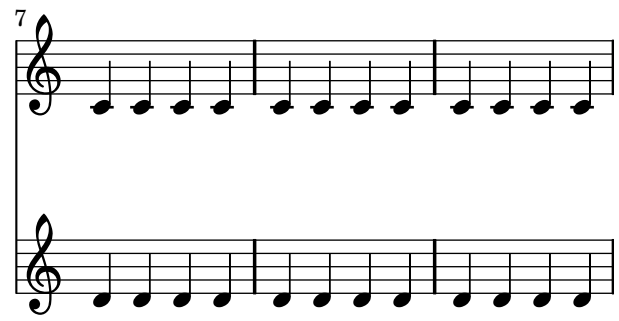


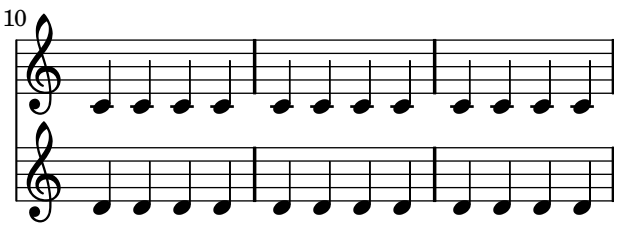
Note that `line-break-system-details` takes an associative list of potentially many values, but that we set only one value here. Note, too, that the `Y-offset` property here determines the exact vertical position on the page at which each new system will render.

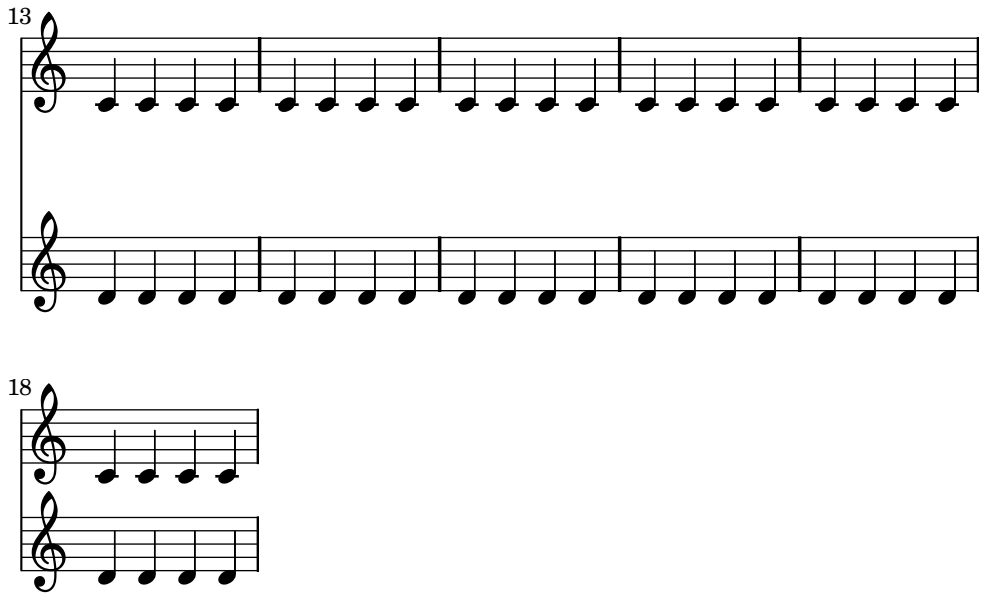
Now that we have set the vertical startpoint of each system explicitly, we can also set the vertical startpoint of each staff within each system manually. We do this using the `alignment-offsets` subproperty of `line-break-system-details`.



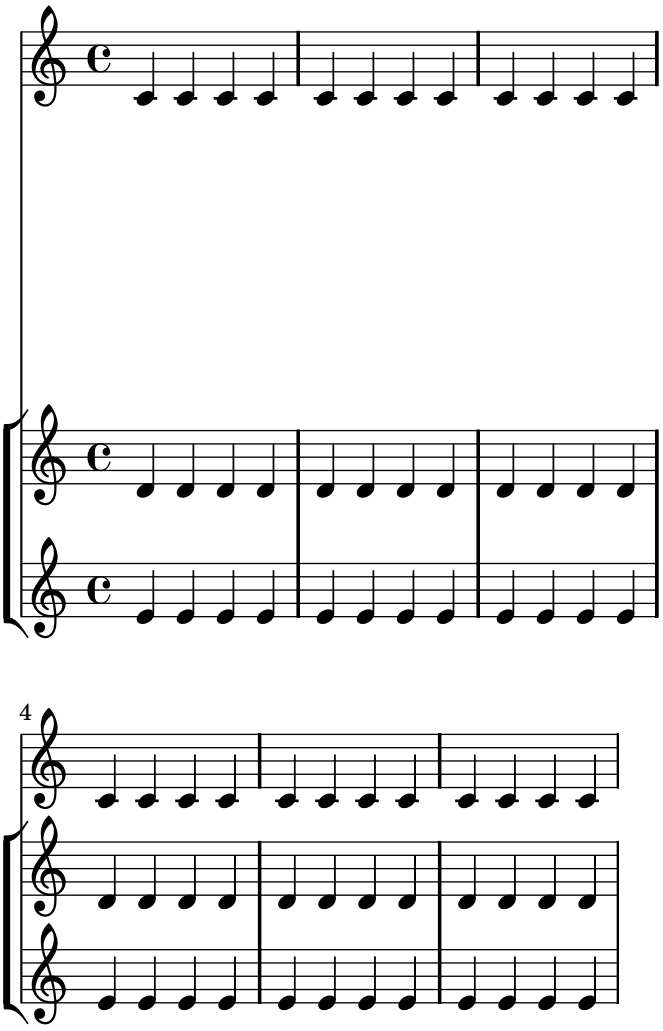


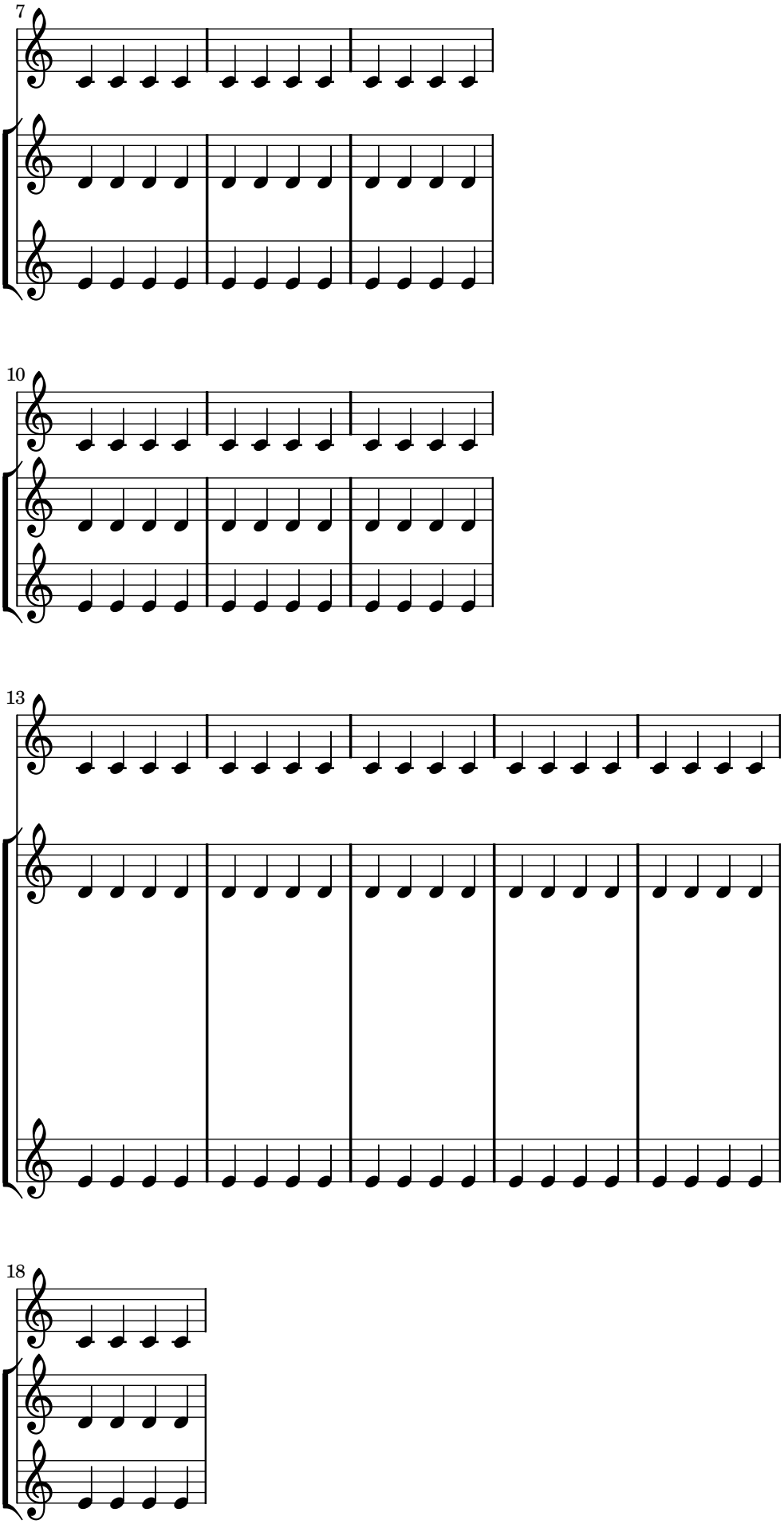






Note that here we assign two different values to the `line-break-system-details` attribute of the `NonMusicalPaperColumn` grob. Though the `line-break-system-details` attribute alist accepts many additional spacing parameters (including, for example, a corresponding `X-offset` pair), we need only set the `Y-offset` and `alignment-offsets` pairs to control the vertical startpoint of every system and every staff. Finally, note that `alignment-offsets` specifies the vertical positioning of staves but not of staff groups.




7
10
13
18

Some points to consider:

- When using `alignment-offsets`, lyrics count as a staff.
- The units of the numbers passed to `X-offset`, `Y-offset` and `alignment-offsets` are interpreted as multiples of the distance between adjacent staff lines. Positive values move staves and lyrics up, negative values move staves and lyrics down.
- Because the `NonMusicalPaperColumn #'line-break-system-details` settings given here allow the positioning of staves and systems anywhere on the page, it is possible to violate paper or margin boundaries or even to print staves or systems on top of one another. Reasonable values passed to these different settings will avoid this.

11.4.5 Two-pass vertical spacing

In order to automatically stretch systems so that they should fill the space left on a page, a two-pass technique can be used:

1. In the first pass, the amount of vertical space used to increase the height of each system is computed and dumped to a file.
2. In the second pass, spacing inside the systems are stretched according to the data in the page layout file.

The `ragged-bottom` property adds space between systems, while the two-pass technique adds space between staffs inside a system.

To allow this behaviour, a `tweak-key` variable has to be set in each score `\layout` block, and the tweaks included in each score music, using the `\scoreTweak` music function.

```
%% include the generated page layout file:
\includePageLayoutFile

\score {
  \new StaffGroup <<
    \new Staff <<
      %% Include this score tweaks:
      \scoreTweak "scoreA"
      { \clef french c''1 \break c''1 }
    >>
    \new Staff { \clef soprano g'1 g'1 }
    \new Staff { \clef mezzosoprano e'1 e'1 }
    \new Staff { \clef alto g1 g1 }
    \new Staff { \clef bass c1 c1 }
  >>
  \header {
    piece = "Score with tweaks"
  }
  %% Define how to name the tweaks for this score:
  \layout { #(define tweak-key "scoreA") }
}
```

For the first pass, the `dump-tweaks` option should be set to generate the page layout file.

```
lilypond -b null -d dump-tweaks <file>.ly
lilypond <file>.ly
```

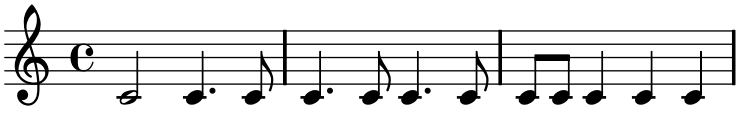
11.5 Horizontal Spacing

11.5.1 Horizontal spacing overview

The spacing engine translates differences in durations into stretchable distances ("springs") of differing lengths. Longer durations get more space, shorter durations get less. The shortest durations get a fixed amount of space (which is controlled by `shortest-duration-space` in the `SpacingSpanner` object). The longer the duration, the more space it gets: doubling a duration adds a fixed amount (this amount is controlled by `spacing-increment`) of space to the note.

For example, the following piece contains lots of half, quarter, and 8th notes; the eighth note is followed by 1 note head width (NHW). The quarter note is followed by 2 NHW, the half by 3 NHW, etc.

```
c2 c4. c8 c4. c8 c4. c8 c8
c8 c4 c4 c4
```



Normally, `spacing-increment` is set to 1.2 staff space, which is approximately the width of a note head, and `shortest-duration-space` is set to 2.0, meaning that the shortest note gets 2.4 staff space (2.0 times the `spacing-increment`) of horizontal space. This space is counted from the left edge of the symbol, so the shortest notes are generally followed by one NHW of space.

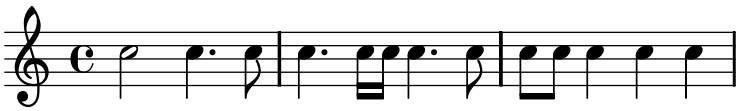
If one would follow the above procedure exactly, then adding a single 32nd note to a score that uses 8th and 16th notes, would widen up the entire score a lot. The shortest note is no longer a 16th, but a 32nd, thus adding 1 NHW to every note. To prevent this, the shortest duration for spacing is not the shortest note in the score, but rather the one which occurs most frequently.

The most common shortest duration is determined as follows: in every measure, the shortest duration is determined. The most common shortest duration is taken as the basis for the spacing, with the stipulation that this shortest duration should always be equal to or shorter than an 8th note. The shortest duration is printed when you run `lilypond` with the `--verbose` option.

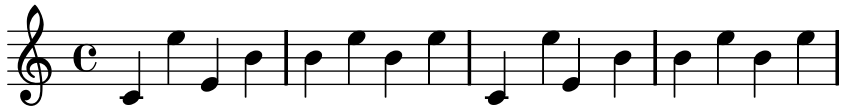
These durations may also be customized. If you set the `common-shortest-duration` in `SpacingSpanner`, then this sets the base duration for spacing. The maximum duration for this base (normally an 8th), is set through `base-shortest-duration`.

Notes that are even shorter than the common shortest note are followed by a space that is proportional to their duration relative to the common shortest note. So if we were to add only a few 16th notes to the example above, they would be followed by half a NHW:

```
c2 c4. c8 c4. c16[ c] c4. c8 c8 c8 c4 c4 c4
```



In the introduction (see Section 1.1 [Engraving], page 2), it was explained that stem directions influence spacing. This is controlled with the `stem-spacing-correction` property in the `NoteSpacing`, object. These are generated for every `Voice` context. The `StaffSpacing` object (generated in `Staff` context) contains the same property for controlling the stem/bar line spacing. The following example shows these corrections, once with default settings, and once with exaggerated corrections:



Proportional notation is supported; see Section 8.4.3 [Proportional notation], page 204.

See also

Internals: `SpacingSpanner`, `NoteSpacing`, `StaffSpacing`, `SeparationItem`, and `SeparatingGroupSpanner`.

Bugs

There is no convenient mechanism to manually override spacing. The following work-around may be used to insert extra space into a score.

```
\once \override Score.SeparationItem #'padding = #1
```

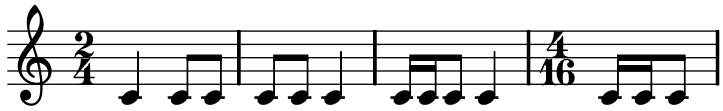
No work-around exists for decreasing the amount of space.

11.5.2 New spacing area

New sections with different spacing parameters can be started with `newSpacingSection`. This is useful when there are sections with a different notions of long and short notes.

In the following example, the time signature change introduces a new section, and hence the 16ths notes are spaced wider.

```
\time 2/4
c4 c8 c
c8 c c4 c16[ c c8] c4
\newSpacingSection
\time 4/16
c16[ c c8]
```

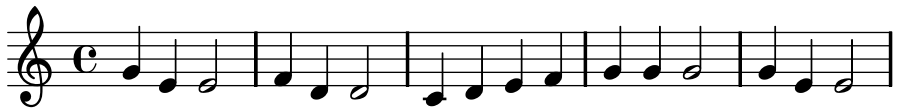


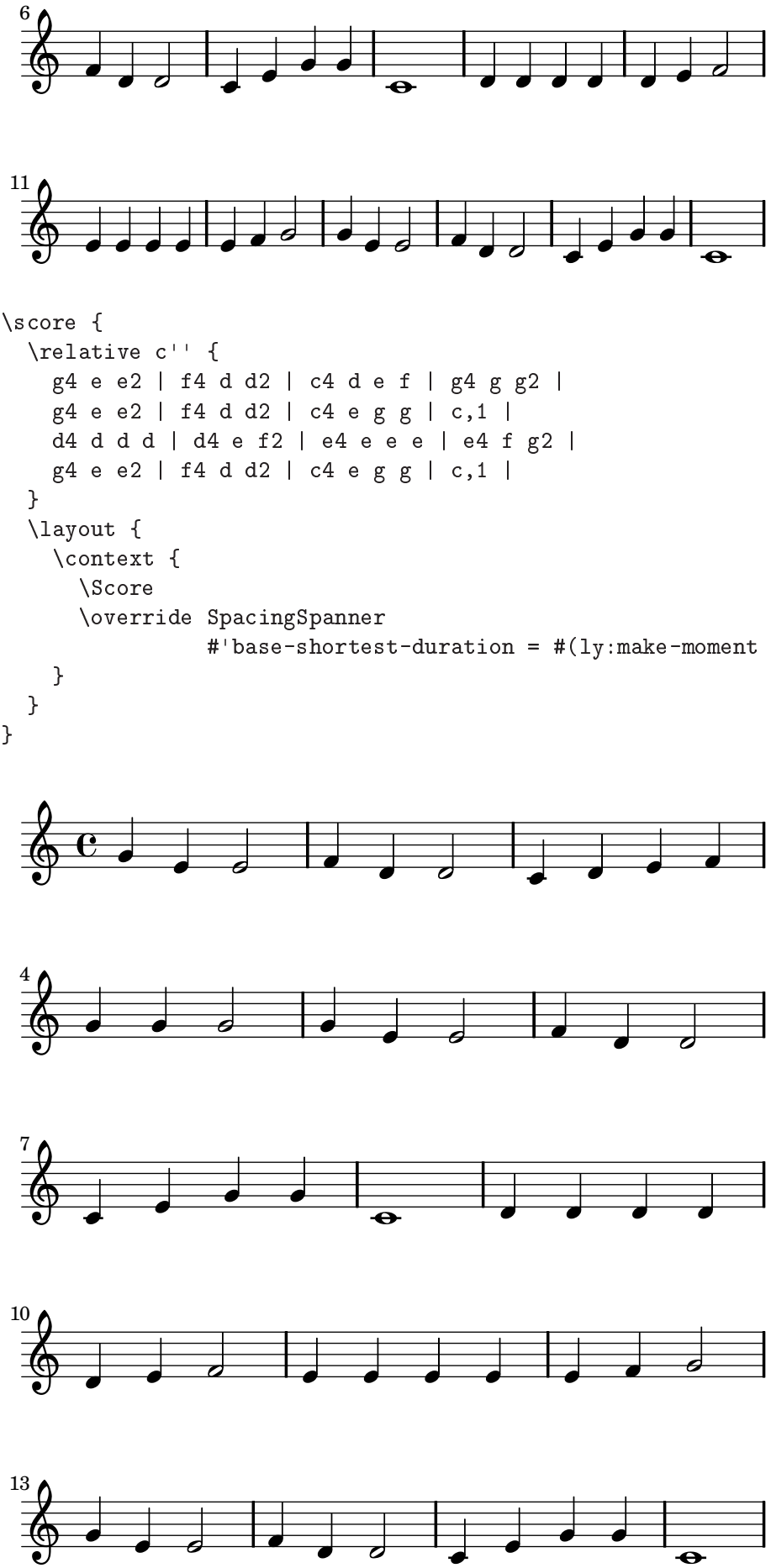
The `\newSpacingSection` command creates a new `SpacingSpanner` object, and hence new `\overrides` may be used in that location.

11.5.3 Changing horizontal spacing

Horizontal spacing may be altered with the `base-shortest-duration` property. Here we compare the same music; once without altering the property, and then altered. Larger values of `ly:make-moment` will produce smaller music. Note that `ly:make-moment` constructs a duration, so `1 4` is a longer duration than `1 16`.

```
\score {
  \relative c'' {
    g4 e e2 | f4 d d2 | c4 d e f | g4 g g2 |
    g4 e e2 | f4 d d2 | c4 e g g | c,1 |
    d4 d d d | d4 e f2 | e4 e e e | e4 f g2 |
    g4 e e2 | f4 d d2 | c4 e g g | c,1 |
  }
}
```



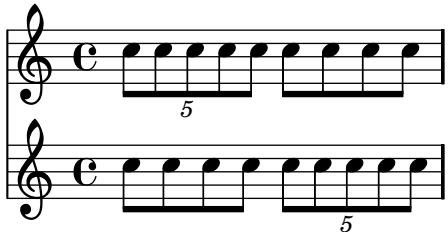


```
\score {
  \relative c'' {
    g4 e e2 | f4 d d2 | c4 d e f | g4 g g2 |
    g4 e e2 | f4 d d2 | c4 e g g | c,1 |
    d4 d d d | d4 e f2 | e4 e e e | e4 f g2 |
    g4 e e2 | f4 d d2 | c4 e g g | c,1 |
  }
  \layout {
    \context {
      \Score
      \override SpacingSpanner
                #'base-shortest-duration = #(ly:make-moment 1 16)
    }
  }
}
```

Commonly tweaked properties

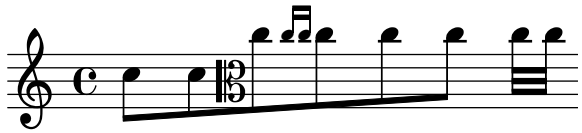
By default, spacing in tuplets depends on various non-duration factors (such as accidentals, clef changes, etc). To disregard such symbols and force uniform equal-duration spacing, use `Score.SpacingSpanner #'uniform-stretching`. This property can only be changed at the beginning of a score,

```
\new Score \with {
  \override SpacingSpanner #'uniform-stretching = ##t
} <<
  \new Staff{
    \times 4/5 {
      c8 c8 c8 c8 c8
    }
    c8 c8 c8 c8
  }
  \new Staff{
    c8 c8 c8 c8
    \times 4/5 {
      c8 c8 c8 c8 c8
    }
  }
>>
```



When `strict-note-spacing` is set, notes are spaced without regard for clefs, bar lines, and grace notes,

```
\override Score.SpacingSpanner #'strict-note-spacing = ##t
\new Staff { c8[ c \clef alto c \grace { c16[ c] } c8 c c]  c32[ c32] }
```



11.5.4 Line length

The most basic settings influencing the spacing are `indent` and `line-width`. They are set in the `\layout` block. They control the indentation of the first line of music, and the lengths of the lines.

If `ragged-right` is set to true in the `\layout` block, then systems ends at their natural horizontal length, instead of being spread horizontally to fill the whole line. This is useful for short fragments, and for checking how tight the natural spacing is.

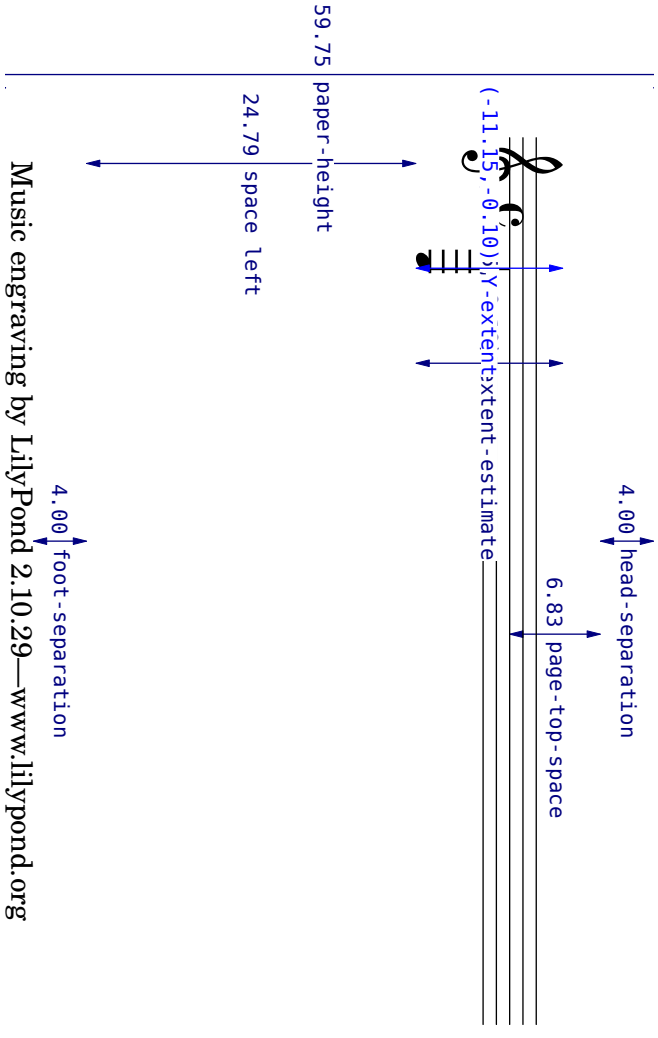
The option `ragged-last` is similar to `ragged-right`, but only affects the last line of the piece. No restrictions are put on that line. The result is similar to formatting text paragraphs. In a paragraph, the last line simply takes its natural horizontal length.

```
\layout {
  indent = #0
  line-width = #150
  ragged-last = ##t
}
```

11.6 Displaying spacing

To graphically display the dimensions of vertical properties that may be altered for page formatting, set `annotate-spacing` in the `\paper` block, like this

```
#(set-default-paper-size "a6" 'landscape)

\book {
  \score { { c4 } }
  \paper { annotate-spacing = ##t }
}
```



Some unit dimensions are measured in staff spaces, while others are measured in millimeters. The pairs (a,b) are intervals, where a is the lower edge and b the upper edge of the interval.

12 Interfaces for programmers

Advanced tweaks may be performed by using Scheme. If you are not familiar with Scheme, you may wish to read our Appendix B [Scheme tutorial], page 310.

12.1 Music functions

This section discusses how to create music functions within LilyPond.

12.1.1 Overview of music functions

Making a function which substitutes a variable into LilyPond code is easy. The general form of these functions is

```
function =
#(define-music-function (parser location var1 var2... )
                        (var1-type? var2-type?...)
  #{
    ...music...
  #})
```

where

argi	*i*th variable
argi-type?	type of variable
...music...	normal LilyPond input, using variables as `#$var1`.

There following input types may be used as variables in a music function. This list is not exhaustive; see other documentation specifically about Scheme for more variable types.

Input type	*argi-type?* **notation**
Integer	`integer?`
Float (decimal number)	`number?`
Text string	`string?`
Markup	`markup?`
Music expression	`ly:music?`
A pair of variables	`pair?`

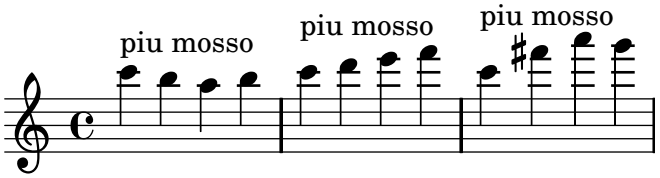
The `parser` and `location` argument are mandatory, and are used in some advanced situations. The `parser` argument is used to access to the value of another LilyPond variable. The `location` argument is used to set the "origin" of the music expression that is built by the music function, so that in case of a syntax error LilyPond can tell the user an appropriate place to look in the input file.

12.1.2 Simple substitution functions

Here is a simple example,

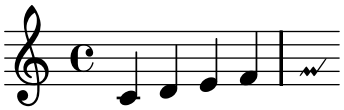
```
padText = #(define-music-function (parser location padding) (number?)
  #{
    \once \override TextScript #'padding = #$padding
  #})

\relative c''' {
  c4^"piu mosso" b a b
  \padText #1.8
  c4^"piu mosso" d e f
  \padText #2.6
```

```
  c4^"piu mosso" fis a g
}
```



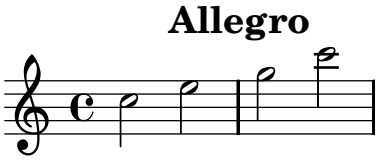
Music expressions may be substituted as well,

```
custosNote = #(define-music-function (parser location note)
                                     (ly:music?)
  #{
    \once \override Voice.NoteHead #'stencil =
      #ly:text-interface::print
    \once \override Voice.NoteHead #'text =
      \markup \musicglyph #"custodes.mensural.u0"
    \once \override Voice.Stem #'stencil = ##f
    $note
  #})

{ c' d' e' f' \custosNote g' }
```



Multiple variables may be used,

```
tempoMark = #(define-music-function (parser location padding marktext)
                                    (number? string?)
#{
  \once \override Score . RehearsalMark #'padding = $padding
  \once \override Score . RehearsalMark #'no-spacing-rods = ##t
  \mark \markup { \bold $marktext }
#})

\relative c'' {
c2 e
\tempoMark #3.0 #"Allegro"
g c
}
```



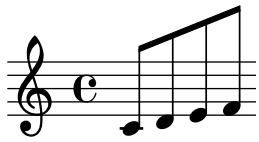
12.1.3 Paired substitution functions

Some `\override` commands require a pair of numbers (called a `cons cell` in Scheme). To pass these numbers into a function, either use a `pair?` variable, or insert the `cons` into the music function.

```
manualBeam =
#(define-music-function (parser location beg-end)
                        (pair?)
#{
  \once \override Beam #'positions = #$beg-end
#})

\relative {
  \manualBeam #'(3 . 6) c8 d e f
}
```

or

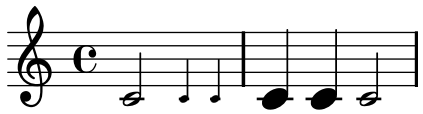
```
manualBeam =
#(define-music-function (parser location beg end)
                        (number? number?)
#{
  \once \override Beam #'positions = #(cons $beg $end)
#})

\relative {
  \manualBeam #3 #6 c8 d e f
}
```



12.1.4 Mathematics in functions

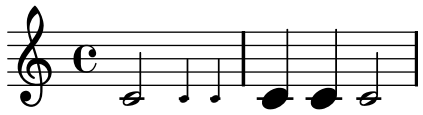
Music functions can involve Scheme programming in addition to simple substitution,

```
AltOn = #(define-music-function (parser location mag) (number?)
  #{ \override Stem #'length = #$(* 7.0 mag)
     \override NoteHead #'font-size =
       #$(inexact->exact (* (/ 6.0 (log 2.0)) (log mag))) #})

AltOff = {
  \revert Stem #'length
  \revert NoteHead #'font-size
}

{ c'2 \AltOn #0.5 c'4 c'
  \AltOn #1.5 c' c' \AltOff c'2 }
```



This example may be rewritten to pass in music expressions,

```
withAlt = #(define-music-function (parser location mag music) (number? ly:music?)
  #{ \override Stem #'length = #$(* 7.0 mag)
     \override NoteHead #'font-size =
       #$(inexact->exact (* (/ 6.0 (log 2.0)) (log mag)))
     $music
     \revert Stem #'length
     \revert NoteHead #'font-size #})

{ c'2 \withAlt #0.5 {c'4 c'}
  \withAlt #1.5 {c' c'} c'2 }
```



12.1.5 Void functions

A music function must return a music expression, but sometimes we may want to have a function which does not involve music (such as turning off Point and Click). To do this, we return a `void` music expression.

That is why the form that is returned is the `(make-music ...)`. With the `'void` property set to `#t`, the parser is told to actually disregard this returned music expression. Thus the important part of the void music function is the processing done by the function, not the music expression that is returned.

```
noPointAndClick =
#(define-music-function (parser location) ()
   (ly:set-option 'point-and-click #f)
   (make-music 'SequentialMusic 'void #t))
...
\noPointAndClick   % disable point and click
```

12.1.6 Functions without arguments

In most cases a function without arguments should be written with an identifier,

```
dolce = \markup{ \italic \bold dolce }
```

However, in rare cases it may be useful to create a music function without arguments,

```
displayBarNum =
#(define-music-function (parser location) ()
   (if (eq? #t (ly:get-option display-bar-numbers))
       #{ \once \override Score.BarNumber #'break-visibility = ##f #}
       #{#}))
```

To actually display bar numbers where this function is called, invoke lilypond with

```
lilypond -d display-bar-numbers FILENAME.ly
```

12.2 Programmer interfaces

This section contains information about mixing LilyPond and Scheme.

12.2.1 Input variables and Scheme

The input format supports the notion of variables: in the following example, a music expression is assigned to a variable with the name `traLaLa`.

```
traLaLa = { c'4 d'4 }
```

There is also a form of scoping: in the following example, the `\layout` block also contains a `traLaLa` variable, which is independent of the outer `\traLaLa`.

```
traLaLa = { c'4 d'4 }
\layout { traLaLa = 1.0 }
```

In effect, each input file is a scope, and all `\header`, `\midi`, and `\layout` blocks are scopes nested inside that toplevel scope.

Both variables and scoping are implemented in the GUILE module system. An anonymous Scheme module is attached to each scope. An assignment of the form

```
traLaLa = { c'4 d'4 }
```

is internally converted to a Scheme definition

```
(define traLaLa Scheme value of this music expression)
```

This means that input variables and Scheme variables may be freely mixed. In the following example, a music fragment is stored in the variable `traLaLa`, and duplicated using Scheme. The result is imported in a `\score` block by means of a second variable `twice`:

```
traLaLa = { c'4 d'4 }

%% dummy action to deal with parser lookahead
#(display "this needs to be here, sorry!")

#(define newLa (map ly:music-deep-copy
  (list traLaLa traLaLa)))
#(define twice
  (make-sequential-music newLa))

{ \twice }
```

Due to parser lookahead

In this example, the assignment happens after parser has verified that nothing interesting happens after `traLaLa = { ... }`. Without the dummy statement in the above example, the `newLa` definition is executed before `traLaLa` is defined, leading to a syntax error.

The above example shows how to 'export' music expressions from the input to the Scheme interpreter. The opposite is also possible. By wrapping a Scheme value in the function `ly:export`, a Scheme value is interpreted as if it were entered in LilyPond syntax. Instead of defining `\twice`, the example above could also have been written as

```
...
{ #(ly:export (make-sequential-music (list newLa))) }
```

Scheme code is evaluated as soon as the parser encounters it. To define some Scheme code in a macro (to be called later), use Section 12.1.5 [Void functions], page 274 or

```
#(define (nopc)
  (ly:set-option 'point-and-click #f))

...
#(nopc)
{ c'4 }
```

Bugs

Mixing Scheme and LilyPond identifiers is not possible with the `--safe` option.

12.2.2 Internal music representation

When a music expression is parsed, it is converted into a set of Scheme music objects. The defining property of a music object is that it takes up time. Time is a rational number that measures the length of a piece of music in whole notes.

A music object has three kinds of types:

- music name: Each music expression has a name. For example, a note leads to a `NoteEvent`, and `\simultaneous` leads to a `SimultaneousMusic`. A list of all expressions available is in the Program reference manual, under `Music expressions`.
- 'type' or interface: Each music name has several 'types' or interfaces, for example, a note is an `event`, but it is also a `note-event`, a `rhythmic-event`, and a `melodic-event`. All classes of music are listed in the Program reference, under `Music classes`.
- C++ object: Each music object is represented by an object of the C++ class `Music`.

The actual information of a music expression is stored in properties. For example, a `NoteEvent` has `pitch` and `duration` properties that store the pitch and duration of that note. A list of all properties available is in the internals manual, under `Music properties`.

A compound music expression is a music object that contains other music objects in its properties. A list of objects can be stored in the `elements` property of a music object, or a single 'child' music object in the `element` object. For example, `SequentialMusic` has its children in `elements`, and `GraceMusic` has its single argument in `element`. The body of a repeat is stored in the `element` property of `RepeatedMusic`, and the alternatives in `elements`.

12.3 Building complicated functions

This section explains how to gather the information necessary to create complicated music functions.

12.3.1 Displaying music expressions

When writing a music function it is often instructive to inspect how a music expression is stored internally. This can be done with the music function `\displayMusic`

```
{
  \displayMusic { c'4\f }
}
```

will display

```
(make-music
  'SequentialMusic
  'elements
  (list (make-music
          'EventChord
          'elements
          (list (make-music
```

```
                    'NoteEvent
                    'duration
                    (ly:make-duration 2 0 1 1)
                    'pitch
                    (ly:make-pitch 0 0 0))
                  (make-music
                    'AbsoluteDynamicEvent
                    'text
                    "f")))))
```

By default, LilyPond will print these messages to the console along with all the other messages. To split up these messages and save the results of `\display{STUFF}`, redirect the output to a file.

```
lilypond file.ly >display.txt
```

With a bit of reformatting, the above information is easier to read,

```
(make-music 'SequentialMusic
  'elements (list (make-music 'EventChord
                    'elements (list (make-music 'NoteEvent
                                      'duration (ly:make-duration 2 0 1 1)
                                      'pitch (ly:make-pitch 0 0 0))
                                    (make-music 'AbsoluteDynamicEvent
                                      'text "f")))))
```

A `{ ... }` music sequence has the name `SequentialMusic`, and its inner expressions are stored as a list in its `'elements` property. A note is represented as an `EventChord` expression, containing a `NoteEvent` object (storing the duration and pitch properties) and any extra information (in this case, an `AbsoluteDynamicEvent` with a `"f"` text property.

12.3.2 Music properties

The `NoteEvent` object is the first object of the `'elements` property of `someNote`.

```
someNote = c'
\displayMusic \someNote
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1 1)
          'pitch
          (ly:make-pitch 0 0 0))))
```

The `display-scheme-music` function is the function used by `\displayMusic` to display the Scheme representation of a music expression.

```
#(display-scheme-music (first (ly:music-property someNote 'elements)))
===>
(make-music
  'NoteEvent
  'duration
  (ly:make-duration 2 0 1 1)
  'pitch
  (ly:make-pitch 0 0 0))
```

Then the note pitch is accessed through the 'pitch property of the NoteEvent object,

```
#(display-scheme-music
   (ly:music-property (first (ly:music-property someNote 'elements))
                      'pitch))
===>
(ly:make-pitch 0 0 0)
```

The note pitch can be changed by setting this 'pitch property,

```
#(set! (ly:music-property (first (ly:music-property someNote 'elements))
                          'pitch)
       (ly:make-pitch 0 1 0)) ;; set the pitch to d'.
\displayLilyMusic \someNote
===>
d'
```

12.3.3 Doubling a note with slurs (example)

Suppose we want to create a function which translates input like "a'' into "a(a)''. We begin by examining the internal representation of the music we want to end up with.

```
\displayMusic{ a'( a') }
===>
(make-music
  'SequentialMusic
  'elements
  (list (make-music
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 5 0))
                (make-music
                  'SlurEvent
                  'span-direction
                  -1)))
        (make-music
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 5 0))
                (make-music
                  'SlurEvent
                  'span-direction
                  1)))))
```

The bad news is that the SlurEvent expressions must be added "inside" the note (or more precisely, inside the EventChord expression).

Now we examine the input,

```
(make-music
  'SequentialMusic
  'elements
  (list (make-music
          'EventChord
          'elements
          (list (make-music
                  'NoteEvent
                  'duration
                  (ly:make-duration 2 0 1 1)
                  'pitch
                  (ly:make-pitch 0 5 0))))))
```

So in our function, we need to clone this expression (so that we have two notes to build the sequence), add `SlurEvents` to the `'elements` property of each one, and finally make a `SequentialMusic` with the two `EventChords`.

```
doubleSlur = #(define-music-function (parser location note) (ly:music?)
         "Return: { note ( note ) }.
         `note'
 is supposed to be an EventChord."
         (let ((note2 (ly:music-deep-copy note)))
           (set! (ly:music-property note 'elements)
                 (cons (make-music 'SlurEvent 'span-direction -1)
                       (ly:music-property note 'elements)))
           (set! (ly:music-property note2 'elements)
                 (cons (make-music 'SlurEvent 'span-direction 1)
                       (ly:music-property note2 'elements)))
           (make-music 'SequentialMusic 'elements (list note note2))))
```

12.3.4 Adding articulation to notes (example)

The easy way to add articulation to notes is to merge two music expressions into one context, as explained in Section 9.2.2 [Creating contexts], page 219. However, suppose that we want to write a music function which does this.

A `$variable` inside the `#{...#}` notation is like using a regular `\variable` in classical LilyPond notation. We know that

```
{ \music -. -> }
```

will not work in LilyPond. We could avoid this problem by attaching the articulation to a fake note,

```
{ << \music s1*0-.-> }
```

but for the sake of this example, we will learn how to do this in Scheme. We begin by examining our input and desired output,

```
%  input
\displayMusic c4
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1 1)
```

```
          'pitch
          (ly:make-pitch -1 0 0))))
=====
%  desired output
\displayMusic c4->
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1 1)
          'pitch
          (ly:make-pitch -1 0 0))
        (make-music
          'ArticulationEvent
          'articulation-type
          "marcato")))
```

We see that a note (`c4`) is represented as an `EventChord` expression, with a `NoteEvent` expression in its elements list. To add a marcato articulation, an `ArticulationEvent` expression must be added to the elements property of the `EventChord` expression.

To build this function, we begin with

```
(define (add-marcato event-chord)
  "Add a marcato ArticulationEvent to the elements of `event-chord'
,
  which is supposed to be an EventChord expression."
  (let ((result-event-chord (ly:music-deep-copy event-chord)))
    (set! (ly:music-property result-event-chord 'elements)
          (cons (make-music 'ArticulationEvent
                  'articulation-type "marcato")
                (ly:music-property result-event-chord 'elements)))
    result-event-chord))
```

The first line is the way to define a function in Scheme: the function name is `add-marcato`, and has one variable called `event-chord`. In Scheme, the type of variable is often clear from its name. (this is good practice in other programming languages, too!)

```
"Add a marcato..."
```

is a description of what the function does. This is not strictly necessary, but just like clear variable names, it is good practice.

```
(let ((result-event-chord (ly:music-deep-copy event-chord)))
```

'`let`' is used to declare local variables. Here we use one local variable, named '`result-event-chord`', to which we give the value `(ly:music-deep-copy event-chord)`. '`ly:music-deep-copy`' is a function specific to LilyPond, like all functions prefixed by '`ly:`'. It is use to make a copy of a music expression. Here we copy '`event-chord` (the parameter of the function). Recall that our purpose is to add a marcato to an `EventChord` expression. It is better to not modify the `EventChord` which was given as an argument, because it may be used elsewhere.

Now we have a `result-event-chord`, which is a `NoteEventChord` expression and is a copy of `event-chord`. We add the marcato to its elements list property.

```
(set! place new-value)
```

Here, what we want to set (the "place") is the "elements" property of `result-event-chord` expression

```
(ly:music-property result-event-chord 'elements)
```

`ly:music-property` is the function used to access music properties (the `'elements`, `'duration`, `'pitch`, etc, that we see in the `\displayMusic` output above). The new value is the former elements property, with an extra item: the `MarcatoEvent` expression, which we copy from the `\displayMusic` output,

```
(cons (make-music 'ArticulationEvent
        'articulation-type "marcato")
      (ly:music-property result-event-chord 'elements))
```

'`cons`' is used to add an element to a list without modifying the original list. This is what we want: the same list as before, plus the new `ArticulationEvent` expression. The order inside the elements property is not important here.

Finally, once we have added the `MarcatoEvent` to its elements property, we can return `result-event-chord`, hence the last line of the function.

Now we transform the `add-marcato` function into a music function,

```
addMarcato = #(define-music-function (parser location event-chord)
                                     (ly:music?)
    "Add a marcato ArticulationEvent to the elements of `event-chord'
,
    which is supposed to be an EventChord expression."
    (let ((result-event-chord (ly:music-deep-copy event-chord)))
      (set! (ly:music-property result-event-chord 'elements)
            (cons (make-music 'ArticulationEvent
                    'articulation-type "marcato")
                  (ly:music-property result-event-chord 'elements)))
      result-event-chord))
```

We may verify that this music function works correctly,

```
\displayMusic \addMarcato c4
```

12.4 Markup programmer interface

Markups are implemented as special Scheme functions which produce a Stencil object given a number of arguments.

12.4.1 Markup construction in Scheme

The `markup` macro builds markup expressions in Scheme while providing a LilyPond-like syntax. For example,

```
(markup #:column (#:line (#:bold #:italic "hello" #:raise 0.4 "world")
                  #:bigger #:line ("foo" "bar" "baz")))
```

is equivalent to:

```
\markup \column { \line { \bold \italic "hello" \raise #0.4 "world" }
                  \bigger \line { foo bar baz } }
```

This example demonstrates the main translation rules between regular LilyPond markup syntax and Scheme markup syntax.

LilyPond	**Scheme**
`\markup markup1`	`(markup markup1)`

`\markup { markup1 markup2 ... }`	`(markup markup1 markup2 ... )`
`\command`	`#:command`
`\variable`	`variable`
`\center-align { ... }`	`#:center-align ( ... )`
`string`	`"string"`
`#scheme-arg`	`scheme-arg`

The whole Scheme language is accessible inside the `markup` macro. For example, You may use function calls inside `markup` in order to manipulate character strings. This is useful when defining new markup commands (see Section 12.4.3 [New markup command definition], page 282).

Bugs

The markup-list argument of commands such as `#:line`, `#:center`, and `#:column` cannot be a variable or the result of a function call.

```
(markup #:line (function-that-returns-markups))
```

is invalid. One should use the `make-line-markup`, `make-center-markup`, or `make-column-markup` functions instead,

```
(markup (make-line-markup (function-that-returns-markups)))
```

12.4.2 How markups work internally

In a markup like

```
\raise #0.5 "text example"
```

`\raise` is actually represented by the `raise-markup` function. The markup expression is stored as

```
(list raise-markup 0.5 (list simple-markup "text example"))
```

When the markup is converted to printable objects (Stencils), the `raise-markup` function is called as

```
(apply raise-markup
       \layout object
       list of property alists
       0.5
       the "text example" markup)
```

The `raise-markup` function first creates the stencil for the `text example` string, and then it raises that Stencil by 0.5 staff space. This is a rather simple example; more complex examples are in the rest of this section, and in '`scm/define-markup-commands.scm`'.

12.4.3 New markup command definition

New markup commands can be defined with the `define-markup-command` Scheme macro.

```
(define-markup-command (command-name layout props arg1 arg2 ...)
            (arg1-type? arg2-type? ...)
  ..command body..)
```

The arguments are

argi *i*th command argument

argi-type? a type predicate for the i*th* argument

layout the 'layout' definition

props a list of alists, containing all active properties.

As a simple example, we show how to add a `\smallcaps` command, which selects a small caps font. Normally we could select the small caps font,

```
\markup { \override #'(font-shape . caps) Text-in-caps }
```

This selects the caps font by setting the `font-shape` property to `#'caps` for interpreting `Text-in-caps`.

To make the above available as `\smallcaps` command, we must define a function using `define-markup-command`. The command should take a single argument of type `markup`. Therefore the start of the definition should read

```
(define-markup-command (smallcaps layout props argument) (markup?)
```

What follows is the content of the command: we should interpret the `argument` as a markup, i.e.,

```
(interpret-markup layout ... argument)
```

This interpretation should add `'(font-shape . caps)` to the active properties, so we substitute the following for the ... in the above example:

```
(cons (list '(font-shape . caps) ) props)
```

The variable `props` is a list of alists, and we prepend to it by cons'ing a list with the extra setting.

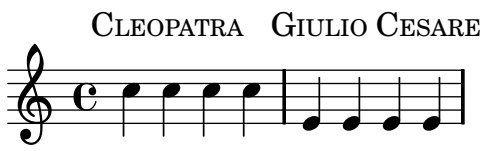
Suppose that we are typesetting a recitative in an opera and we would like to define a command that will show character names in a custom manner. Names should be printed with small caps and moved a bit to the left and top. We will define a `\character` command which takes into account the necessary translation and uses the newly defined `\smallcaps` command:

```
#(define-markup-command (character layout props name) (string?)
  "Print the character name in small caps, translated to the left and
  top.  Syntax: \\character #\"name\""
  (interpret-markup layout props
   (markup #:hspace 0 #:translate (cons -3 1) #:smallcaps name)))
```

There is one complication that needs explanation: texts above and below the staff are moved vertically to be at a certain distance (the `padding` property) from the staff and the notes. To make sure that this mechanism does not annihilate the vertical effect of our `#:translate`, we add an empty string (`#:hspace 0`) before the translated text. Now the `#:hspace 0` will be put above the notes, and the `name` is moved in relation to that empty string. The net effect is that the text is moved to the upper left.

The final result is as follows:

```
{
  c''^\markup \character #"Cleopatra"
  e'^\markup \character #"Giulio Cesare"
}
```



We have used the `caps` font shape, but suppose that our font does not have a small-caps variant. In that case we have to fake the small caps font by setting a string in upcase with the first letter a little larger:

```
#(define-markup-command (smallcaps layout props str) (string?)
  "Print the string argument in small caps."
  (interpret-markup layout props
   (make-line-markup
    (map (lambda (s)
          (if (= (string-length s) 0)
              s
              (markup #:large (string-upcase (substring s 0 1))
                      #:translate (cons -0.6 0)
                      #:tiny (string-upcase (substring s 1)))))
         (string-split str #\Space)))))
```

The `smallcaps` command first splits its string argument into tokens separated by spaces (`(string-split str #\Space)`); for each token, a markup is built with the first letter made large and upcased (`#:large (string-upcase (substring s 0 1))`), and a second markup built with the following letters made tiny and upcased (`#:tiny (string-upcase (substring s 1))`). As LilyPond introduces a space between markups on a line, the second markup is translated to the left (`#:translate (cons -0.6 0) ...`). Then, the markups built for each token are put in a line by `(make-line-markup ...)`. Finally, the resulting markup is passed to the `interpret-markup` function, with the `layout` and `props` arguments.

Note: there is now an internal command `\smallCaps` which can be used to set text in small caps. See Section 8.1.6 [Overview of text markup commands], page 174 for details.

12.5 Contexts for programmers

12.5.1 Context evaluation

Contexts can be modified during interpretation with Scheme code. The syntax for this is

```
\applyContext function
```

function should be a Scheme function taking a single argument, being the context to apply it to. The following code will print the current bar number on the standard output during the compile:

```
\applyContext
  #(lambda (x)
    (format #t "\nWe were called in barnumber ~a.\n"
     (ly:context-property x 'currentBarNumber)))
```

12.5.2 Running a function on all layout objects

The most versatile way of tuning an object is `\applyOutput`. Its syntax is

```
\applyOutput context proc
```

where *proc* is a Scheme function, taking three arguments.

When interpreted, the function *proc* is called for every layout object found in the context *context*, with the following arguments:

- the layout object itself,
- the context where the layout object was created, and
- the context where `\applyOutput` is processed.

In addition, the cause of the layout object, i.e., the music expression or object that was responsible for creating it, is in the object property `cause`. For example, for a note head, this is a `NoteHead` event, and for a `Stem` object, this is a `NoteHead` object.

Here is a function to use for `\applyOutput`; it blanks note-heads on the center-line:

```
(define (blanker grob grob-origin context)
 (if (and (memq (ly:grob-property grob 'interfaces)
                note-head-interface)
          (eq? (ly:grob-property grob 'staff-position) 0))
     (set! (ly:grob-property grob 'transparent) #t)))
```

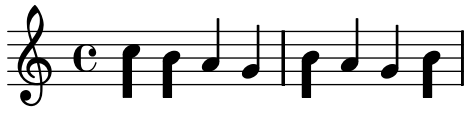
12.6 Scheme procedures as properties

Properties (like thickness, direction, etc.) can be set at fixed values with \override, e.g.

```
\override Stem #'thickness = #2.0
```

Properties can also be set to a Scheme procedure,

```
\override Stem #'thickness = #(lambda (grob)
    (if (= UP (ly:grob-property grob 'direction))
        2.0
        7.0))
c b a g b a g b
```



In this case, the procedure is executed as soon as the value of the property is requested during the formatting process.

Most of the typesetting engine is driven by such callbacks. Properties that typically use callbacks include

`stencil` The printing routine, that constructs a drawing for the symbol

`X-offset` The routine that sets the horizontal position

`X-extent` The routine that computes the width of an object

The procedure always takes a single argument, being the grob.

If routines with multiple arguments must be called, the current grob can be inserted with a grob closure. Here is a setting from `AccidentalSuggestion`,

```
(X-offset .
  ,(ly:make-simple-closure
    `(,+
        ,(ly:make-simple-closure
           (list ly:self-alignment-interface::centered-on-x-parent))
      ,(ly:make-simple-closure
           (list ly:self-alignment-interface::x-aligned-on-self)))))
```

In this example, both `ly:self-alignment-interface::x-aligned-on-self` and `ly:self-alignment-interface::centered-on-x-parent` are called with the grob as argument. The results are added with the `+` function. To ensure that this addition is properly executed, the whole thing is enclosed in `ly:make-simple-closure`.

In fact, using a single procedure as property value is equivalent to

```
(ly:make-simple-closure (ly:make-simple-closure (list proc)))
```

The inner `ly:make-simple-closure` supplies the grob as argument to *proc*, the outer ensures that result of the function is returned, rather than the `simple-closure` object.

13 Running LilyPond

This chapter details the technicalities of running LilyPond.

Some of these commands are run from the command-line. By "command-line", we mean the command line in the operating system. Windows users might be more familiar with the terms "DOS shell" or "command shell"; OSX users might be more familiar with the terms "terminal" or "console". OSX users should also consult Section 13.2 [Notes for the MacOS X app], page 289.

Describing how to use this part of an operating system is outside the scope of this manual; please consult other documentation on this topic if you are unfamiliar with the command-line.

13.1 Invoking lilypond

The `lilypond` executable may be called as follows from the command line.

```
lilypond [option]... file...
```

When invoked with a filename that has no extension, the '`.ly`' extension is tried first. To read input from stdin, use a dash (`-`) for *file*.

When '`filename.ly`' is processed it will produce '`filename.tex`' as output (or '`filename.ps`' for PostScript output). If '`filename.ly`' contains more than one `\score` block, then the rest of the scores will be output in numbered files, starting with '`filename-1.tex`'. Several files can be specified; they will each be processed independently.[1]

13.1.1 Command line options

The following options are supported:

`-e,--evaluate=`*expr*
: Evaluate the Scheme *expr* before parsing any '`.ly`' files. Multiple `-e` options may be given, they will be evaluated sequentially.

  The expression will be evaluated in the `guile-user` module, so if you want to use definitions in *expr*, use

  ```
  lilypond -e '(define-public a 42)'
  ```

  on the command-line, and include

  ```
  #(use-modules (guile-user))
  ```

  at the top of the `.ly` file.

`-f,--format=`*format*
: which formats should be written. Choices for `format` are `svg`, `ps`, `pdf`, `png`, `tex`, `dvi`.

  Example: `lilypond -fpng filename.ly`

`-b,--backend=`*format*
: the output format to use for the back-end. Choices for `format` are

  `tex` for TEX output, to be processed with LaTEX. If present, the file '`file.textmetrics`' is read to determine text extents.

  `texstr` dump text strings to '`.texstr`' file, which can be run through (La)TEX, resulting in a `.textmetrics` file, which contains the extents of strings of text. **Warning:** this functionality is currently missing due to heavy restructuring of the source code.

[1] The status of GUILE is not reset after processing a `.ly` file, so be careful not to change any system defaults from within Scheme.

`ps` for PostScript.

Postscript files include TTF, Type1 and OTF fonts. No subsetting of these fonts is done. When using oriental character sets, this can lead to huge files.

`eps` for encapsulated PostScript. This dumps every page (system) as a separate '`EPS`' file, without fonts, and as one collated '`EPS`' file with all pages (systems) including fonts.

This mode is used by default by lilypond-book.

`svg` for SVG (Scalable Vector Graphics). This dumps every page as a separate '`SVG`' file, with embedded fonts. You need a SVG viewer which supports embedded fonts, or a SVG viewer which is able to replace the embedded fonts with OTF fonts. Under Unix, you may use Inkscape (version 0.42 or later), after copying the OTF fonts in directory '`PATH/TO/share/lilypond/VERSION/fonts/otf/`' to '`~/.fonts/`'.

`scm` for a dump of the raw, internal Scheme-based drawing commands.

Example: `lilypond -bsvg filename.ly`

`-d,--define-default=`*var*`=`*val*

This sets the internal program option *var* to the Scheme value *val*. If *val* is not supplied, then *#t* is used. To switch off an option, `no-` may be prefixed to *var*, e.g.

```
-dno-point-and-click
```

is the same as

```
-dpoint-and-click='#f'
```

Another notable option is

```
-dpaper-size=\"letter\"
```

Note that the string must be enclosed in escaped quotes (`\"`).

Setting the `-dhelp` option will print a summary of the options available, and exit.

`-h,--help`

Show a summary of usage.

`-H,--header=FIELD`

Dump a header field to file BASENAME.FIELD

`--include, -I=`*directory*

Add *directory* to the search path for input files.

`-i,--init=`*file*

Set init file to *file* (default: '`init.ly`').

`-o,--output=`*FILE*

Set the default output file to *FILE*. The appropriate suffix will be added (ie `.pdf` for pdf, `.tex` for tex, etc).

`--ps` Generate PostScript.

`--dvi` Generate DVI files. In this case, the TEX backend should be specified, i.e., `-b tex`.

`--png` Generate pictures of each page, in PNG format. This implies `--ps`. The resolution in DPI of the image may be set with

```
-dresolution=110
```

`--pdf` Generate PDF. This implies `--ps`.

`--preview`
: Generate an output file containing the titles and the first system

`--no-pages`
: Do not generate the full pages. Useful in combination with `--preview`.

`-s,--safe`
: Do not trust the `.ly` input.

  When LilyPond formatting is available through a web server, either the `--safe` or the `--jail` option **MUST** be passed. The `--safe` option will prevent inline Scheme code from wreaking havoc, for example

```
#(system "rm -rf /")
{
  c4^#(ly:export (ly:gulp-file "/etc/passwd"))
}
```

  The `--safe` option works by evaluating in-line Scheme expressions in a special safe module. This safe module is derived from GUILE '`safe-r5rs`' module, but adds a number of functions of the LilyPond API. These functions are listed in '`scm/safe-lily.scm`'.

  In addition, `--safe` disallows `\include` directives and disables the use of backslashes in TEX strings.

  In `--safe` mode, it is not possible to import LilyPond variables into Scheme.

  `--safe` does *not* detect resource overuse. It is still possible to make the program hang indefinitely, for example by feeding cyclic data structures into the backend. Therefore, if using LilyPond on a publicly accessible webserver, the process should be limited in both CPU and memory usage.

  Note that `--safe` will prevent many useful LilyPond snippets from being compiled. For a softer but secure alternative you can use the `--jail` option.

`-j,--jail=`*`user,group,jail,dir`*
: Run LilyPond in a chroot jail.

  The `--jail` option provides a more flexible alternative to `--safe` when LilyPond formatting is available through a web server or whenever LilyPond executes externally provided sources.

  The `--jail` option works by changing the root of LilyPond to *jail* just before starting the actual compilation process. The user and group are then changed to match those provided, and the current directory is changed to *dir*. This setup guarantees that it is not possible (at least in theory) to escape from the jail. Note that for `--jail` to work LilyPond must be run as root, which is usually accomplished in a safe way using `sudo`.

  Setting up a jail is a slightly delicate matter, as we must be sure that LilyPond is able to find whatever it needs to compile the source *inside the jail*. A typical setup comprises the following items:

  Setting up a separate filesystem
  : A separate filesystem should be created for LilyPond, so that it can be mounted with safe options such as `noexec`, `nodev`, and `nosuid`. In this way, it is impossible to run executables or to write directly to a device from LilyPond. If you do not want to create a separate partition, just create a file of reasonable size and use it to mount a loop device. A separate filesystem also guarantees that LilyPond cannot write more space than it is allowed.

Setting up a separate user
: A separate user and group (say, '`lily`'/'`lily`') with low privileges should be used to run LilyPond inside the jail. There should be a single directory writable by this user, which should be passed in *dir*.

Preparing the jail
: LilyPond needs to read a number of files while running. All these files are to be copied into the jail, under the same path they appear in the real root filesystem. The entire content of the LilyPond installation (e.g., '`/usr/share/lilypond`') should be copied.

  If problems arise, the simplest way to trace them down is to run LilyPond using `strace`, which will allow you to determine which files are missing.

Running LilyPond
: In a jail mounted with `noexec` it is impossible to execute any external program. Therefore LilyPond must be run with a backend that does not require any such program. As we already mentioned, it must be also run with superuser privileges (which, of course, it will lose immediately), possibly using `sudo`. It is a good idea to limit the number of seconds of CPU time LilyPond can use (e.g., using `ulimit -t`), and, if your operating system supports it, the amount of memory that can be allocated.

`-v,--version`
: Show version information.

`-V,--verbose`
: Be verbose: show full paths of all files read, and give timing information.

`-w,--warranty`
: Show the warranty with which GNU LilyPond comes. (It comes with **NO WARRANTY**!)

13.1.2 Environment variables

`Lilypond` recognizes the following environment variables:

`LILYPOND_DATADIR`
: This specifies a directory where locale messages and data files will be looked up by default. The directory should contain subdirectories called '`ly/`', '`ps/`', '`tex/`', etc.

`LANG`
: This selects the language for the warning messages.

`LILYPOND_GC_YIELD`
: With this variable the memory footprint and performance can be adjusted. It is a percentage tunes memory management behavior. With higher values, the program uses more memory, with smaller values, it uses more CPU time. The default value is `70`.

13.2 Notes for the MacOS X app

The scripts (such as lilypond-book, convert-ly, abc2ly, and even lilypond itself) are also included inside MacOS X .app. They can be run from the command line by invoking them directly, e.g.

```
path/to/LilyPond.app/Contents/Resources/bin/lilypond
```

The same is true of the other scripts in that directory, including lilypond-book, convert-ly, abc2ly, etc.

Alternatively, you may create scripts which add the path automatically. Create a directory to store these scripts,

```
mkdir -p ~/bin
cd ~/bin
```

Create a file called `lilypond` which contains

```
exec path/to/LilyPond.app/Contents/Resources/bin/lilypond "$@"
```

Create similar files `lilypond-book`, `convert-ly`, and any other helper programs you use (`abc2ly`, `midi2ly`, etc). Simply replace the `bin/lilypond` with `bin/convert-ly` (or other program name) in the above file.

Make the file executable,

```
chmod u+x lilypond
```

Now, add this directory to your path. Modify (or create) a file called `.profile` in your home directory such that it contains

```
export PATH=$PATH:~/bin
```

This file should end with a blank line.

Note that *path/to* will generally be `/Applications/`.

13.3 Updating with `convert-ly`

The LilyPond input syntax is routinely changed to simplify it or improve it in different ways. As a side effect of this, the LilyPond interpreter often is no longer compatible with older input files. To remedy this, the program `convert-ly` can be used to deal with most of the syntax changes between LilyPond versions.

It uses `\version` statements in the input files to detect the old version number. In most cases, to upgrade your input file it is sufficient to run[2]

```
convert-ly -e myfile.ly
```

If there are no changes to myfile.ly and file called myfile.ly.NEW is created, then myfile.ly is already updated.

`convert-ly` always converts up to the last syntax change handled by it. This means that the `\version` number left in the file is usually lower than the version of `convert-ly` itself.

To upgrade LilyPond fragments in texinfo files, use

```
convert-ly --from=... --to=... --no-version *.itely
```

To see the changes in the LilyPond syntax between two versions, use

```
convert-ly --from=... --to=... -s
```

To upgrade many files at once, combine `convert-ly` with standard unix commands. This example will upgrade all `.ly` files in the current directory

```
for f in *.ly; do convert-ly -e $f; done;
```

In general, the program is invoked as follows:

```
convert-ly [option]... file...
```

The following options can be given:

`-e,--edit`
: Do an inline edit of the input file. Overrides `--output`.

`-f,--from=`*`from-patchlevel`*
: Set the version to convert from. If this is not set, `convert-ly` will guess this, on the basis of `\version` strings in the file.

[2] MacOS X users may execute this command under the menu entry '`Compile > Update syntax`'.

`-n,--no-version`
: Normally, `convert-ly` adds a `\version` indicator to the output. Specifying this option suppresses this.

`-s, --show-rules`
: Show all known conversions and exit.

`--to=`*`to-patchlevel`*
: Set the goal version of the conversion. It defaults to the latest available version.

`-h, --help`
: Print usage help.

Bugs

Not all language changes are handled. Only one output option can be specified. Automatically updating scheme and lilypond scheme interfaces is quite unlikely; be prepared to tweak scheme code manually.

```
There are a few things that the convert-ly cannot handle. Here's a list of
limitations
that the community has complained about.

This bug report structure has been chosen because convert-ly has a structure
that doesn't
allow to smoothly implement all needed changes. Thus this is just a wishlist,
placed
here for reference.

1.6->2.0:
 Doesn't always convert figured bass correctly, specifically things like {<
>}. Mats' comment on working around this:
   To be able to run convert-ly
   on it, I first replaced all occurencies of '{<' to some dummy like '{#'
   and similarly I replaced '>}' with '&}'. After the conversion, I could
   then change back from '{ #' to '{ <' and from '& }' to '> }'.
 Doesn't convert all text markup correctly. In the old markup syntax,
 it was possible to group a number of markup commands together within
parentheses, e.g.
   -#'((bold italic) "string")
   This will incorrectly be converted into
   -\markup{{\bold italic} "string"}
   instead of the correct
   -\markup{\bold \italic "string"}
2.0->2.2:
 Doesn't handle \partcombine
 Doesn't do \addlyrics => \lyricsto, this breaks some scores with multiple
stanzas.
2.0->2.4:
 \magnify isn't changed to \fontsize.
    - \magnify #m => \fontsize #f, where f = 6ln(m)/ln(2)
 remove-tag isn't changed.
    - \applyMusic #(remove-tag '. . .) => \keepWithTag #'. . .
 first-page-number isn't changed.
```

```
    - first-page-number no => print-first-page-number = ##f
 Line breaks in header strings aren't converted.
    - \\\\  as line break in \header strings => \markup \center-align <
      "First Line" "Second Line" >
 Crescendo and decrescendo terminators aren't converted.
    - \rced => \!
    - \rc => \!
2.2->2.4:
 \turnOff (used in \set Staff.VoltaBracket = \turnOff) is not properly
converted.
2.4.2->2.5.9
 \markup{ \center-align <{ ... }> } should be converted to:
 \markup{ \center-align {\line { ... }} }
 but now, \line is missing.
2.4->2.6
 Special LaTeX characters such as $~$ in text are not converted to UTF8.
2.8
 \score{} must now begin with a music expression.  Anything else
 (particularly \header{}) must come after the music.
```

13.4 Reporting bugs

If you have input that results in a crash or an erroneous output, then that is a bug. There is a list of current bugs on our google bug tracker,

http://code.google.com/p/lilypond/issues/list

If you have discovered a bug which is not listed, please report the bug by following the directions on

http://lilypond.org/web/devel/participating/bugs

Please construct submit Section 4.6 [Minimal examples], page 49 of bug reports. We do not have the resources to investigate reports which are not as small as possible.

13.5 Error messages

Different error messages can appear while compiling a file:

Warning Something looks suspect. If you are requesting something out of the ordinary then you will understand the message, and can ignore it. However, warnings usually indicate that something is wrong with the input file.

Error Something is definitely wrong. The current processing step (parsing, interpreting, or formatting) will be finished, but the next step will be skipped.

Fatal error
Something is definitely wrong, and LilyPond cannot continue. This happens rarely. The most usual cause is misinstalled fonts.

Scheme error
Errors that occur while executing Scheme code are caught by the Scheme interpreter. If running with the verbose option (`-V` or `--verbose`) then a call trace of the offending function call is printed.

Programming error
There was some internal inconsistency. These error messages are intended to help the programmers and debuggers. Usually, they can be ignored. Sometimes, they

come in such big quantities that they obscure other output. In this case, file a bug-report.

Aborted (core dumped)
: This signals a serious programming error that caused the program to crash. Such errors are considered critical. If you stumble on one, send a bug-report.

If warnings and errors can be linked to some part of the input file, then error messages have the following form

```
filename:lineno:columnno: message
offending input line
```

A line-break is inserted in the offending line to indicate the column where the error was found. For example,

```
test.ly:2:19: error: not a duration: 5:
  { c'4 e'5
              g' }
```

These locations are LilyPond's best guess about where the warning or error occurred, but (by their very nature) warnings and errors occur when something unexpected happens. If you can't see an error in the indicated line of your input file, try checking one or two lines above the indicated position.

13.6 Editor support

There is support from different editors for LilyPond.

Emacs
: Emacs has a '`lilypond-mode`', which provides keyword autocompletion, indentation, LilyPond specific parenthesis matching and syntax coloring, handy compile short-cuts and reading LilyPond manuals using Info. If '`lilypond-mode`' is not installed on your platform, then read the installation instructions.

VIM
: For VIM, a '`vimrc`' is supplied, along with syntax coloring tools. For more information, refer to the installation instructions.

LilyPondTool
: Created as a plugin for the jEdit text editor, LilyPondTool is the most feature-rich text-based tool for editing LilyPond scores. Its features include a Document Wizard with lyrics support to set up documents easier, and embedded PDF viewer with advanced point-and-click support. For screenshots, demos and installation instructions, visit `http://lilypondtool.organum.hu`

All these editors can be made to jump into the input file to the source of a symbol in the graphical output. See Section 13.7 [Point and click], page 293.

13.7 Point and click

Point and click lets you find notes in the input by clicking on them in the PDF viewer. This makes it easier to find input that causes some error in the sheet music.

When this functionality is active, LilyPond adds hyperlinks to the PDF file. These hyperlinks are sent to the web-browser, which opens a text-editor with the cursor in the right place.

To make this chain work, you should configure your PDF viewer to follow hyperlinks using the '`lilypond-invoke-editor`' script supplied with LilyPond.

For Xpdf on Unix, the following should be present in '`xpdfrc`'[3]

[3] On unix, this file is found either in '`/etc/xpdfrc`' or as '`.xpdfrc`' in your home directory.

```
urlCommand     "lilypond-invoke-editor %s"
```

The program '`lilypond-invoke-editor`' is a small helper program. It will invoke an editor for the special `textedit` URIs, and run a web browser for others. It tests the environment variable `EDITOR` for the following patterns,

`emacs` this will invoke

```
emacsclient --no-wait +line:column file
```

`vim` this will invoke

```
gvim --remote +:line:normchar file
```

`nedit` this will invoke

```
nc -noask +line file'
```

The environment variable `LYEDITOR` is used to override this. It contains the command line to start the editor, where `%(file)s`, `%(column)s`, `%(line)s` is replaced with the file, column and line respectively. The setting

```
emacsclient --no-wait +%(line)s:%(column)s %(file)s
```

for `LYEDITOR` is equivalent to the standard emacsclient invocation.

The point and click links enlarge the output files significantly. For reducing the size of PDF and PS files, point and click may be switched off by issuing

```
#(ly:set-option 'point-and-click #f)
```

in a '`.ly`' file. Alternately, you may pass this as an command-line option

```
lilypond -dno-point-and-click file.ly
```

14 lilypond-book: Integrating text and music

If you want to add pictures of music to a document, you can simply do it the way you would do with other types of pictures. The pictures are created separately, yielding PostScript output or PNG images, and those are included into a LaTeX or HTML document.

`lilypond-book` provides a way to automate this process: This program extracts snippets of music from your document, runs `lilypond` on them, and outputs the document with pictures substituted for the music. The line width and font size definitions for the music are adjusted to match the layout of your document.

This procedure may be applied to LaTeX, HTML, Texinfo or DocBook documents.

14.1 An example of a musicological document

Some texts contain music examples. These texts are musicological treatises, songbooks, or manuals like this. Such texts can be made by hand, simply by importing a PostScript figure into the word processor. However, there is an automated procedure to reduce the amount of work involved in HTML, LaTeX, Texinfo and DocBook documents.

A script called `lilypond-book` will extract the music fragments, format them, and put back the resulting notation. Here we show a small example for use with LaTeX. The example also contains explanatory text, so we will not comment on it further.

```
\documentclass[a4paper]{article}

\begin{document}

Documents for @command{lilypond-book} may freely mix music and text.
For example,

\begin{lilypond}
\relative c' {
  c2 g'2 \times 2/3 { f8 e d } c'2 g4
}
\end{lilypond}

Options are put in brackets.

\begin[fragment,quote,staffsize=26,verbatim]{lilypond}
  c'4 f16
\end{lilypond}

Larger examples can be put into a separate file, and introduced with
\verb+\lilypondfile+.

\lilypondfile[quote,noindent]{screech-boink.ly}

\end{document}
```

Under Unix, you can view the results as follows

```
cd input/tutorial
mkdir -p out/
lilypond-book --output=out --psfonts lilybook.tex
lilypond-book (GNU LilyPond) 2.6.0
Reading lilybook.tex...
```

```
..lots of stuff deleted..
Compiling out/lilybook.tex...
cd out
latex lilybook
lots of stuff deleted
xdvi lilybook
```

To convert the file into a PDF document, run the following commands

```
dvips -o -Ppdf -h lilybook.psfonts lilybook
ps2pdf lilybook.ps
```

If you are running latex in twocolumn mode, remember to add `-t landscape` to the dvips options.

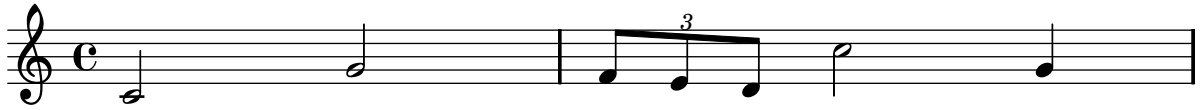
Running `lilypond-book` and `latex` creates a lot of temporary files, which would clutter up the working directory. To remedy this, use the `--output=dir` option. It will create the files in a separate subdirectory '`dir`'.

Running dvips will produce many warnings about fonts. They are not harmful; please ignore them.

Finally the result of the LaTeX example shown above.[1] This finishes the tutorial section.

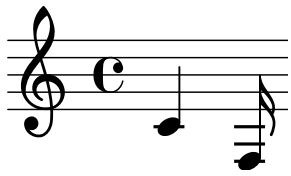
[1] This tutorial is processed with Texinfo, so the example gives slightly different results in layout.

Documents for `lilypond-book` may freely mix music and text. For example,



Options are put in brackets.

```
c'4 f16
```

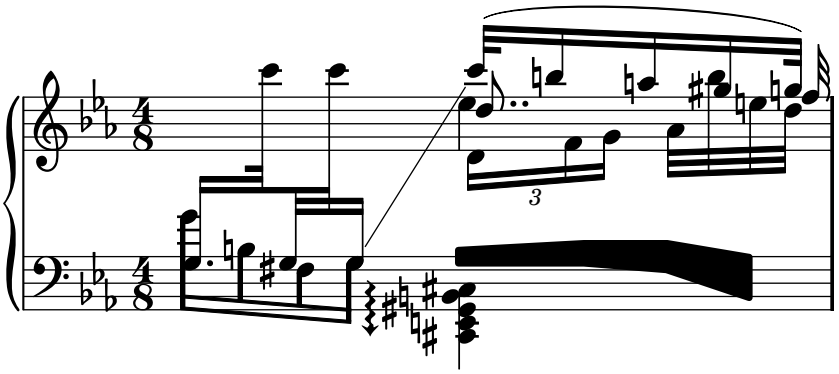


Larger examples can be put into a separate file, and introduced with `\lilypondfile`.

Screech and boink

Random complex notation

Han-Wen Nienhuys



14.2 Integrating LaTeX and music

LaTeX is the de-facto standard for publishing layouts in the exact sciences. It is built on top of the TeX typesetting engine, providing the best typography available anywhere.

See *The Not So Short Introduction to LaTeX* for an overview on how to use LaTeX.

Music is entered using

```
\begin[options,go,here]{lilypond}
  YOUR LILYPOND CODE
\end{lilypond}
```

or

```
\lilypondfile[options,go,here]{filename}
```

or

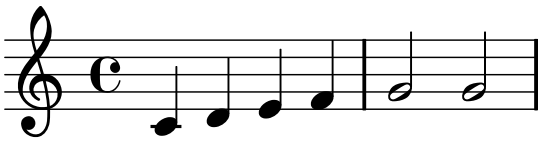
```
\lilypond{ YOUR LILYPOND CODE }
```

Running `lilypond-book` yields a file that can be further processed with LaTeX.

We show some examples here. The lilypond environment

```
\begin[quote,fragment,staffsize=26]{lilypond}
  c' d' e' f' g'2 g'2
\end{lilypond}
```

produces



The short version

```
\lilypond[quote,fragment,staffsize=11]{<c' e' g'>}
```

produces

Currently, you cannot include `{` or `}` within `\lilypond{}`, so this command is only useful with the `fragment` option.

The default line width of the music will be adjusted by examining the commands in the document preamble, the part of the document before `\begin{document}`. The `lilypond-book` command sends these to LaTeX to find out how wide the text is. The line width for the music fragments is then adjusted to the text width. Note that this heuristic algorithm can fail easily; in such cases it is necessary to use the `line-width` music fragment option.

Each snippet will call the following macros if they have been defined by the user:

`\preLilyPondExample` called before the music

`\postLilyPondExample` called after the music

`\betweenLilyPondSystem[1]` is called between systems if `lilypond-book` has split the snippet into several postscript files. It must be defined as taking one parameter and will be passed the number of files already included in this snippet. The default is to simply insert a `\linebreak`.

For printing the LaTeX document you need a DVI to PostScript translator like `dvips`. To use `dvips` to produce a PostScript file, add the following options to the `dvips` command line:

```
-o -Ppdf -h file.psfonts
```

where the *file*`psfonts` file is obtained from `lilypond-book`, See Section 14.7 [Invoking lilypond-book], page 303, for details. PDF can then be produced with a PostScript to PDF translator like `ps2pdf` (which is part of GhostScript). Running `dvips` will produce some warnings about fonts; these are harmless and may be ignored.

If you are running latex in twocolumn mode, remember to add `-t landscape` to the dvips options.

Sometimes it is useful to display music elements (such as ties and slurs) as if they continued after the end of the fragment. This can be done by breaking the staff and suppressing inclusion of the rest of the lilypond output.

In LaTeX, define `\betweenLilyPondSystem` in such a way that inclusion of other systems is terminated once the required number of systems are included. Since `\betweenLilypondSystem` is first called **after** the first system, including only the first system is trivial.

```
\def\betweenLilyPondSystem#1{\endinput}

\begin[fragment]{lilypond}
  c'1\( e'( c'~ \break c' d) e f\)
\end{lilypond}
```

If a greater number of systems is requested, a TeX conditional must be used before the `\endinput`. In this example, replace "2" by the numer of systems you want in the output,

```
\def\betweenLilyPondSystem#1{
    \ifnum##1<2\else\endinput\fi
}
```

Remember that the definition of `\betweenLilyPondSystem` is effective until TeX quits the current group (such as the LaTeX environment) or is overridden by another definition (which is, in most cases, for the rest of the document). To reset your definition, write

```
\let\betweenLilyPondSystem\undefined
```

in your LaTeX source.

This may be simplified by defining a TeX macro

```
\def\onlyFirstNSystems#1{
    \def\betweenLilyPondSystem##1{\ifnum##1<#1\else\endinput\fi}
}
```

and then saying only how many systems you want before each fragment,

```
\onlyFirstNSystems{3}
\begin{lilypond}...\end{lilypond}
\onlyFirstNSystems{1}
\begin{lilypond}...\end{lilypond}
```

14.3 Integrating Texinfo and music

Texinfo is the standard format for documentation of the GNU project. An example of a Texinfo document is this manual. The HTML, PDF, and Info versions of the manual are made from the Texinfo document.

In the input file, music is specified with

```
@lilypond[options,go,here]
  YOUR LILYPOND CODE
@end lilypond
```

or

```
@lilypond[options,go,here]{ YOUR LILYPOND CODE }
```

or

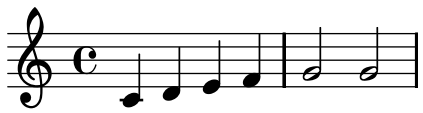
```
@lilypondfile[options,go,here]{filename}
```

When `lilypond-book` is run on it, this results in a Texinfo file (with extension '`.texi`') containing `@image` tags for HTML and info output. For the printed edition, the raw TEX output of LilyPond is included in the main document.

We show two simple examples here. A `lilypond` environment

```
@lilypond[fragment]
c' d' e' f' g'2 g'
@end lilypond
```

produces



The short version

```
@lilypond[fragment,staffsize=11]{<c' e' g'>}
```

produces

Contrary to LaTEX, `@lilypond{...}` does not generate an in-line image. It always gets a paragraph of its own.

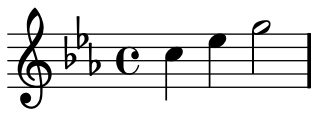
When using the Texinfo output format, `lilypond-book` also generates bitmaps of the music (in PNG format), so you can make an HTML document with embedded music.

14.4 Integrating HTML and music

Music is entered using

```
<lilypond fragment relative=2>
\key c \minor c4 es g2
</lilypond>
```

`lilypond-book` then produces an HTML file with appropriate image tags for the music fragments:



For inline pictures, use `<lilypond ... />`, where the options are separated by a colon from the music, for example

```
Some music in <lilypond relative=2: a b c/> a line of text.
```

To include separate files, say

```
<lilypondfile option1 option2 ...>filename</lilypondfile>
```

14.5 Integrating DocBook and music

For inserting LilyPond snippets it is good to keep the conformity of our DocBook document, thus allowing us to use DocBook editors, validation etc. So we don't use custom tags, only specify a convention based on the standard DocBook elements.

Common conventions

For inserting all type of snippets we use the `mediaobject` and `inlinemediaobject` element, so our snippets can be formatted inline or not inline. The snippet formatting options are always provided in the `role` property of the innermost element (see in next sections). Tags are chosen to allow DocBook editors format the content gracefully. The DocBook files to be processed with `lilypond-book` should have the extension '`.lyxml`'.

Including a LilyPond file

This is the most simple case. We must use the '`.ly`' extension for the included file, and insert it as a standard `imageobject`, with the following structure:

```
<mediaobject>
  <imageobject>
    <imagedata fileref="music1.ly" role="printfilename" />
  </imageobject>
</mediaobject>
```

Note that you can use mediaobject or inlinemediaobject as the outermost element as you wish.

Including LilyPond code

Including LilyPond code is possible by using a `programlisting`, where the language is set to `lilypond` with the following structure:

```
<inlinemediaobject>
  <textobject>
    <programlisting language="lilypond" role="fragment verbatim staffsize=16 ragged-ri
\context Staff \with {
  \remove Time_signature_engraver
  \remove Clef_engraver}
  { c4( fis) }
    </programlisting>
  </textobject>
</inlinemediaobject>
```

As you can see, the outermost element is a `mediaobject` or `inlinemediaobject`, and there is a `textobject` containing the `programlisting` inside.

Processing the DocBook document

Running `lilypond-book` on our '`.lyxml`' file will create a valid DocBook document to be further processed with '`.xml`' extension. If you use dblatex, it will create a PDF file from this document automatically. For HTML (HTML Help, JavaHelp etc.) generation you can use the official DocBook XSL stylesheets, however, it is possible that you have to make some customization for it.

14.6 Music fragment options

In the following, a "LilyPond command" refers to any command described in the previous sections which is handled by `lilypond-book` to produce a music snippet. For simplicity, LilyPond commands are only shown in LaTeX syntax.

Note that the option string is parsed from left to right; if an option occurs multiple times, the last one is taken.

The following options are available for LilyPond commands:

`staffsize=`*ht*
: Set staff size to *ht*, which is measured in points.

`ragged-right`
: Produce ragged-right lines with natural spacing (i.e., `ragged-right = ##t` is added to the LilyPond snippet). This is the default for the `\lilypond{}` command if no `line-width` option is present. It is also the default for the `lilypond` environment if the `fragment` option is set, and no line width is explicitly specified.

`packed`
: Produce lines with packed spacing (i.e., `packed = ##t` is added to the LilyPond snippet).

`line-width`
`line-width=`*size*`\`*unit*
: Set line width to *size*, using *unit* as units. *unit* is one of the following strings: `cm`, `mm`, `in`, or `pt`. This option affects LilyPond output (this is, the staff length of the music snippet), not the text layout.

  If used without an argument, set line width to a default value (as computed with a heuristic algorithm).

  If no `line-width` option is given, `lilypond-book` tries to guess a default for `lilypond` environments which don't use the `ragged-right` option.

`notime`
: Do not print the time signature, and turns off the timing (key signature, bar lines) in the score.

`fragment`
: Make `lilypond-book` add some boilerplate code so that you can simply enter, say,

```
c'4
```

  without `\layout`, `\score`, etc.

`nofragment`
: Don't add additional code to complete LilyPond code in music snippets. Since this is the default, `nofragment` is redundant normally.

`indent=`*size*`\`*unit*
: Set indentation of the first music system to *size*, using *unit* as units. *unit* is one of the following strings: `cm`, `mm`, `in`, or `pt`. This option affects LilyPond, not the text layout.

`noindent`
: Set indentation of the first music system to zero. This option affects LilyPond, not the text layout. Since no indentation is the default, `noindent` is redundant normally.

`quote`
: Reduce line length of a music snippet by 2∗0.4 in and put the output into a quotation block. The value '0.4 in' can be controlled with the `exampleindent` option.

`exampleindent`
: Set the amount by which the `quote` option indents a music snippet.

`relative`
`relative=`*n*
: Use relative octave mode. By default, notes are specified relative to middle C. The optional integer argument specifies the octave of the starting note, where the default `1` is middle C.

LilyPond also uses `lilypond-book` to produce its own documentation. To do that, some more obscure music fragment options are available.

`verbatim` The argument of a LilyPond command is copied to the output file and enclosed in a verbatim block, followed by any text given with the `intertext` option (not implemented yet); then the actual music is displayed. This option does not work well with `\lilypond{}` if it is part of a paragraph.

`texidoc` (Only for Texinfo output.) If `lilypond` is called with the '`--header=texidoc`' option, and the file to be processed is called '`foo.ly`', it creates a file '`foo.texidoc`' if there is a `texidoc` field in the `\header`. The `texidoc` option makes `lilypond-book` include such files, adding its contents as a documentation block right before the music snippet.

Assuming the file '`foo.ly`' contains

```
\header {
  texidoc = "This file demonstrates a single note."
}
{ c'4 }
```

and we have this in our Texinfo document '`test.texinfo`'

```
@lilypondfile[texidoc]{foo.ly}
```

the following command line gives the expected result

```
lilypond-book --process="lilypond --format=tex --tex \
              --header=texidoc test.texinfo
```

Most LilyPond test documents (in the '`input`' directory of the distribution) are small '`.ly`' files which look exactly like this.

`printfilename`
If a LilyPond input file is included with `\lilypondfile`, print the file name right before the music snippet. For HTML output, this is a link.

`fontload` This option includes fonts in all of the generated EPS-files for this snippet. This should be used if the snippet uses any font that LaTeX cannot find on its own.

14.7 Invoking `lilypond-book`

`lilypond-book` produces a file with one of the following extensions: '`.tex`', '`.texi`', '`.html`' or '`.xml`', depending on the output format. All of '`.tex`', '`.texi`' and '`.xml`' files need further processing.

`lilypond-book` can also create a PSFONTS file, which is required by `dvips` to produce Postscript and PDF files.

To produce PDF output from the lilypond-book file (here called `yourfile.lytex`) via LaTeX, you should do

```
lilypond-book --psfonts yourfile.lytex
latex yourfile.tex
dvips -o -h yourfile.psfonts -Ppdf yourfile.dvi
ps2pdf yourfile.ps
```

The '`.dvi`' file created by this process will not contain noteheads. This is normal; if you follow the instructions, they will be included in the '`.ps`' and '`.pdf`' files.

To produce a PDF file through PDF(La)TeX, use

```
lilypond-book --pdf yourfile.pdftex
pdflatex yourfile.tex
```

To produce a Texinfo document (in any output format), follow the normal procedures for Texinfo (this is, either call `texi2dvi` or `makeinfo`, depending on the output format you want to create). See the documentation of Texinfo for further details.

`lilypond-book` accepts the following command line options:

`-f` *format*
`--format=`*format*
: Specify the document type to process: `html`, `latex`, `texi` (the default) or `docbook`. If this option is missing, `lilypond-book` tries to detect the format automatically.

  The `texi` document type produces a Texinfo file with music fragments in the DVI output only. For getting images in the HTML version, the format `texi-html` must be used instead.

  [Note: Currently, `texi` is the same as `texi-html`.]

`-F` *filter*
`--filter=`*filter*
: Pipe snippets through *filter*. `lilypond-book` will not –filter and –process at the same time.

  Example:

  ```
  lilypond-book --filter='convert-ly --from=2.0.0 -' my-book.tely
  ```

`-h`
`--help`
: Print a short help message.

`-I` *dir*
`--include=`*dir*
: Add *dir* to the include path.

`-o` *dir*
`--output=`*dir*
: Place generated files in directory *dir*. Running `lilypond-book` generates lots of small files that LilyPond will process. To avoid all that garbage in the source directory use the '`--output`' command line option, and change to that directory before running `latex` or `makeinfo`:

  ```
  lilypond-book --output=out yourfile.lytex
  cd out
  ...
  ```

`--padding=`*amount*
: Pad EPS boxes by this much. *amount* is measured in milimeters, and is 3.0 by default. This option should be used if the lines of music stick out of the right margin.

  The width of a tightly clipped systems can vary, due to notation elements that stick into the left margin, such as bar numbers and instrument names. This option will shorten each line and move each line to the right by the same amount.

`-P` *process*
`--process=`*command*
: Process LilyPond snippets using *command*. The default command is `lilypond`. `lilypond-book` will not –filter and –process at the same time.

`--psfonts`
: extract all PostScript fonts into '*file*`.psfonts`' for dvips. This is necessary for `dvips -h` *file*`.psfonts`.

`-V`
`--verbose`
: Be verbose.

`-v`
`--version`
: Print version information.

Bugs

The Texinfo command `@pagesizes` is not interpreted. Similarly, LaTeX commands that change margins and line widths after the preamble are ignored.

Only the first `\score` of a LilyPond block is processed.

14.8 Filename extensions

You can use any filename extension for the input file, but if you do not use the recommended extension for a particular format you may need to manually specify the output format. See Section 14.7 [Invoking lilypond-book], page 303, for details. Otherwise, `lilypond-book` automatically selects the output format based on the input filename's extension.

extension	output format
'`.html`'	HTML
'`.itely`'	Texinfo
'`.latex`'	LaTeX
'`.lytex`'	LaTeX
'`.lyxml`'	DocBook
'`.tely`'	Texinfo
'`.tex`'	LaTeX
'`.texi`'	Texinfo
'`.texinfo`'	Texinfo
'`.xml`'	HTML

14.9 Many quotes of a large score

If you need to quote many fragments of a large score, you can also use the clip systems feature, see Section 10.1.5 [Extracting fragments of notation], page 237.

14.10 Inserting LilyPond output into OpenOffice.org

LilyPond notation can be added to OpenOffice.org with OOoLilyPond

14.11 Inserting LilyPond output into other programs

To insert LilyPond output in other programs, use `lilypond` instead of `lilypond-book`. Each example must be created individually and added to the document; consult the documentation for that program. Most programs will be able to insert lilypond output in '`PNG`', '`EPS`', or '`PDF`' formats.

To reduce the white space around your lilypond score, use the following options

```
\paper{
  indent=0\mm
  line-width=120\mm
  oddFooterMarkup=##f
  oddHeaderMarkup=##f
  bookTitleMarkup = ##f
  scoreTitleMarkup = ##f
}

{ c1 }
```

To produce a useful '`eps`' file, use

```
lilypond -b eps -dno-gs-load-fonts -dinclude-eps-fonts   myfile.ly
```

15 Converting from other formats

Music can be entered also by importing it from other formats. This chapter documents the tools included in the distribution to do so. There are other tools that produce LilyPond input, for example GUI sequencers and XML converters. Refer to the website for more details.

These are separate programs from lilypond itself, and are run on the command-line. By "command-line", we mean the command line in the operating system. Windows users might be more familiar with the terms "DOS shell" or "command shell"; OSX users might be more familiar with the terms "terminal" or "console". OSX users should also consult Section 13.2 [Notes for the MacOS X app], page 289.

Describing how to use this part of an operating system is outside the scope of this manual; please consult other documentation on this topic if you are unfamiliar with the command-line.

15.1 Invoking `midi2ly`

`midi2ly` translates a Type 1 MIDI file to a LilyPond source file.

MIDI (Music Instrument Digital Interface) is a standard for digital instruments: it specifies cabling, a serial protocol and a file format. The MIDI file format is a de facto standard format for exporting music from other programs, so this capability may come in useful when importing files from a program that has a convertor for a direct format.

`midi2ly` converts tracks into `Staff` and channels into `Voice` contexts. Relative mode is used for pitches, durations are only written when necessary.

It is possible to record a MIDI file using a digital keyboard, and then convert it to '`.ly`'. However, human players are not rhythmically exact enough to make a MIDI to LY conversion trivial. When invoked with quantizing (`-s` and `-d` options) `midi2ly` tries to compensate for these timing errors, but is not very good at this. It is therefore not recommended to use `midi2ly` for human-generated midi files.

It is invoked from the command-line as follows,

```
midi2ly [option]... midi-file
```

Note that by "command-line", we mean the command line of the operating system. See Chapter 15 [Converting from other formats], page 306 for more information about this.

The following options are supported by `midi2ly`.

`-a, --absolute-pitches`
: Print absolute pitches.

`-d, --duration-quant=`*DUR*
: Quantize note durations on *DUR*.

`-e, --explicit-durations`
: Print explicit durations.

`-h,--help`
: Show summary of usage.

`-k, --key=`*acc*`[:`*minor*`]`
: Set default key. *acc* > 0 sets number of sharps; *acc* < 0 sets number of flats. A minor key is indicated by ":1".

`-o, --output=`*file*
: Write output to *file*.

`-s, --start-quant=`*DUR*
: Quantize note starts on DUR.

`-t, --allow-tuplet=`*`DUR*NUM/DEN`*
: Allow tuplet durations *DUR*NUM/DEN*.

`-V, --verbose`
: Be verbose.

`-v, --version`
: Print version number.

`-w, --warranty`
: Show warranty and copyright.

`-x, --text-lyrics`
: Treat every text as a lyric.

Bugs

Overlapping notes in an arpeggio will not be correctly rendered. The first note will be read and the others will be ignored. Set them all to a single duration and add phrase markings or pedal indicators.

15.2 Invoking `etf2ly`

ETF (Enigma Transport Format) is a format used by Coda Music Technology's Finale product. `etf2ly` will convert part of an ETF file to a ready-to-use LilyPond file.

It is invoked from the command-line as follows.

```
etf2ly [option]... etf-file
```

Note that by "command-line", we mean the command line of the operating system. See Chapter 15 [Converting from other formats], page 306 for more information about this.

The following options are supported by `etf2ly`:

`-h,--help`
: this help

`-o,--output=FILE`
: set output filename to FILE

`-v,--version`
: version information

Bugs

The list of articulation scripts is incomplete. Empty measures confuse `etf2ly`. Sequences of grace notes are ended improperly.

15.3 Invoking `musicxml2ly`

MusicXML is an XML dialect for representing music notation.

`musicxml2ly` extracts the notes from part-wise MusicXML files, and writes it to a .ly file. It is invoked from the command-line.

Note that by "command-line", we mean the command line of the operating system. See Chapter 15 [Converting from other formats], page 306 for more information about this.

The following options are supported by `musicxml2ly`:

`-h,--help`
: print usage and option summary.

`-o,--output=`*file*
: set output filename to *file*. (default: print to stdout)

`-v,--version`
: print version information.

15.4 Invoking `abc2ly`

ABC is a fairly simple ASCII based format. It is described at the ABC site:

http://www.walshaw.plus.com/abc/abc2mtex/abc.txt.

`abc2ly` translates from ABC to LilyPond. It is invoked as follows:

```
abc2ly [option]... abc-file
```

The following options are supported by `abc2ly`:

`-h,--help`
: this help

`-o,--output=`*file*
: set output filename to *file*.

`-v,--version`
: print version information.

There is a rudimentary facility for adding LilyPond code to the ABC source file. If you say:

```
%%LY voices \set autoBeaming = ##f
```

This will cause the text following the keyword "voices" to be inserted into the current voice of the LilyPond output file.

Similarly,

```
%%LY slyrics more words
```

will cause the text following the "slyrics" keyword to be inserted into the current line of lyrics.

Bugs

The ABC standard is not very "standard". For extended features (e.g., polyphonic music) different conventions exist.

Multiple tunes in one file cannot be converted.

ABC synchronizes words and notes at the beginning of a line; `abc2ly` does not.

`abc2ly` ignores the ABC beaming.

15.5 Generating LilyPond files

LilyPond itself does not come with support for any other formats, but there are some external tools that also generate LilyPond files.

These tools include

- Denemo, a graphical score editor.
- Rumor, a realtime monophonic MIDI to LilyPond converter.
- lyqi, an Emacs major mode.
- xml2ly, which imports MusicXML
- NoteEdit which imports MusicXML
- Rosegarden, which imports MIDI
- FOMUS, a LISP library to generate music notation

Appendix A Literature list

If you need to know more about music notation, here are some interesting titles to read.

Ignatzek 1995
: Klaus Ignatzek, Die Jazzmethode für Klavier. Schott's Söhne 1995. Mainz, Germany ISBN 3-7957-5140-3.

  A tutorial introduction to playing Jazz on the piano. One of the first chapters contains an overview of chords in common use for Jazz music.

Gerou 1996
: Tom Gerou and Linda Lusk, Essential Dictionary of Music Notation. Alfred Publishing, Van Nuys CA ISBN 0-88284-768-6.

  A concise, alphabetically ordered list of typesetting and music (notation) issues, covering most of the normal cases.

Read 1968
: Gardner Read, Music Notation: A Manual of Modern Practice. Taplinger Publishing, New York (2nd edition).

  A standard work on music notation.

Ross 1987
: Ted Ross, Teach yourself the art of music engraving and processing. Hansen House, Miami, Florida 1987.

  This book is about music engraving, i.e., professional typesetting. It contains directions on stamping, use of pens and notational conventions. The sections on reproduction technicalities and history are also interesting.

Schirmer 2001
: The G.Schirmer/AMP Manual of Style and Usage. G.Schirmer/AMP, NY, 2001. (This book can be ordered from the rental department.)

  This manual specifically focuses on preparing print for publication by Schirmer. It discusses many details that are not in other, normal notation books. It also gives a good idea of what is necessary to bring printouts to publication quality.

Stone 1980
: Kurt Stone, Music Notation in the Twentieth Century. Norton, New York 1980.

  This book describes music notation for modern serious music, but starts out with a thorough overview of existing traditional notation practices.

The source archive includes a more elaborate BibTeX bibliography of over 100 entries in '`Documentation/bibliography/`'. It is also available online from the website.

Appendix B Scheme tutorial

LilyPond uses the Scheme programming language, both as part of the input syntax, and as internal mechanism to glue modules of the program together. This section is a very brief overview of entering data in Scheme. If you want to know more about Scheme, see http://www.schemers.org.

The most basic thing of a language is data: numbers, character strings, lists, etc. Here is a list of data types that are relevant to LilyPond input.

Booleans Boolean values are True or False. The Scheme for True is `#t` and False is `#f`.

Numbers Numbers are entered in the standard fashion, `1` is the (integer) number one, while `-1.5` is a floating point number (a non-integer number).

Strings Strings are enclosed in double quotes,

```
"this is a string"
```

Strings may span several lines

```
"this
is
a string"
```

Quotation marks and newlines can also be added with so-called escape sequences. The string `a said "b"` is entered as

```
"a said \"b\""
```

Newlines and backslashes are escaped with `\n` and `\\` respectively.

In a music file, snippets of Scheme code are introduced with the hash mark `#`. So, the previous examples translated in LilyPond are

```
##t ##f
#1 #-1.5
#"this is a string"
#"this
is
a string"
```

For the rest of this section, we will assume that the data is entered in a music file, so we add `#`s everywhere.

Scheme can be used to do calculations. It uses *prefix* syntax. Adding 1 and 2 is written as `(+ 1 2)` rather than the traditional $1 + 2$.

```
#(+ 1 2)
  ⇒ #3
```

The arrow ⇒ shows that the result of evaluating `(+ 1 2)` is `3`. Calculations may be nested; the result of a function may be used for another calculation.

```
#(+ 1 (* 3 4))
  ⇒ #(+ 1 12)
  ⇒ #13
```

These calculations are examples of evaluations; an expression like `(* 3 4)` is replaced by its value `12`. A similar thing happens with variables. After defining a variable

```
twelve = #12
```

variables can also be used in expressions, here

```
twentyFour = #(* 2 twelve)
```

the number 24 is stored in the variable `twentyFour`. The same assignment can be done in completely in Scheme as well,

```
#(define twentyFour (* 2 twelve))
```

The *name* of a variable is also an expression, similar to a number or a string. It is entered as

```
#'twentyFour
```

The quote mark ' prevents the Scheme interpreter from substituting 24 for the twentyFour. Instead, we get the name twentyFour.

This syntax will be used very frequently, since many of the layout tweaks involve assigning (Scheme) values to internal variables, for example

```
\override Stem #'thickness = #2.6
```

This instruction adjusts the appearance of stems. The value 2.6 is put into the thickness variable of a Stem object. thickness is measured relative to the thickness of staff lines, so these stem lines will be 2.6 times the width of staff lines. This makes stems almost twice as thick as their normal size. To distinguish between variables defined in input files (like twentyFour in the example above) and variables of internal objects, we will call the latter "properties" and the former "identifiers." So, the stem object has a thickness property, while twentyFour is an identifier.

Two-dimensional offsets (X and Y coordinates) as well as object sizes (intervals with a left and right point) are entered as pairs. A pair[1] is entered as (first . second) and, like symbols, they must be quoted,

```
\override TextScript #'extra-offset = #'(1 . 2)
```

This assigns the pair (1, 2) to the extra-offset property of the TextScript object. These numbers are measured in staff-spaces, so this command moves the object 1 staff space to the right, and 2 spaces up.

The two elements of a pair may be arbitrary values, for example

```
#'(1 . 2)
#'(#t . #f)
#'("blah-blah" . 3.14159265)
```

A list is entered by enclosing its elements in parentheses, and adding a quote. For example,

```
#'(1 2 3)
#'(1 2 "string" #f)
```

We have been using lists all along. A calculation, like (+ 1 2) is also a list (containing the symbol + and the numbers 1 and 2). Normally lists are interpreted as calculations, and the Scheme interpreter substitutes the outcome of the calculation. To enter a list, we stop the evaluation. This is done by quoting the list with a quote ' symbol. So, for calculations do not use a quote.

Inside a quoted list or pair, there is no need to quote anymore. The following is a pair of symbols, a list of symbols and a list of lists respectively,

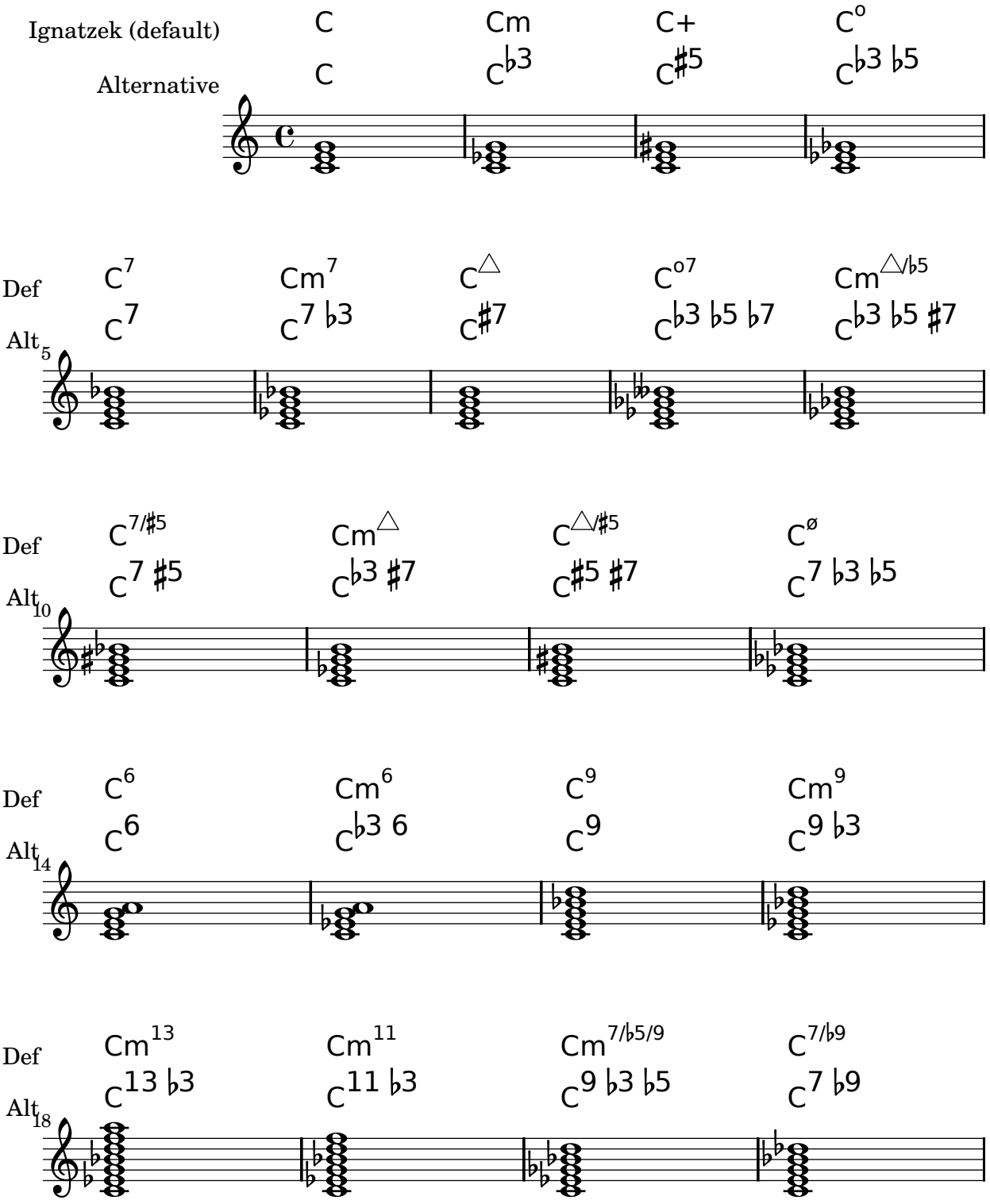
```
#'(stem . head)
#'(staff clef key-signature)
#'((1) (2))
```

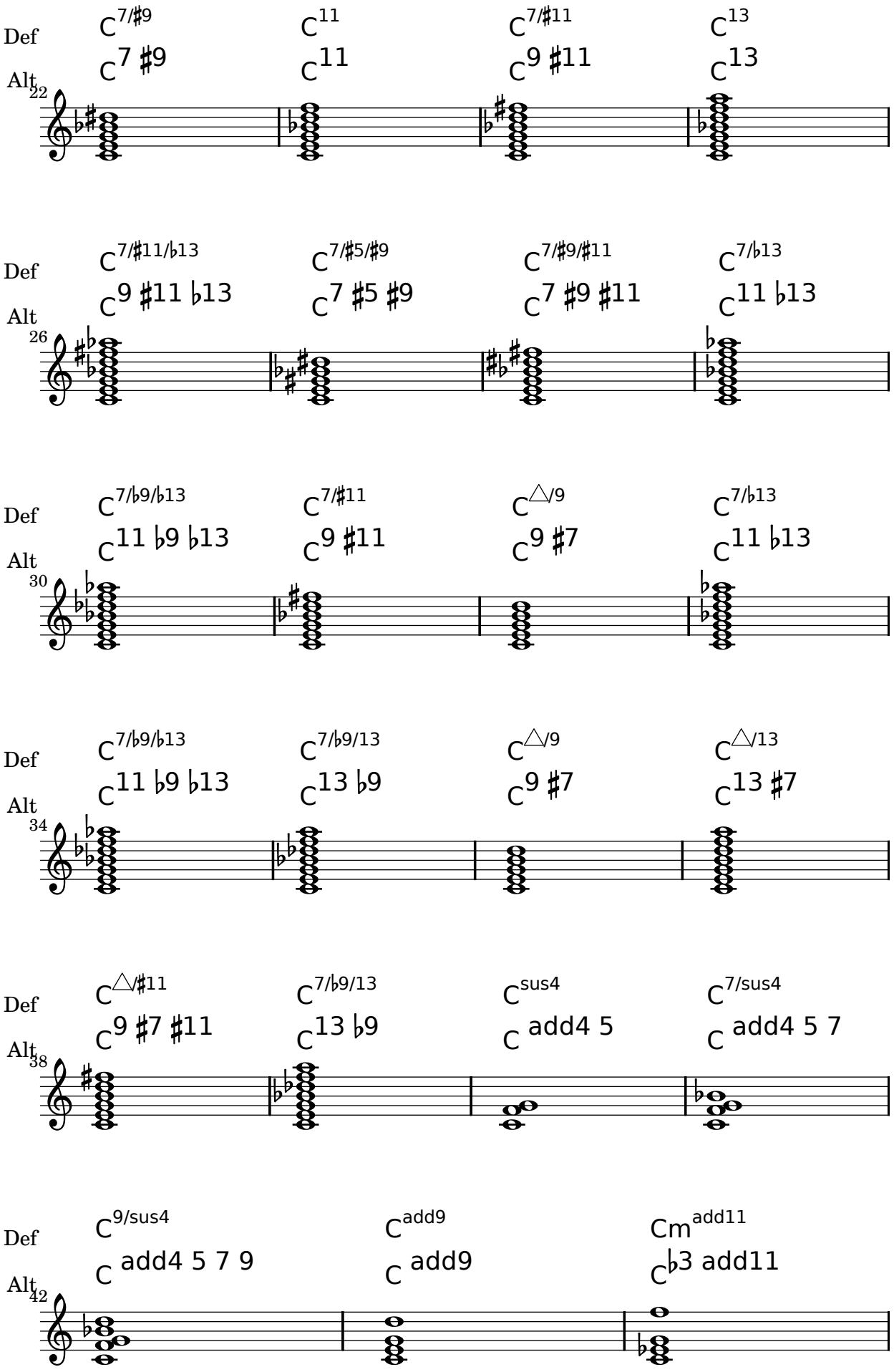
[1] In Scheme terminology, the pair is called cons, and its two elements are called car and cdr respectively.

Appendix C Notation manual tables

C.1 Chord name chart

The following charts shows two standard systems for printing chord names, along with the pitches they represent.




Def
Alt
22
C7/♯9
C7 ♯9
C11
C11
C7/♯11
C9 ♯11
C13
C13
26
C7/♯11/♭13
C9 ♯11 ♭13
C7/♯5/♯9
C7 ♯5 ♯9
C7/♯9/♯11
C7 ♯9 ♯11
C7/♭13
C11 ♭13
30
C7/♭9/♭13
C11 ♭9 ♭13
C7/♯11
C9 ♯11
C△/9
C9 ♯7
C7/♭13
C11 ♭13
34
C7/♭9/♭13
C11 ♭9 ♭13
C7/♭9/13
C13 ♭9
C△/9
C9 ♯7
C△/13
C13 ♯7
38
C△/♯11
C9 ♯7 ♯11
C7/♭9/13
C13 ♭9
Csus4
C add4 5
C7/sus4
C add4 5 7
42
C9/sus4
C add4 5 7 9
Cadd9
C add9
Cmadd11
C♭3 add11

C.2 MIDI instruments

The following is a list of names that can be used for the `midiInstrument` property.

```
acoustic grand
bright acoustic
electric grand
honky-tonk
electric piano 1
electric piano 2
harpsichord
clav
celesta
glockenspiel
music box
vibraphone
marimba
xylophone
tubular bells
dulcimer
drawbar organ
percussive organ
rock organ
church organ
reed organ
accordion
harmonica
concertina
acoustic guitar (nylon)
acoustic guitar (steel)
electric guitar (jazz)
electric guitar (clean)
electric guitar (muted)
overdriven guitar
distorted guitar
guitar harmonics
acoustic bass
electric bass (finger)
electric bass (pick)
fretless bass
slap bass 1
slap bass 2
synth bass 1
synth bass 2
violin
viola
cello
contrabass
tremolo strings
pizzicato strings
orchestral strings
timpani
string ensemble 1
string ensemble 2
synthstrings 1
synthstrings 2
choir aahs
voice oohs
synth voice
orchestra hit
trumpet
trombone
tuba
muted trumpet
french horn
brass section
synthbrass 1
synthbrass 2
soprano sax
alto sax
tenor sax
baritone sax
oboe
english horn
bassoon
clarinet
piccolo
flute
recorder
pan flute
blown bottle
shakuhachi
whistle
ocarina
lead 1 (square)
lead 2 (sawtooth)
lead 3 (calliope)
lead 4 (chiff)
lead 5 (charang)
lead 6 (voice)
lead 7 (fifths)
lead 8 (bass+lead)
pad 1 (new age)
pad 2 (warm)
pad 3 (polysynth)
pad 4 (choir)
pad 5 (bowed)
pad 6 (metallic)
pad 7 (halo)
pad 8 (sweep)
fx 1 (rain)
fx 2 (soundtrack)
fx 3 (crystal)
fx 4 (atmosphere)
fx 5 (brightness)
fx 6 (goblins)
fx 7 (echoes)
fx 8 (sci-fi)
sitar
banjo
shamisen
koto
kalimba
bagpipe
fiddle
shanai
tinkle bell
agogo
steel drums
woodblock
taiko drum
melodic tom
synth drum
reverse cymbal
guitar fret noise
breath noise
seashore
bird tweet
telephone ring
helicopter
applause
gunshot
```

C.3 List of colors

Normal colors

Usage syntax is detailed in Section 8.5.7 [Coloring objects], page 210.

```
black       white          red         green
blue        cyan           magenta     yellow
grey        darkred        darkgreen   darkblue
darkcyan    darkmagenta    darkyellow
```

X color names

X color names come several variants:

Any name that is spelled as a single word with capitalisation (e.g. "LightSlateBlue") can also be spelled as space separated words without capitalisation (e.g. "light slate blue").

The word "grey" can always be spelled "gray" (e.g. "DarkSlateGray").

Some names can take a numerical suffix (e.g. "LightSalmon4").

Color Names without a numerical suffix:

```
snow            GhostWhite      WhiteSmoke      gainsboro       FloralWhite
OldLace         linen           AntiqueWhite    PapayaWhip      BlanchedAlmond
bisque          PeachPuff       NavajoWhite     moccasin        cornsilk
ivory           LemonChiffon    seashell        honeydew        MintCream
azure           AliceBlue       lavender        LavenderBlush   MistyRose
white           black           DarkSlateGrey   DimGrey         SlateGrey
LightSlateGrey  grey            LightGrey       MidnightBlue    navy
NavyBlue        CornflowerBlue  DarkSlateBlue   SlateBlue       MediumSlateBlue
LightSlateBlue  MediumBlue      RoyalBlue       blue            DodgerBlue
DeepSkyBlue     SkyBlue         LightSkyBlue    SteelBlue       LightSteelBlue
LightBlue       PowderBlue      PaleTurquoise   DarkTurquoise   MediumTurquoise
turquoise       cyan            LightCyan       CadetBlue       MediumAquamarine
aquamarine      DarkGreen       DarkOliveGreen  DarkSeaGreen    SeaGreen
MediumSeaGreen  LightSeaGreen   PaleGreen       SpringGreen     LawnGreen
green           chartreuse      MediumSpringGreen       GreenYellow     LimeGreen
YellowGreen     ForestGreen     OliveDrab       DarkKhaki       khaki
PaleGoldenrod   LightGoldenrodYellow    LightYellow     yellow  gold
LightGoldenrod  goldenrod       DarkGoldenrod   RosyBrown       IndianRed
SaddleBrown     sienna          peru            burlywood       beige
wheat           SandyBrown      tan             chocolate       firebrick
brown           DarkSalmon      salmon          LightSalmon     orange
DarkOrange      coral           LightCoral      tomato          OrangeRed
red             HotPink         DeepPink        pink            LightPink
PaleVioletRed   maroon          MediumVioletRed VioletRed       magenta
violet          plum            orchid          MediumOrchid    DarkOrchid
DarkViolet      BlueViolet      purple          MediumPurple    thistle
DarkGrey        DarkBlue        DarkCyan        DarkMagenta     DarkRed
LightGreen
```

Color names with a numerical suffix

In the following names the suffix N can be a number in the range 1-4:

```
snowN           seashellN       AntiqueWhiteN   bisqueN         PeachPuffN
NavajoWhiteN    LemonChiffonN   cornsilkN       ivoryN          honeydewN
LavenderBlushN  MistyRoseN      azureN          SlateBlueN      RoyalBlueN
blueN           DodgerBlueN     SteelBlueN      DeepSkyBlueN    SkyBlueN
LightSkyBlueN   LightSteelBlueN LightBlueN      LightCyanN      PaleTurquoiseN
CadetBlueN      turquoiseN      cyanN           aquamarineN     DarkSeaGreenN
SeaGreenN       PaleGreenN      SpringGreenN    greenN          chartreuseN
```

```
OliveDrabN      DarkOliveGreenN khakiN          LightGoldenrodN LightYellowN
yellowN         goldN           goldenrodN      DarkGoldenrodN  RosyBrownN
IndianRedN      siennaN         burlywoodN      wheatN          tanN
chocolateN      firebrickN      brownN          salmonN         LightSalmonN
orangeN         DarkOrangeN     coralN          tomatoN         OrangeRedN
redN            DeepPinkN       HotPinkN        pinkN           LightPinkN
PaleVioletRedN  maroonN         VioletRedN      magentaN        orchidN
plumN           MediumOrchidN   DarkOrchidN     purpleN         MediumPurpleN
thistleN
```

Grey Scale

A grey scale can be obtained using:

```
greyN
```

Where N is in the range 0-100.

C.4 The Feta font

The following symbols are available in the Emmentaler font and may be accessed directly using text markup such as `g^\markup { \musicglyph #"scripts.segno" }`, see Section 8.1.4 [Text markup], page 170.

.notdef	space		
plus	+	comma	,
hyphen	-	period	.
zero	0	one	1
two	2	three	3
four	4	five	5
six	6	seven	7
eight	8	nine	9
f	*f*	m	*m*
p	*p*	r	*r*
s	*s*	z	*z*
rests.0	𝄻	rests.1	𝄻
rests.0o	𝄼	rests.1o	𝄼
rests.M3	𝄺	rests.M2	𝄺
rests.M1	𝄩	rests.2	𝄽
rests.2classical	𝄽	rests.3	𝄾
rests.4	𝄾	rests.5	𝄿
rests.6	𝅀	rests.7	𝅁
accidentals.2	♯	accidentals.1	♯
accidentals.3	𝄪	accidentals.0	♮
accidentals.M2	♭	accidentals.M1	♭
accidentals.M4	𝄫	accidentals.M3	𝄫
accidentals.4	𝄪	accidentals.rightparen	)
accidentals.leftparen	(	arrowheads.open.01	𝅘
arrowheads.open.0M1	𝅘	arrowheads.open.11	𝅘
arrowheads.open.1M1	𝅘	arrowheads.close.01	𝅘
arrowheads.close.0M1	𝅘	arrowheads.close.11	𝅘
arrowheads.close.1M1	𝅘	dots.dot	.
noteheads.uM2	𝅜	noteheads.dM2	𝅜
noteheads.sM1	𝅜	noteheads.s0	𝅝
noteheads.s1	𝅗𝅥	noteheads.s2	𝅗𝅥
noteheads.s0diamond	𝅝	noteheads.s1diamond	𝅗𝅥
noteheads.s2diamond	𝅗𝅥	noteheads.s0triangle	𝅝
noteheads.d1triangle	𝅗𝅥	noteheads.u1triangle	𝅗𝅥
noteheads.u2triangle	𝅗𝅥	noteheads.d2triangle	𝅗𝅥
noteheads.s0slash	𝅝	noteheads.s1slash	𝅗𝅥
noteheads.s2slash	𝅗𝅥	noteheads.s0cross	𝅝
noteheads.s1cross	𝅗𝅥	noteheads.s2cross	𝅗𝅥
noteheads.s2xcircle	𝅗𝅥	noteheads.s0do	𝅝
noteheads.d1do	𝅗𝅥	noteheads.u1do	𝅗𝅥
noteheads.d2do	𝅗𝅥	noteheads.u2do	𝅗𝅥
noteheads.s0re	𝅝	noteheads.u1re	𝅝
noteheads.d1re	𝅝	noteheads.u2re	𝅗𝅥
noteheads.d2re	𝅗𝅥	noteheads.s0mi	𝅝
noteheads.s1mi	𝅗𝅥	noteheads.s2mi	𝅗𝅥
noteheads.u0fa	𝅝	noteheads.d0fa	𝅝
noteheads.u1fa	𝅗𝅥	noteheads.d1fa	𝅗𝅥
noteheads.u2fa	𝅗𝅥	noteheads.d2fa	𝅗𝅥

Appendix D Templates

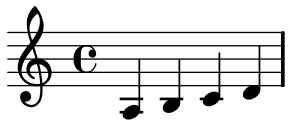
This section of the manual contains templates with the LilyPond score already set up for you. Just add notes, run LilyPond, and enjoy beautiful printed scores!

D.1 Single staff

D.1.1 Notes only

The first example gives you a staff with notes, suitable for a solo instrument or a melodic fragment. Cut and paste this into a file, add notes, and you're finished!

```
\version "2.10.10"
melody = \relative c' {
   \clef treble
   \key c \major
   \time 4/4

   a4 b c d
}

\score {
   \new Staff \melody
   \layout { }
   \midi {}
}
```

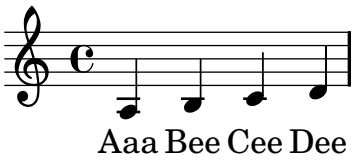


D.1.2 Notes and lyrics

The next example demonstrates a simple melody with lyrics. Cut and paste, add notes, then words for the lyrics. This example turns off automatic beaming, which is common for vocal parts. If you want to use automatic beaming, you'll have to change or comment out the relevant line.

```
\version "2.10.10"
melody = \relative c' {
   \clef treble
   \key c \major
   \time 4/4

   a4 b c d
}

text = \lyricmode {
   Aaa Bee Cee Dee
}

\score{
   <<
      \new Voice = "one" {
```

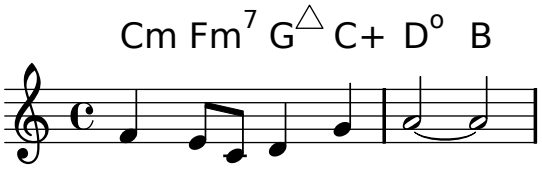
```
            \autoBeamOff
            \melody
         }
         \new Lyrics \lyricsto "one" \text
      >>
      \layout { }
      \midi { }
   }
```



D.1.3 Notes and chords

Want to prepare a lead sheet with a melody and chords? Look no further!

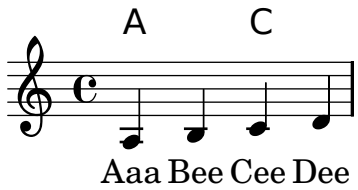
```
\version "2.10.10"
melody = \relative c' {
   \clef treble
   \key c \major
   \time 4/4

   f4 e8[ c] d4 g |
   a2 ~ a2 |
}

harmonies = \chordmode {
   c4:m f:min7 g:maj c:aug d2:dim b:sus
}

\score {
   <<
      \new ChordNames {
         \set chordChanges = ##t
         \harmonies
      }
   \new Staff \melody
   >>

   \layout{ }
   \midi { }
}
```



D.1.4 Notes, lyrics, and chords.

This template allows you to prepare a song with melody, words, and chords.

```
\version "2.10.10"
melody = \relative c' {
   \clef treble
   \key c \major
   \time 4/4

   a b c d
}

text = \lyricmode {
   Aaa Bee Cee Dee
}

harmonies = \chordmode {
   a2 c2
}

\score {
   <<
      \new ChordNames {
         \set chordChanges = ##t
         \harmonies
      }
   \new Voice = "one" {
      \autoBeamOff
      \melody
   }
   \new Lyrics \lyricsto "one" \text
   >>
   \layout { }
   \midi { }
}
```



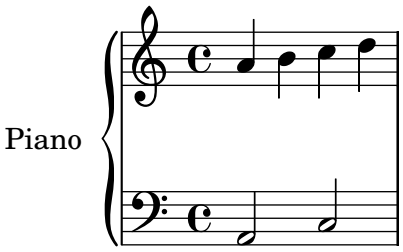
D.2 Piano templates

D.2.1 Solo piano

Here is a simple piano staff.

```
\version "2.10.10"
upper = \relative c'' {
   \clef treble
   \key c \major
```

```
   \time 4/4

   a b c d
}

lower = \relative c {
   \clef bass
   \key c \major
   \time 4/4

   a2 c
}

\score {
   \new PianoStaff <<
      \set PianoStaff.instrumentName = "Piano  "
      \new Staff = "upper" \upper
      \new Staff = "lower" \lower
   >>
   \layout { }
   \midi { }
}
```

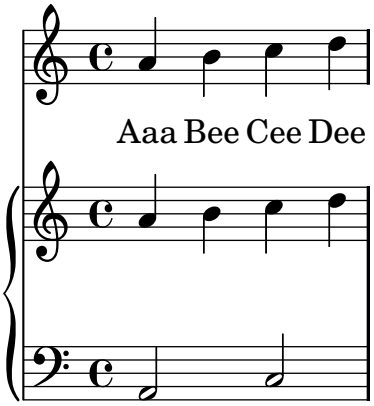


D.2.2 Piano and melody with lyrics

Here is a typical song format: one staff with the melody and lyrics, with piano accompaniment underneath.

```
\version "2.10.10"
melody = \relative c'' {
   \clef treble
   \key c \major
   \time 4/4

   a b c d
}

text = \lyricmode {
   Aaa Bee Cee Dee
}

upper = \relative c'' {
   \clef treble
   \key c \major
```

```
    \time 4/4

    a b c d
}

lower = \relative c {
    \clef bass
    \key c \major
    \time 4/4

    a2 c
}

\score {
    <<
        \new Voice = "mel" {
            \autoBeamOff
            \melody
        }
        \new Lyrics \lyricsto mel \text

        \new PianoStaff <<
            \new Staff = "upper" \upper
            \new Staff = "lower" \lower
        >>
    >>
    \layout {
        \context { \RemoveEmptyStaffContext }
    }
    \midi { }
}
```

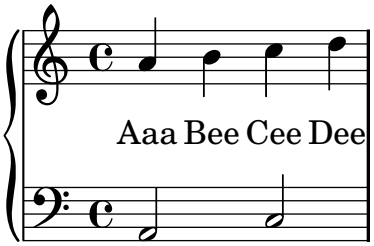


D.2.3 Piano centered lyrics

Instead of having a full staff for the melody and lyrics, you can place the lyrics between the piano staff (and omit the separate melody staff).

```
\version "2.10.10"
upper = \relative c'' {
    \clef treble
```

```
    \key c \major
    \time 4/4

    a b c d
}

lower = \relative c {
    \clef bass
    \key c \major
    \time 4/4

    a2 c
}

text = \lyricmode {
    Aaa Bee Cee Dee
}

\score {
  \new GrandStaff <<
    \new Staff = upper { \new Voice = "singer" \upper }
    \new Lyrics \lyricsto "singer" \text
    \new Staff = lower {
      \clef bass
      \lower
    }
  >>
  \layout {
    \context { \GrandStaff \accepts "Lyrics" }
    \context { \Lyrics \consists "Bar_engraver" }
  }
  \midi { }
}
```



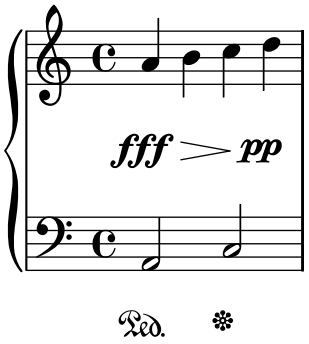
D.2.4 Piano centered dynamics

Many piano scores have the dynamics centered between the two staffs. This requires a bit of tweaking to implement, but since the template is right here, you don't have to do the tweaking yourself.

```
\version "2.10.10"
upper = \relative c'' {
  \clef treble
  \key c \major
```

```
  \time 4/4

  a b c d
}

lower = \relative c {
  \clef bass
  \key c \major
  \time 4/4

  a2 c
}

dynamics = {
  s2\fff\> s4
  s\!\pp
}

pedal = {
  s2\sustainDown s2\sustainUp
}

\score {
  \new PianoStaff <<
    \new Staff = "upper" \upper
    \new Dynamics = "dynamics" \dynamics
    \new Staff = "lower" <<
      \clef bass
      \lower
    >>
    \new Dynamics = "pedal" \pedal
  >>
  \layout {
    \context {
      \type "Engraver_group"
      \name Dynamics
      \alias Voice % So that \cresc works, for example.
      \consists "Output_property_engraver"

      \override VerticalAxisGroup #'minimum-Y-extent = #'(-1 . 1)
      pedalSustainStrings = #'("Ped." "*Ped." "*")
      pedalUnaCordaStrings = #'("una corda" "" "tre corde")

      \consists "Piano_pedal_engraver"
      \consists "Script_engraver"
      \consists "Dynamic_engraver"
      \consists "Text_engraver"

      \override TextScript #'font-size = #2
      \override TextScript #'font-shape = #'italic
      \override DynamicText #'extra-offset = #'(0 . 2.5)
      \override Hairpin #'extra-offset = #'(0 . 2.5)
```

```
        \consists "Skip_event_swallow_translator"

        \consists "Axis_group_engraver"
      }
      \context {
        \PianoStaff
        \accepts Dynamics
        \override VerticalAlignment #'forced-distance = #7
      }
    }
  }
  \score {
    \new PianoStaff <<
      \new Staff = "upper" << \upper \dynamics >>
      \new Staff = "lower" << \lower \dynamics >>
      \new Dynamics = "pedal" \pedal
    >>
    \midi {
      \context {
        \type "Performer_group"
        \name Dynamics
        \consists "Piano_pedal_performer"
      }
      \context {
        \PianoStaff
        \accepts Dynamics
      }
    }
  }
```



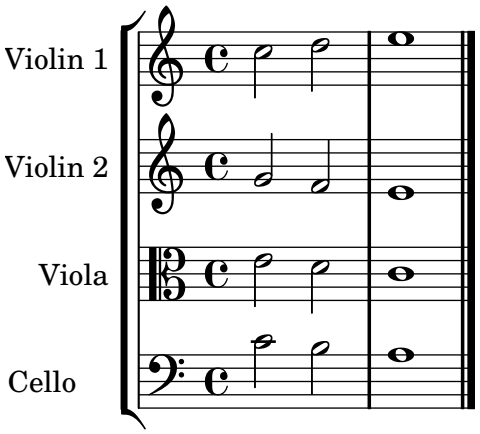
D.3 String quartet

D.3.1 String quartet

This template demonstrates a string quartet. It also uses a `\global` section for time and key signatures.

```
  \version "2.10.10"

  global= {
    \time 4/4
```

```
  \key c \major
}

violinOne = \new Voice { \relative c''{
  \set Staff.instrumentName = "Violin 1 "

  c2 d e1

\bar "|." }}
violinTwo = \new Voice { \relative c''{
  \set Staff.instrumentName = "Violin 2 "

  g2 f e1

\bar "|." }}
viola = \new Voice { \relative c' {
  \set Staff.instrumentName = "Viola "
  \clef alto

  e2 d c1

\bar "|." }}
cello = \new Voice { \relative c' {
  \set Staff.instrumentName = "Cello     "
  \clef bass

  c2 b a1

\bar "|."}}

\score {
   \new StaffGroup <<
      \new Staff << \global \violinOne >>
      \new Staff << \global \violinTwo >>
      \new Staff << \global \viola >>
      \new Staff << \global \cello >>
   >>
   \layout { }
   \midi { }
}
```



D.3.2 String quartet parts

The previous example produces a nice string quartet, but what if you needed to print parts? This template demonstrates how to use the `\tag` feature to easily split a piece into individual parts.

You need to split this template into separate files; the filenames are contained in comments at the beginning of each file. `piece.ly` contains all the music definitions. The other files – `score.ly`, `vn1.ly`, `vn2.ly`, `vla.ly`, and `vlc.ly` – produce the appropiate part.

```
%%%%% piece.ly
\version "2.10.10"

global= {
  \time 4/4
  \key c \major
}

Violinone = \new Voice { \relative c''{
  \set Staff.instrumentName = "Violin 1 "

  c2 d e1

\bar "|." }}   %*********************************
Violintwo = \new Voice { \relative c''{
  \set Staff.instrumentName = "Violin 2 "

  g2 f e1

\bar "|." }}   %*********************************
Viola = \new Voice { \relative c' {
  \set Staff.instrumentName = "Viola "
  \clef alto

  e2 d c1

\bar "|." }}   %*********************************
Cello = \new Voice { \relative c' {
  \set Staff.instrumentName = "Cello     "
  \clef bass

  c2 b a1
```

```
\bar "|."}}   %*********************************

music = {
  <<
    \tag #'score \tag #'vn1 \new Staff { << \global \Violinone >> }
    \tag #'score \tag #'vn2 \new Staff { << \global \Violintwo>> }
    \tag #'score \tag #'vla \new Staff { << \global \Viola>> }
    \tag #'score \tag #'vlc \new Staff { << \global \Cello>> }
  >>
}



%%%%% score.ly
\version "2.10.10"
\include "piece.ly"
#(set-global-staff-size 14)
\score {
  \new StaffGroup \keepWithTag #'score \music
  \layout { }
  \midi { }
}


%%%%% vn1.ly
\version "2.10.10"
\include "piece.ly"
\score {
  \keepWithTag #'vn1 \music
  \layout { }
}


%%%%% vn2.ly
\version "2.10.10"
\include "piece.ly"
\score {
  \keepWithTag #'vn2 \music
  \layout { }
}


%%%%% vla.ly
\version "2.10.10"
\include "piece.ly"
\score {
  \keepWithTag #'vla \music
  \layout { }
}


%%%%% vlc.ly
```

```
\version "2.10.10"
\include "piece.ly"
\score {
  \keepWithTag #'vlc \music
  \layout { }
}
```

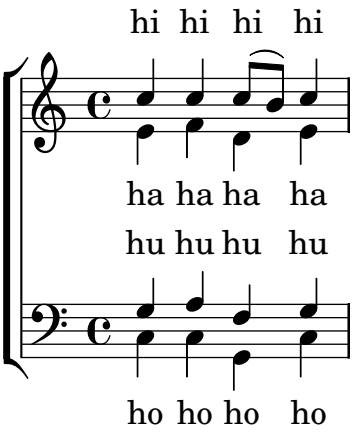
D.4 Vocal ensembles

D.4.1 SATB vocal score

Here is a standard four-part SATB vocal score. With larger ensembles, it's often useful to include a section which is included in all parts. For example, the time signature and key signatures are almost always the same for all parts.

```
\version "2.10.10"
global = {
   \key c \major
   \time 4/4
}

sopMusic = \relative c'' {
   c4 c c8[( b)] c4
}
sopWords = \lyricmode {
   hi hi hi hi
}

altoMusic = \relative c' {
   e4 f d e
}
altoWords =\lyricmode {
   ha ha ha ha
}

tenorMusic = \relative c' {
   g4 a f g
}
tenorWords = \lyricmode {
   hu hu hu hu
}

bassMusic = \relative c {
   c4 c g c
}
bassWords = \lyricmode {
   ho ho ho ho
}

\score {
   \new ChoirStaff <<
      \new Lyrics = sopranos { s1 }
      \new Staff = women <<
```

```
            \new Voice =
              "sopranos" { \voiceOne << \global \sopMusic >> }
            \new Voice =
              "altos" { \voiceTwo << \global \altoMusic >> }
         >>
         \new Lyrics = "altos" { s1 }
         \new Lyrics = "tenors" { s1 }
         \new Staff = men <<
            \clef bass
            \new Voice =
              "tenors" { \voiceOne <<\global \tenorMusic >> }
            \new Voice =
              "basses" { \voiceTwo <<\global \bassMusic >> }
         >>
         \new Lyrics = basses { s1 }

         \context Lyrics = sopranos \lyricsto sopranos \sopWords
         \context Lyrics = altos \lyricsto altos \altoWords
         \context Lyrics = tenors \lyricsto tenors \tenorWords
         \context Lyrics = basses \lyricsto basses \bassWords
      >>

      \layout {
         \context {
            % a little smaller so lyrics
            % can be closer to the staff
            \Staff
            \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
         }
      }
   }
```

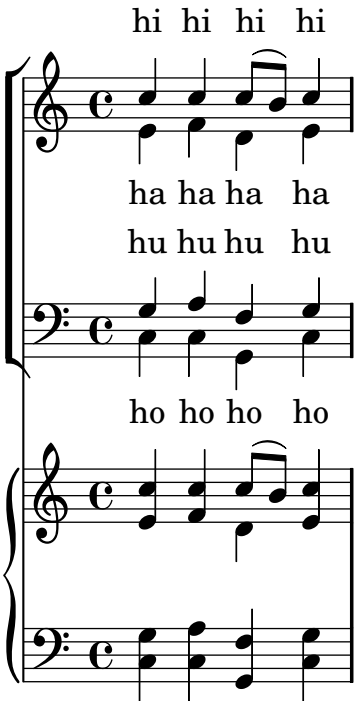


D.4.2 SATB vocal score and automatic piano reduction

This template adds an automatic piano reduction to the SATB vocal score. This demonstrates one of the strengths of LilyPond – you can use a music definition more than once. If you make any changes to the vocal notes (say, tenorMusic), then the changes will also apply to the piano reduction.

```
\version "2.10.10"
global = {
   \key c \major
   \time 4/4
}

sopMusic = \relative c'' {
   c4 c c8[( b)] c4
}
sopWords = \lyricmode {
   hi hi hi hi
}

altoMusic = \relative c' {
   e4 f d e
}
altoWords =\lyricmode {
   ha ha ha ha
}

tenorMusic = \relative c' {
   g4 a f g
}
tenorWords = \lyricmode {
   hu hu hu hu
}

bassMusic = \relative c {
   c4 c g c
}
bassWords = \lyricmode {
   ho ho ho ho
}

\score {
  <<
    \new ChoirStaff <<
      \new Lyrics = sopranos { s1 }
      \new Staff = women <<
        \new Voice =
          "sopranos" { \voiceOne << \global \sopMusic >> }
        \new Voice =
          "altos" { \voiceTwo << \global \altoMusic >> }
      >>
      \new Lyrics = "altos" { s1 }
      \new Lyrics = "tenors" { s1 }
      \new Staff = men <<
        \clef bass
        \new Voice =
          "tenors" { \voiceOne <<\global \tenorMusic >> }
        \new Voice =
          "basses" { \voiceTwo <<\global \bassMusic >> }
```

```
        >>
        \new Lyrics = basses { s1 }

        \context Lyrics = sopranos \lyricsto sopranos \sopWords
        \context Lyrics = altos \lyricsto altos \altoWords
        \context Lyrics = tenors \lyricsto tenors \tenorWords
        \context Lyrics = basses \lyricsto basses \bassWords
      >>
    \new PianoStaff <<
      \new Staff <<
        \set Staff.printPartCombineTexts = ##f
        \partcombine
        << \global \sopMusic >>
        << \global \altoMusic >>
      >>
      \new Staff <<
        \clef bass
        \set Staff.printPartCombineTexts = ##f
        \partcombine
        << \global \tenorMusic >>
        << \global \bassMusic >>
      >>
     >>
    >>
    \layout {
      \context {
        % a little smaller so lyrics
        % can be closer to the staff
        \Staff
          \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
      }
    }
  }
```



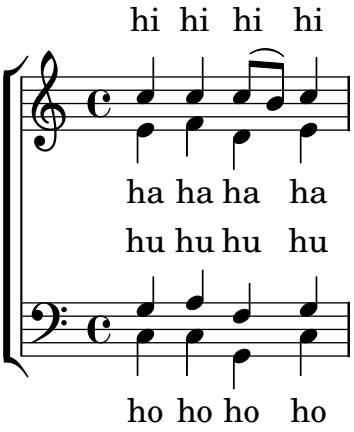
D.4.3 SATB with aligned contexts

Here all the lyrics lines are placed using `alignAboveContext` and `alignBelowContext`.

```
\version "2.10.10"
global = {
  \key c \major
  \time 4/4
}

sopMusic = \relative c'' {
  c4 c c8[( b)] c4
}
sopWords = \lyricmode {
  hi hi hi hi
}

altoMusic = \relative c' {
  e4 f d e
}
altoWords =\lyricmode {
  ha ha ha ha
}

tenorMusic = \relative c' {
  g4 a f g
}
tenorWords = \lyricmode {
  hu hu hu hu
}

bassMusic = \relative c {
  c4 c g c
}
bassWords = \lyricmode {
```

```
  ho ho ho ho
}

\score {
  \new ChoirStaff <<
     \new Staff = women <<
        \new Voice =
          "sopranos" { \voiceOne << \global \sopMusic >> }
        \new Voice =
          "altos" { \voiceTwo << \global \altoMusic >> }
     >>
     \new Lyrics \with {alignAboveContext=women} \lyricsto sopranos \sopWords
     \new Lyrics \with {alignBelowContext=women} \lyricsto altos \altoWords
% we could remove the line about this with the line below, since we want
% the alto lyrics to be below the alto Voice anyway.
%    \new Lyrics \lyricsto altos \altoWords

     \new Staff = men <<
        \clef bass
        \new Voice =
          "tenors" { \voiceOne <<\global \tenorMusic >> }
        \new Voice =
          "basses" { \voiceTwo <<\global \bassMusic >> }
     >>

     \new Lyrics \with {alignAboveContext=men} \lyricsto tenors \tenorWords
     \new Lyrics \with {alignBelowContext=men} \lyricsto basses \bassWords
% again, we could replace the line above this with the line below.
%    \new Lyrics \lyricsto basses \bassWords
  >>

  \layout {
     \context {
        % a little smaller so lyrics
        % can be closer to the staff
        \Staff
        \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
     }
  }
}


\score {
  \new ChoirStaff <<
     \new Staff = women <<
        \new Voice =
          "sopranos" { \voiceOne << \global \sopMusic >> }
        \new Voice =
          "altos" { \voiceTwo << \global \altoMusic >> }
     >>

     \new Lyrics \with {alignAboveContext=women} \lyricsto sopranos \sopWords
```

```
      \new Lyrics \lyricsto altos \altoWords

      \new Staff = men <<
         \clef bass
         \new Voice =
           "tenors" { \voiceOne <<\global \tenorMusic >> }
         \new Voice =
           "basses" { \voiceTwo <<\global \bassMusic >> }
      >>

      \new Lyrics \with {alignAboveContext=men} \lyricsto tenors \tenorWords
      \new Lyrics \lyricsto basses \bassWords
   >>

   \layout {
      \context {
         % a little smaller so lyrics
         % can be closer to the staff
         \Staff
         \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
      }
   }
}
```



D.5 Ancient notation templates

D.5.1 Transcription of mensural music

When transcribing mensural music, an incipit at the beginning of the piece is useful to indicate the original key and tempo. While today musicians are used to bar lines in order to faster recognize rhythmic patterns, bar lines were not yet invented during the period of mensural music; in fact, the meter often changed after every few notes. As a compromise, bar lines are often printed between the staves rather than on the staves.

```
\version "2.10.10"

global = {
  \set Score.skipBars = ##t
```

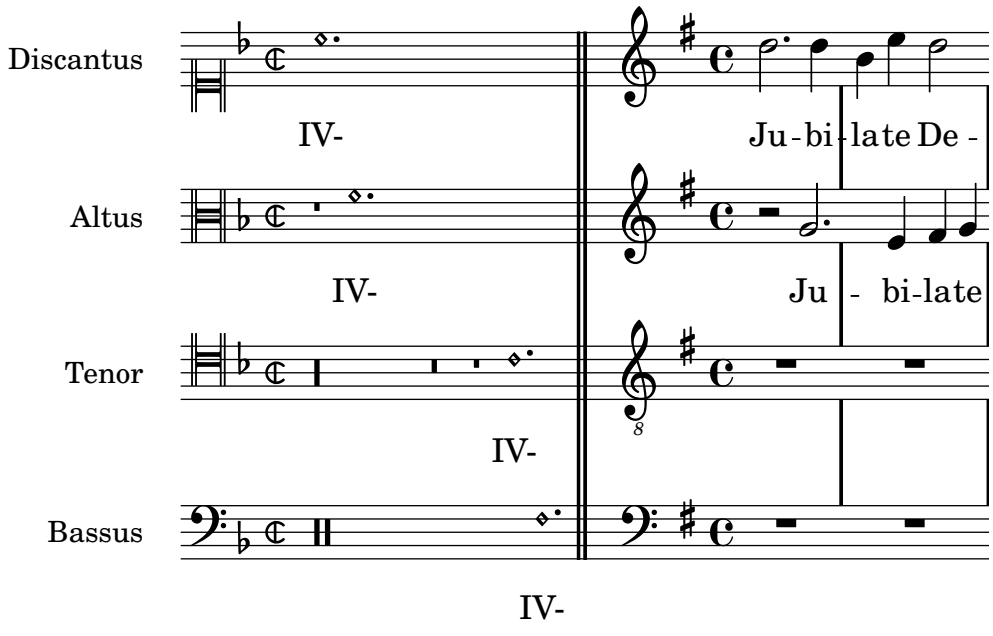
```
  % incipit
  \once \override Score.SystemStartBracket #'transparent = ##t
  \override Score.SpacingSpanner #'spacing-increment = #1.0 % tight spacing
  \key f \major
  \time 2/2
  \once \override Staff.TimeSignature #'style = #'neomensural
  \override Voice.NoteHead #'style = #'neomensural
  \override Voice.Rest #'style = #'neomensural
  \set Staff.printKeyCancellation = ##f
  \cadenzaOn % turn off bar lines
  \skip 1*10
  \once \override Staff.BarLine #'transparent = ##f
  \bar "||"
  \skip 1*1 % need this extra \skip such that clef change comes
            % after bar line
  \bar ""

  % main
  \revert Score.SpacingSpanner #'spacing-increment % CHECK: no effect?
  \cadenzaOff % turn bar lines on again
  \once \override Staff.Clef #'full-size-change = ##t
  \set Staff.forceClef = ##t
  \key g \major
  \time 4/4
  \override Voice.NoteHead #'style = #'default
  \override Voice.Rest #'style = #'default

  % FIXME: setting printKeyCancellation back to #t must not
  % occur in the first bar after the incipit.  Dto. for forceClef.
  % Therefore, we need an extra \skip.
  \skip 1*1
  \set Staff.printKeyCancellation = ##t
  \set Staff.forceClef = ##f

  \skip 1*7 % the actual music

  % let finis bar go through all staves
  \override Staff.BarLine #'transparent = ##f

  % finis bar
  \bar "|."
}

discantusNotes = {
  \transpose c' c'' {
    \set Staff.instrumentName = "Discantus  "

    % incipit
    \clef "neomensural-c1"
    c'1. s2   % two bars
    \skip 1*8 % eight bars
    \skip 1*1 % one bar
```

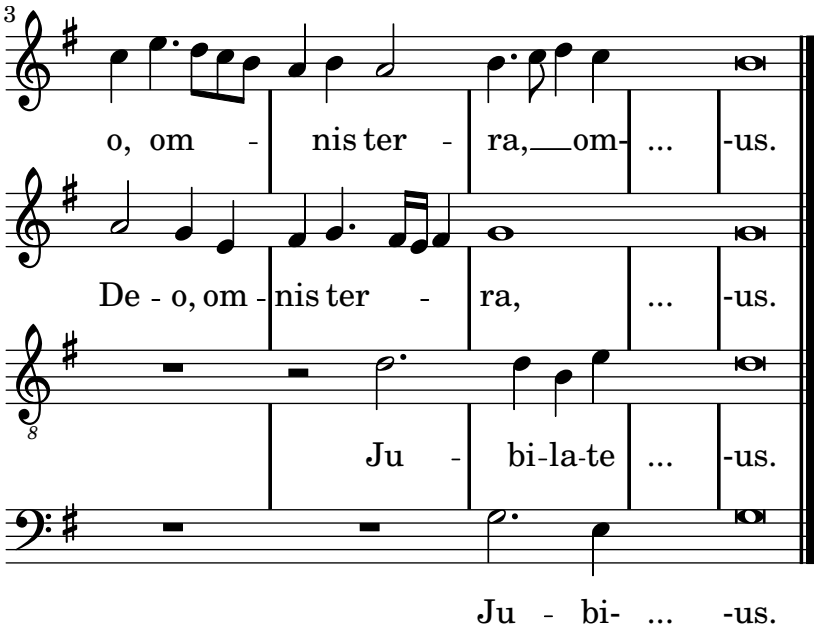
```
    % main
    \clef "treble"
    d'2. d'4 |
    b e' d'2 |
    c'4 e'4.( d'8 c' b |
    a4) b a2 |
    b4.( c'8 d'4) c'4 |
    \once \override NoteHead #'transparent = ##t c'1 |
    b\breve |
  }
}

discantusLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi -- |
  la -- te De -- |
  o, om --
  nis ter -- |
  ra, __ om- |
  "..." |
  -us. |
}

altusNotes = {
  \transpose c' c'' {
    \set Staff.instrumentName = "Altus  "

    % incipit
    \clef "neomensural-c3"
    r1        % one bar
    f1. s2    % two bars
    \skip 1*7 % seven bars
    \skip 1*1 % one bar

    % main
    \clef "treble"
    r2 g2. e4 fis g | % two bars
    a2 g4 e |
    fis g4.( fis16 e fis4) |
    g1 |
    \once \override NoteHead #'transparent = ##t g1 |
    g\breve |
  }
}

altusLyrics = \lyricmode {
  % incipit
  IV-
```

```
  % main
  Ju -- bi -- la -- te | % two bars
  De -- o, om -- |
  nis ter -- ra, |
  "..." |
  -us. |
}

tenorNotes = {
  \transpose c' c' {
    \set Staff.instrumentName = "Tenor  "

    % incipit
    \clef "neomensural-c4"
    r\longa   % four bars
    r\breve   % two bars
    r1        % one bar
    c'1. s2   % two bars
    \skip 1*1 % one bar
    \skip 1*1 % one bar

    % main
    \clef "treble_8"
    R1 |
    R1 |
    R1 |
    r2 d'2. d'4 b e' | % two bars
    \once \override NoteHead #'transparent = ##t e'1 |
    d'\breve |
  }
}

tenorLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi -- la -- te | % two bars
  "..." |
  -us. |
}

bassusNotes = {
  \transpose c' c' {
    \set Staff.instrumentName = "Bassus  "

    % incipit
    \clef "bass"
    r\maxima  % eight bars
    f1. s2    % two bars
    \skip 1*1 % one bar
```

```
    % main
    \clef "bass"
    R1 |
    R1 |
    R1 |
    R1 |
    g2. e4 |
    \once \override NoteHead #'transparent = ##t e1 |
    g\breve |
  }
}

bassusLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi- |
  "..." |
  -us. |
}

\score {
  \new StaffGroup = choirStaff <<
    \new Voice =
      "discantusNotes" << \global \discantusNotes >>
    \new Lyrics =
      "discantusLyrics" \lyricsto discantusNotes { \discantusLyrics }
    \new Voice =
      "altusNotes" << \global \altusNotes >>
    \new Lyrics =
      "altusLyrics" \lyricsto altusNotes { \altusLyrics }
    \new Voice =
      "tenorNotes" << \global \tenorNotes >>
    \new Lyrics =
      "tenorLyrics" \lyricsto tenorNotes { \tenorLyrics }
    \new Voice =
      "bassusNotes" << \global \bassusNotes >>
    \new Lyrics =
      "bassusLyrics" \lyricsto bassusNotes { \bassusLyrics }
  >>
  \layout {
    \context {
      \Score

      % no bars in staves
      \override BarLine #'transparent = ##t

      % incipit should not start with a start delimiter
      \remove "System_start_delimiter_engraver"
    }
```

```
    \context {
      \Voice

      % no slurs
      \override Slur #'transparent = ##t

      % Comment in the below "\remove" command to allow line
      % breaking also at those barlines where a note overlaps
      % into the next bar.  The command is commented out in this
      % short example score, but especially for large scores, you
      % will typically yield better line breaking and thus improve
      % overall spacing if you comment in the following command.
      %\remove "Forbid_line_break_engraver"
    }
  }
}
```



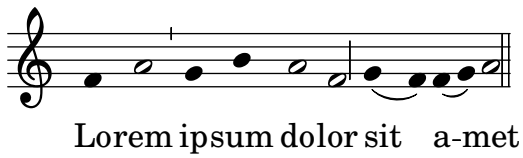


D.5.2 Gregorian transcription template

This example demonstrates how to do modern transcription of Gregorian music. Gregorian music has no measure, no stems; it uses only half and quarter noteheads, and special marks, indicating rests of different length.

```
\include "gregorian-init.ly"
\version "2.10.10"

chant = \relative c' {
  \set Score.timing = ##f
  f4 a2 \divisioMinima
  g4 b a2 f2 \divisioMaior
  g4( f) f( g) a2 \finalis
}

verba = \lyricmode {
  Lo -- rem ip -- sum do -- lor sit a -- met
}

\score {
  \new Staff <<
    \new Voice = "melody" {
      \chant
    }
    \new Lyrics = "one" \lyricsto melody \verba
  >>

  \layout {
    \context {
      \Staff
      \remove "Time_signature_engraver"
      \remove "Bar_engraver"
      \override Stem #'transparent = ##t
    }
    \context {
      \Voice
```

```
      \override Stem #'length = #0
    }
    \context {
      \Score
      barAlways = ##t
    }
  }
}
```



D.6 Jazz combo

This is a much more complicated template, for a jazz ensemble. Note that all instruments are notated in `\key c \major`. This refers to the key in concert pitch; LilyPond will automatically transpose the key if the music is within a `\transpose` section.

```
\version "2.10.10"
\header {
  title = "Song"
  subtitle = "(tune)"
  composer = "Me"
  meter = "moderato"
  piece = "Swing"
  tagline = \markup {
    \column {
      "LilyPond example file by Amelie Zapf,"
      "Berlin 07/07/2003"
    }
  }
  texidoc = "Jazz tune for combo
             (horns, guitar, piano, bass, drums)."
}

#(set-global-staff-size 16)
\include "english.ly"

%%%%%%%%%% Some macros %%%%%%%%%%%%%%%%%%

sl = {
  \override NoteHead #'style = #'slash
  \override Stem #'transparent = ##t
}
nsl = {
  \revert NoteHead #'style
  \revert Stem #'transparent
}
cr = \override NoteHead #'style = #'cross
ncr = \revert NoteHead #'style
```

```
%% insert chord name style stuff here.

jzchords = { }


%%%%%%%%%%% Keys'n'thangs %%%%%%%%%%%%%%%%

global = {
  \time 4/4
}

Key = { \key c \major }

% ############ Horns ############

% ------ Trumpet ------
trpt = \transpose c d \relative c'' {
  \Key
  c1 c c
}
trpharmony = \transpose c' d {
  \jzchords
}
trumpet = {
  \global
  \set Staff.instrumentName = #"Trumpet"
  \clef treble
  <<
    \trpt
  >>
}

% ------ Alto Saxophone ------
alto = \transpose c a \relative c' {
  \Key
  c1 c c
}
altoharmony = \transpose c' a {
  \jzchords
}
altosax = {
  \global
  \set Staff.instrumentName = #"Alto Sax"
  \clef treble
  <<
    \alto
  >>
}

% ------ Baritone Saxophone ------
bari = \transpose c a' \relative c {
```

```
  \Key
  c1 c \sl d4^"Solo" d d d \nsl
}
bariharmony = \transpose c' a \chordmode {
  \jzchords s1 s d2:maj e:m7
}
barisax = {
  \global
  \set Staff.instrumentName = #"Bari Sax"
  \clef treble
  <<
    \bari
  >>
}

% ------ Trombone ------
tbone = \relative c {
  \Key
  c1 c c
}
tboneharmony = \chordmode {
  \jzchords
}
trombone = {
  \global
  \set Staff.instrumentName = #"Trombone"
  \clef bass
  <<
    \tbone
  >>
}

% ############ Rhythm Section #############

% ------ Guitar ------
gtr = \relative c'' {
  \Key
  c1 \sl b4 b b b \nsl c1
}
gtrharmony = \chordmode {
  \jzchords
  s1 c2:min7+ d2:maj9
}
guitar = {
  \global
  \set Staff.instrumentName = #"Guitar"
  \clef treble
  <<
    \gtr
  >>
}
```

```
%% ------ Piano ------
rhUpper = \relative c'' {
  \voiceOne
  \Key
  c1 c c
}
rhLower = \relative c' {
  \voiceTwo
  \Key
  e1 e e
}

lhUpper = \relative c' {
  \voiceOne
  \Key
  g1 g g
}
lhLower = \relative c {
  \voiceTwo
  \Key
  c1 c c
}

PianoRH = {
  \clef treble
  \global
  \set Staff.midiInstrument = "acoustic grand"
  <<
    \new Voice = "one" \rhUpper
    \new Voice = "two" \rhLower
  >>
}
PianoLH = {
  \clef bass
  \global
  \set Staff.midiInstrument = "acoustic grand"
  <<
    \new Voice = "one" \lhUpper
    \new Voice = "two" \lhLower
  >>
}

piano = {
  <<
    \set PianoStaff.instrumentName = #"Piano"
    \new Staff = "upper" \PianoRH
    \new Staff = "lower" \PianoLH
  >>
}

% ------ Bass Guitar ------
Bass = \relative c {
```

```
  \Key
  c1 c c
}
bass = {
  \global
  \set Staff.instrumentName = #"Bass"
  \clef bass
  <<
    \Bass
  >>
}

% ------ Drums ------
up = \drummode {
  hh4 <hh sn>4 hh <hh sn> hh <hh sn>4
  hh4 <hh sn>4
  hh4 <hh sn>4
  hh4 <hh sn>4
}

down = \drummode {
  bd4 s bd s bd s bd s bd s bd s
}

drumContents = {
  \global
  <<
    \set DrumStaff.instrumentName = #"Drums"
    \new DrumVoice { \voiceOne \up }
    \new DrumVoice { \voiceTwo \down }
  >>
}

%%%%%%%% It All Goes Together Here %%%%%%%%%%%%%%%%%%%%

\score {
  <<
    \new StaffGroup = "horns" <<
      \new Staff = "trumpet" \trumpet
      \new Staff = "altosax" \altosax
      \new ChordNames = "barichords" \bariharmony
      \new Staff = "barisax" \barisax
      \new Staff = "trombone" \trombone
    >>

    \new StaffGroup = "rhythm" <<
      \new ChordNames = "chords" \gtrharmony
      \new Staff = "guitar" \guitar
      \new PianoStaff = "piano" \piano
      \new Staff = "bass" \bass
      \new DrumStaff { \drumContents }
    >>
```

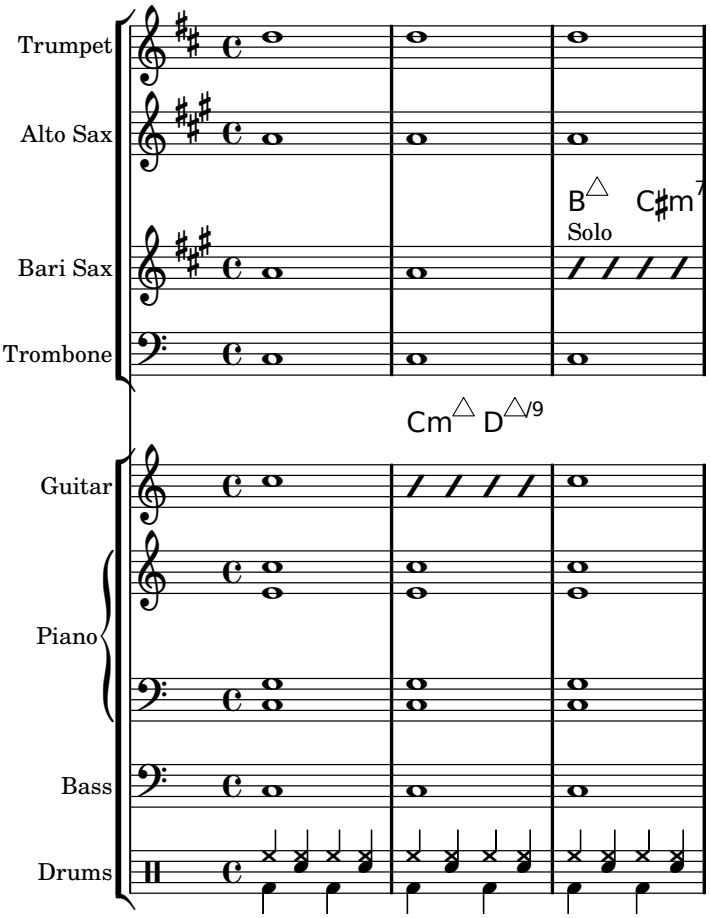
```
    >>

    \layout {
      \context { \RemoveEmptyStaffContext }
      \context {
        \Score
        \override BarNumber #'padding = #3
        \override RehearsalMark #'padding = #2
        skipBars = ##t
      }
    }

    \midi { }
  }
```

Song
(tune)

Me

moderato

Swing



D.7 Lilypond-book templates

These templates are for use with `lilypond-book`. If you're not familiar with this program, please refer to Chapter 14 [LilyPond-book], page 295.

D.7.1 LaTeX

You can include LilyPond fragments in a LaTeX document.

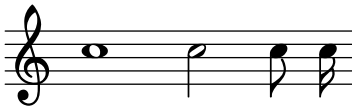
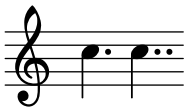
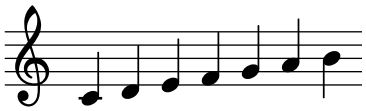
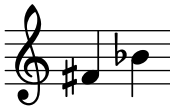
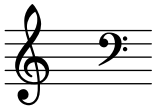
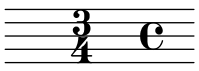
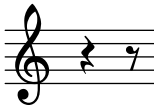
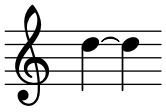
```
\documentclass[]{article}

\begin{document}

Normal LaTeX text.

\begin{lilypond}
\relative c'' {
a4 b c d
}
\end{lilypond}

More LaTeX text.

\begin{lilypond}
\relative c'' {
d4 c b a
}
\end{lilypond}
\end{document}
```

D.7.2 Texinfo

You can include LilyPond fragments in Texinfo; in fact, this entire manual is written in Texinfo.

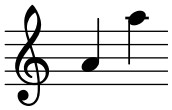
```
\input texinfo
@node Top

Texinfo text

@lilypond[verbatim,fragment,ragged-right]
a4 b c d
@end lilypond

More Texinfo text

@lilypond[verbatim,fragment,ragged-right]
d4 c b a
@end lilypond

@bye
```

Appendix E Cheat sheet

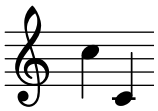
Syntax	Description	Example
`1 2 8 16`	durations	
`c4. c4..`	augmentation dots	
`c d e f g a b`	scale	
`fis bes`	alteration	
`\clef treble \clef bass`	clefs	
`\time 3/4 \time 4/4`	time signature	
`r4 r8`	rest	
`d ~ d`	tie	

`\key es \major` — key signature

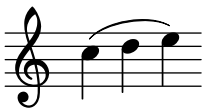
note`'` — raise octave



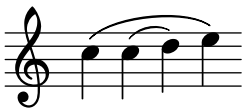
note`,` — lower octave



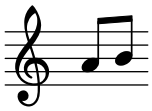
`c( d e)` — slur



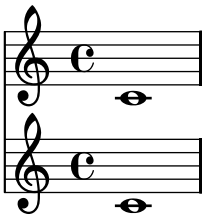
`c\( c( d) e\)` — phrasing slur



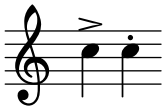
`a8[ b]` — beam



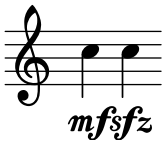
`<< \new Staff ... >>` — more staves



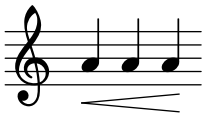
`c-> c-.` — articulations



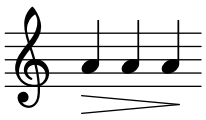
`c\mf c\sfz` dynamics



`a\< a a\!` crescendo

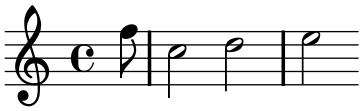


`a\> a a\!` decrescendo

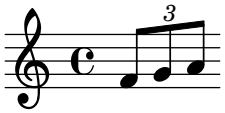


`< >` chord

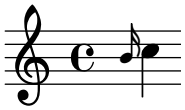
`\partial 8` upstep



`\times 2/3 {f g a}` triplets



`\grace` grace notes

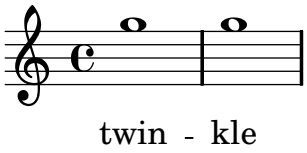


`\lyricmode { twinkle }` entering lyrics twinkle

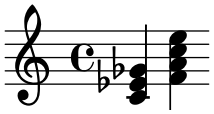
`\new Lyrics` printing lyrics

twinkle

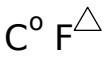
`twin -- kle` lyric hyphen



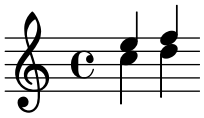
`\chordmode { c:dim f:maj7 }` chords



`\context ChordNames` printing chord names



`<<{e f} \\ {c d}>>` polyphony



`s4 s8 s16` spacer rests

Appendix F GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file

format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to

the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgments", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

F.0.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix G LilyPond command index

|

~

A

B

Appendix H LilyPond index

|

~

1

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W