GNU LilyPond

The music typesetter

The LilyPond development team

(For LilyPond version 2.5.11)

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.

Notes for version 2.4

The most important developments in 2.4 are related. In LilyPond 2.4 TEX is no longer strictly necessary to engrave music. This is because LilyPond can now also layout pages and determine page breaks. Another notable feature is the syntax, which has been simplified even further compared to previous versions.

Han-Wen and Jan

Utrecht/Eindhoven, The Netherlands, September 2004.

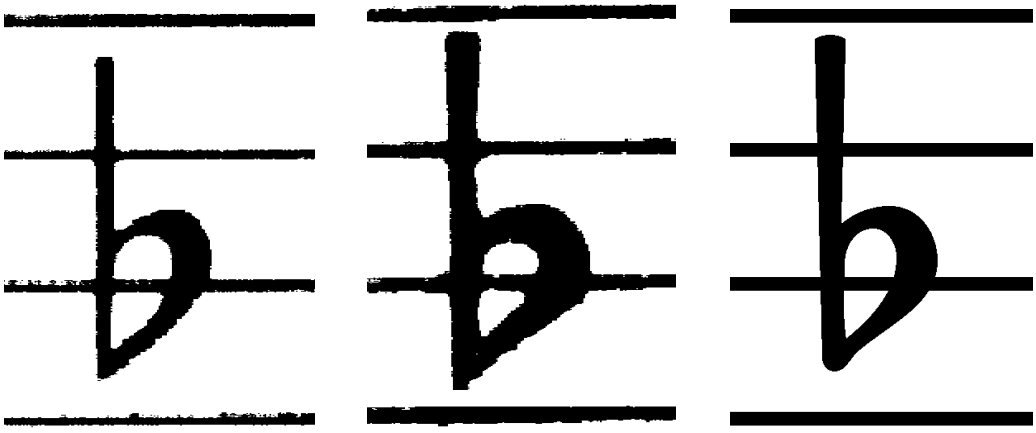
1 Introduction

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.

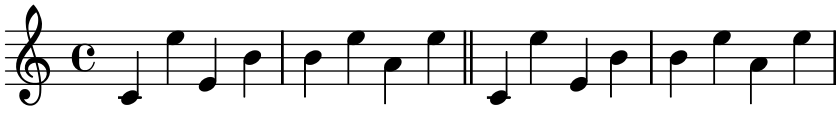
Nowadays, all newly printed music is produced with computers. This has obvious advantages; prints are cheaper to make, and editorial work can be delivered by email. Unfortunately, the pervasive use of computers has also decreased the graphical quality of scores. Computer printouts have a bland, mechanical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



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. It is printed once using exact mathematical spacing, and once with corrections. Can you spot which fragment is which?



The fragment only uses quarter notes: 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 first two measures are printed with this correction, the last two measures without. The notes in the last two measures form 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.

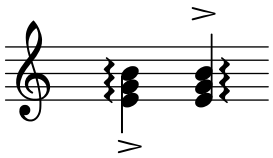
When we started, we wrote the LilyPond program entirely in the C++ programming language; the program's functionality was set in stone by the developers. That proved to be unsatisfactory for a number of reasons:

- 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.

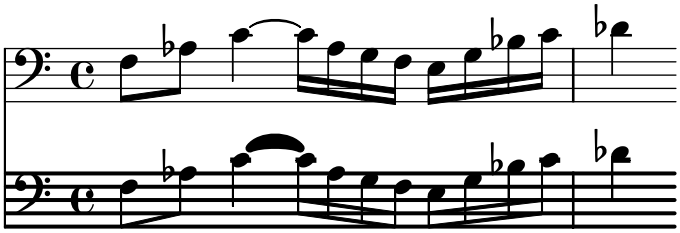
- Finally, we continually refine the formatting algorithms, so we need a flexible approach to rules. The C++ language forces a certain method of grouping rules that do not match well with how music notation works.

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.

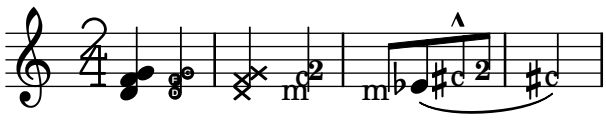
Scheme variables control layout decisions. For example, many graphical objects have a direction variable that encodes the choice between up and down (or left and right). Here you see two chords, with accents and arpeggios. In the first chord, the graphical objects have all directions down (or left).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

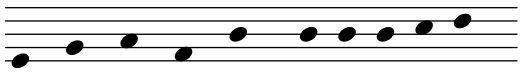
The formatting process decides where to place symbols. However, this can only be done once it is decided *what* symbols should be printed, in other words what notation to use.

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.

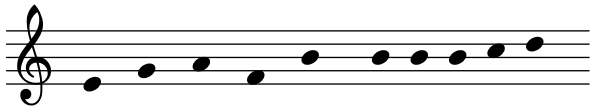
How can we get a grip on such a many-headed beast, and force it into the confines of a computer program? Our solution is to break up the problem of notation (as opposed to engraving, i.e., typography) into digestible and programmable chunks: every type of symbol is handled by a separate module, a so-called 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`.

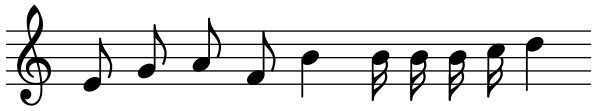
Then a `Staff_symbol_engraver` adds the staff



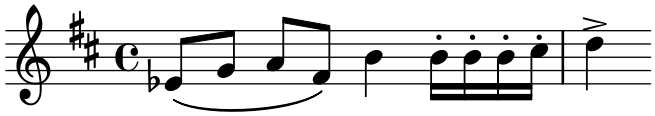
the `Clef_engraver` defines a reference point for the staff



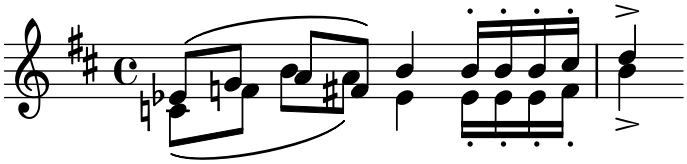
and the `Stem_engraver` adds stems.



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.



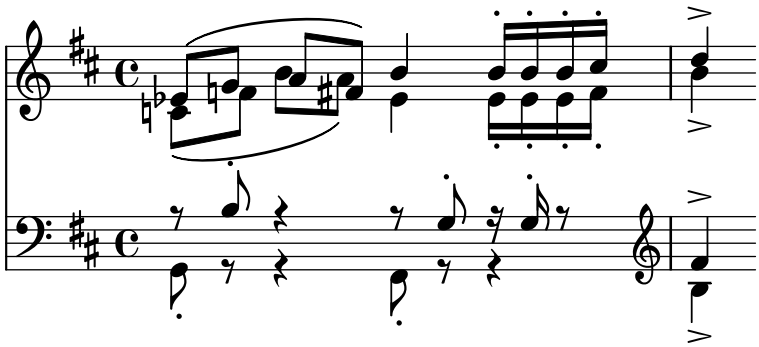
This system works well for monophonic music, but what about polyphony? In polyphonic notation, many voices can share a staff.



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`.



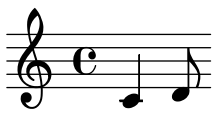
1.4 Music representation

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

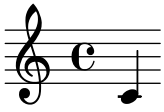
```
c'4 d'8
```

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



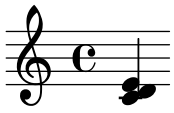
On a microscopic scale, such syntax is easy to use. On a larger scale, syntax also needs structure. How else can you enter complex pieces like symphonies and operas? The structure is formed by the concept of music expressions: by combining small fragments of music into larger ones, more complex music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

```
c4
```



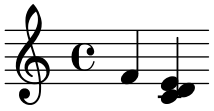
Chords can be constructed with << and >> enclosing the notes

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



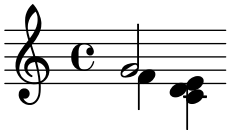
This expression is put in sequence by enclosing it in curly braces { ... }

```
{ 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>> } >>
```

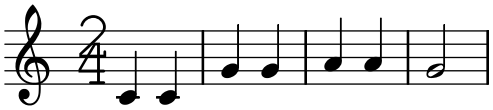


Such recursive structures can be specified neatly and formally in a context-free grammar. The parsing code is also generated from this grammar. In other words, the syntax of LilyPond is clearly and unambiguously defined.

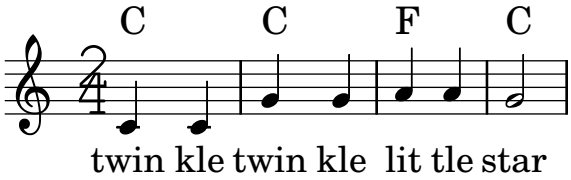
User-interfaces and syntax are what people see and deal with most. They are partly a matter of taste, and also subject of much discussion. Although discussions on taste do have their merit, they are not very productive. In the larger picture of LilyPond, the importance of input syntax is small: inventing 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.

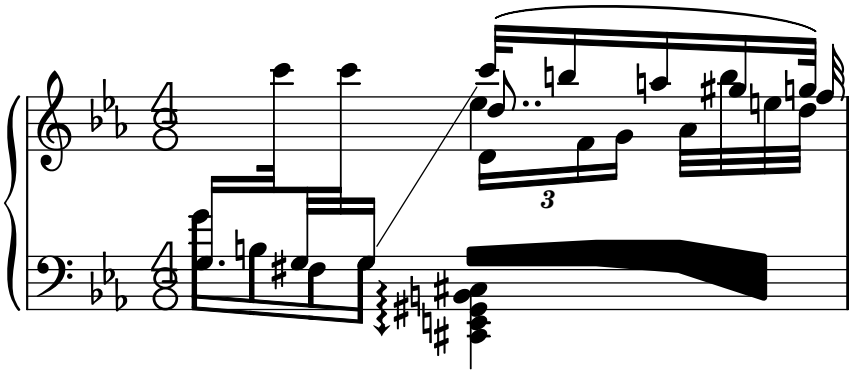


By adding chord names and lyrics we obtain a lead sheet.



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 10* gives a gentle introduction to typesetting music. First time users should start here.
- *Chapter 3 [Example templates], page 29* provides templates of LilyPond pieces. Just cut and paste a template into a file, add notes, and you're done!
- *Chapter 5 [Notation manual], page 66* discusses topics grouped by notation construct. Once you master the basics, this is the place to look up details.
- *Chapter 7 [Changing defaults], page 172* explains how to fine tune layout.
- *Chapter 4 [Running LilyPond], page 60* shows how to run LilyPond and its helper programs.
- *Chapter 9 [LilyPond-book], page 218* explains the details behind creating documents with in-line music examples (like this manual).
- *Chapter 10 [Converting from other formats], page 227* 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. In addition, this section explains how to upgrade input files from previous versions of LilyPond.
- *Appendix A [Literature list], page 232* contains a set of useful reference books for those who wish to know more on notation and engraving.

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 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.

- 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 '`lilypond-2.4.0/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/`.

[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.

2 Tutorial

This tutorial starts with a short introduction to the LilyPond music language. After this first contact we will show you how to produce printed output. Then you will be able to create and print your own sheets of music.

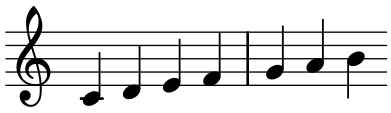
By cutting and pasting the full input into a test file, you have a starting template for experiments. If you like learning in this way, you will probably want to print out or bookmark Appendix F [Cheat sheet], page 257, which is a table listing all commands for quick reference.

2.1 First steps

The first example demonstrates how to enter the most elementary piece of music, a scale. A note can be entered by typing its name, from '`a`' through '`g`'. So, if you enter

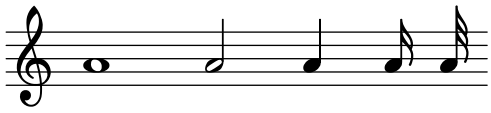
```
c d e f g a b
```

the result looks like this



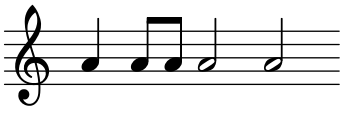
The duration of a note is specified by a number after the note name. '`1`' for a whole note, '`2`' for a half note, '`4`' for a quarter note and so on

```
a1 a2 a4 a16 a32
```



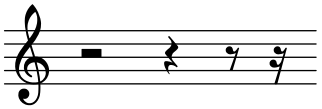
If you do not specify a duration, the duration last entered is used for the next notes. The duration of the first note in input defaults to a quarter

```
a a8 a a2 a
```



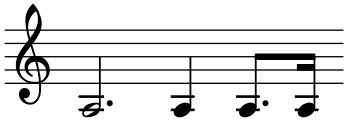
Rests are entered just like notes, but with the name '`r`'

```
r2 r4 r8 r16
```



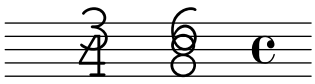
Add a dot '`.`' after the duration to get a dotted note

```
a2. a4 a8. a16
```



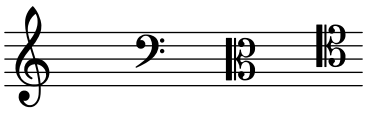
The meter (or time signature) can be set with the `\time` command

```
\time 3/4
\time 6/8
\time 4/4
```

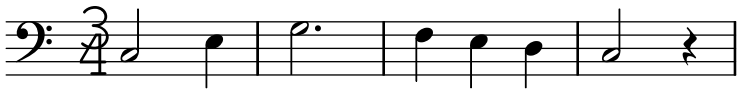


The clef can be set using the `\clef` command

```
\clef treble
\clef bass
\clef alto
\clef tenor
```



Remember to enclose the notes and commands in curly braces `{ ... }` to convert it to printable output.



For more elaborate information on

Entering pitches and durations
: see Section 5.1.2 [Pitches], page 66, and Section 5.1.8 [Durations], page 69.

Clefs
: see Section 5.3.3 [Clef], page 77.

Rests
: see Section 5.1.6 [Rests], page 68.

Time signatures and other timing commands
: see Section 5.3.5 [Time signature], page 79.

2.2 Running LilyPond for the first time

In the last section we explained what kind of things you can enter in a LilyPond file. In this section we will explain what commands to run and how to view or print the output. If you have not used LilyPond before, want to test your setup, or want to run an example file yourself, read this section. The instructions that follow are for Unix-like systems. Some additional instructions for Microsoft Windows are given at the end of this section.

Begin by opening a terminal window and starting a text editor. For example, you could open an xterm and execute `joe`.[1] In your text editor, enter the following input and save the file as '`test.ly`'

[1] 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`'.

```
{ c'4 e' g' }
```

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

```
lilypond test.ly
```

You will see something resembling

```
lilypond test.ly
GNU LilyPond 2.5.0
Processing 'test.ly'
Parsing...
Interpreting music... [1]
Preprocessing graphical objects...
Calculating line breaks... [2]
Layout output to 'test.tex'...
Converting to 'test.dvi'...
Converting to 'test.ps'...
Converting to 'test.pdf'...
```

The result is the file '`test.pdf`'[2] which you can print or view with the standard facilities of your operating system.[3]

On Windows, start up a text-editor[4] and enter

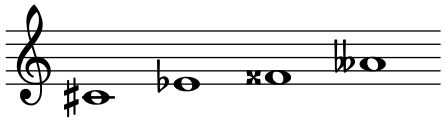
```
{ c'4 e' g' }
```

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.

2.3 More about pitches

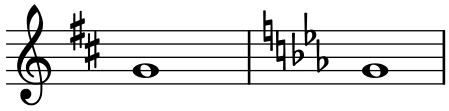
A sharp (FIXME: fetasharp) pitch is made by adding '`is`' to the name, a flat (FIXME: fetaflat) pitch by adding '`es`'. As you might expect, a double sharp or double flat is made by adding '`isis`' or '`eses`'[5]

```
cis1 ees fisis aeses
```



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

```
\key d \major
g1
\key c \minor
g
```



[2] For TEX aficionados: there is also a '`test.dvi`' file. It can be viewed with `xdvi`. The DVI uses a lot of PostScript specials, which do not show up in the magnifying glass. The specials also mean that the DVI file cannot be processed with `dvilj`. Use `dvips` for printing.

[3] If your system does not have any tools installed, you can try Ghostscript (`http://www.cs.wisc.edu/~ghost/`), a freely available package for viewing and printing PDF and PostScript files.

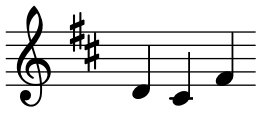
[4] Any simple or programmer-oriented editor will do, for example Notepad. Do not use a word processor, since these insert formatting codes that will confuse LilyPond.

[5] This syntax derived from note naming conventions in Nordic and Germanic languages, like German and Dutch.

Key signatures together with the pitches (including alterations) are used to determine when to print accidentals. 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 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

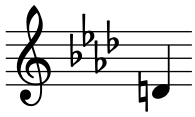


no note has an explicit accidental, but you still must enter

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

The code '`d`' does not mean 'print a black dot just below the staff.' Rather, it means: 'a note with pitch D-natural.' In the key of A-flat major, it does get an accidental

```
\key as \major
d
```



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 5.6 [Accidentals], page 89, for some examples how accidentals can be printed according to different rules.

For more information on

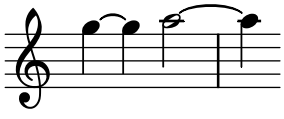
Accidentals
: see Section 5.6 [Accidentals], page 89.

Key signature
: see Section 5.3.2 [Key signature], page 76.

2.4 Entering ties

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

```
g4~ g a2~ a4
```

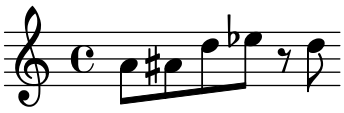


For more information on Ties see Section 5.1.12 [Ties], page 71.

2.5 Automatic and manual beams

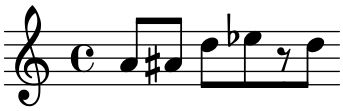
Beams are drawn automatically

```
a8 ais d es r d
```



If you do not like where beams are put, they can be entered by hand. Mark the first note to be beamed with '[' and the last one with ']'.

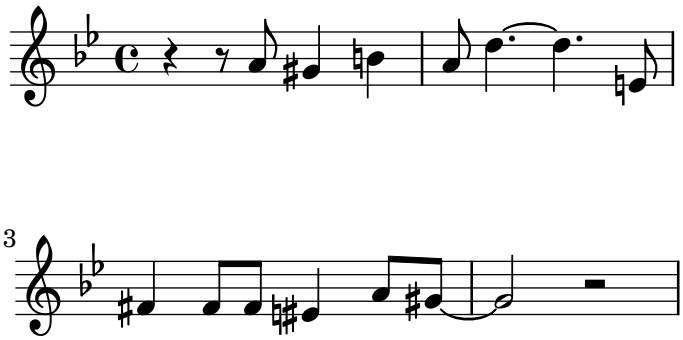
```
a8[ ais] d[ es r d]
```



For more information on beams, see Section 5.5 [Beaming], page 86.

Here are key signatures, accidentals and ties in action

```
\relative c'' {
  \time 4/4
  \key g \minor
  \clef treble
  r4 r8 a8 gis4 b
  a8 d4.~ d e,8
  fis4 fis8 fis8 eis4 a8 gis~
  gis2 r2
}
```

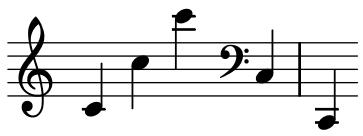


There are some interesting points to note in this example. Bar lines and beams are drawn automatically. Line breaks are calculated automatically; it does not matter where the line breaks are in the source file. Finally, the order in which time, key, and clef changes are entered is not relevant: in the printout, these are ordered according to standard notation conventions.

2.6 Octave entry

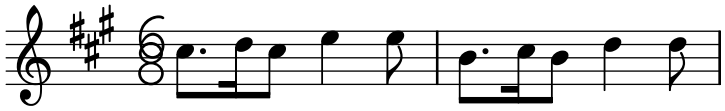
To raise a note by an octave, add a high quote ’ (apostrophe) to the note name, to lower a note one octave, add a ‘low quote’ , (comma). Middle C is `c’`

```
c’4 c’’ c’’’ \clef bass c c,
```



An example of the use of quotes is in the following Mozart fragment

```
\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
```

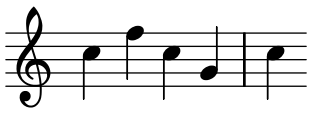


This example shows that music in a high register needs lots of quotes. This makes the input less readable, and it is a source of errors. The solution is to use ‘relative octave’ mode. This is the most convenient way to copy existing music.

In relative mode, a note without octavation quotes (i.e. the ’ or , after a note) is chosen so that it is closest to the previous one. For example, ‘`c f`’ goes up while ‘`c g`’ goes down.

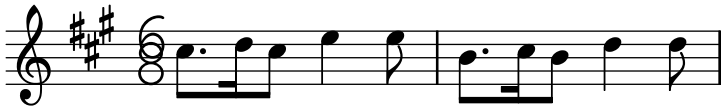
To use relative mode, add `\relative` before the piece of music. The first note is taken relative to the middle C (i.e., `c’`)

```
\relative {
  c’ f c g c
}
```



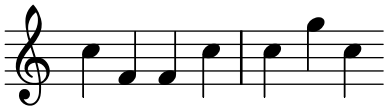
Since most music has small intervals, pieces can be written almost without octavation quotes in relative mode. The previous example is entered as

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



Larger intervals are made by adding octavation quotes.

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



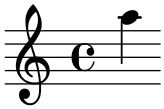
In summary, quotes or commas no longer determine the absolute height of a note in `\relative` mode. Rather, the height of a note is relative to the previous one, and changing the octave of a single note shifts all following notes an octave up or down.

For more information on relative octaves see Section 5.2.1 [Relative octaves], page 73, and Section 5.2.2 [Octave check], page 74.

2.7 Music expressions explained

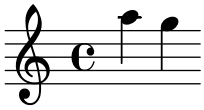
In input files, music is represent by so-called *music expression*s. We have already seen some in the previous examples; a single note is a music expression

```
a4
```



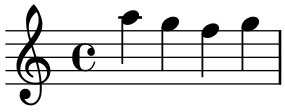
Enclosing a group of notes in braces creates a new music expression

```
{ a4 g4 }
```



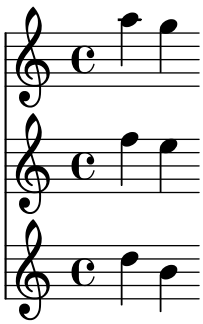
Putting a bunch of music expressions (e.g., notes) in braces, means that they should be played in sequence. The result again is a music expression, which can be grouped with other expressions sequentially. Here, the expression from the previous example is combined with two notes

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



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

```
<<
  { a4 g }
  { f e }
  { d b }
>>
```



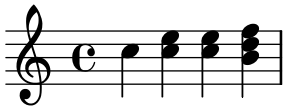
This mechanism is similar to mathematical formulas: a big formula is created by composing small formulas. Such formulas are called expressions, and their definition is recursive, so you can make arbitrarily complex and large expressions. For example,

```
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 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.

Note that this example only has one staff, whereas the previous example had three separate staves. That is because this example begins with a single note. To determine the number of staves, LilyPond looks at the first element. If it is a single note, there is one staff; if there is a simultaneous expression, there is more than one staff.

```
{
  c <<c e>>
  << { e f } { c <<b d>> } >>
}
```



Music files with deep nesting can be confusing to enter and maintain. One convention that helps against this confusion is indenting. When entering a file with deep nesting of braces and angles, it is customary to use an indent that indicates the nesting level. Formatting music like this eases reading and helps you insert the right number of closing braces at the end of an expression. For example,

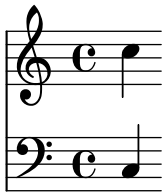
```
<<
  {
    ...
  }
  {
    ...
  }
>>
```

Some editors have special support for entering LilyPond, and can help indenting source files. See Section 4.6 [Editor support], page 63 for more information.

2.8 More staves

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 `>>`, as demonstrated here

```
<<
  \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. This ensures that each fragment gets its own stave.

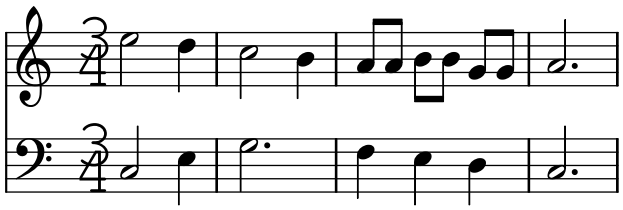
There are several types of contexts. `Staff`, `Voice` and `Score` handle melodic notation, while `Lyrics` sets lyric texts and `ChordNames` prints chord names.

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.

We can now typeset a melody with two staves

```
\relative <<
  \new Staff {
    \time 3/4
    \clef treble

    e'2 d4 c2 b4 a8[ a]
    b[ b] g[ g] a2.
  }
  \new Staff {
     \clef bass
     c,,2 e4 g2.
     f4 e d c2.
```

```
  }
>>
```

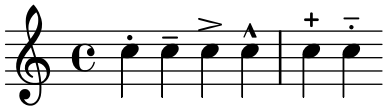


For more information on contexts see the description in Section 7.1 [Interpretation contexts], page 172.

2.9 Adding articulation marks to notes

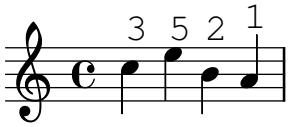
Common accents can be added to a note using a dash (‘-’) and a single character

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



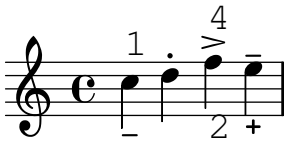
Similarly, fingering indications can be added to a note using a dash (‘-’) and the digit to be printed

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



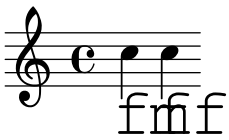
Articulations and fingerings are usually placed automatically, but you can specify a direction using ‘^’ (up) or ‘_’ (down). You can also use multiple articulations on the same note. In most cases, however, it is best to let LilyPond determine the articulation directions.

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



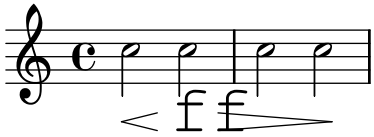
Dynamic signs are made by adding the markings (with a backslash) to the note

```
c\ff c\mf
```



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

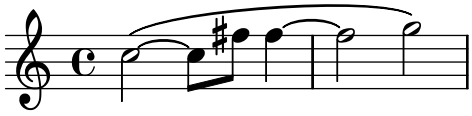
```
c2\< c2\ff\> c2 c2\!
```



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

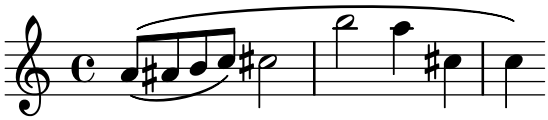
```
d4( c16) cis( d e c cis d) e( d4)
```

A slur looks like a tie, but it has a different meaning. A tie simply makes the first note sound 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



Slurs to indicate phrasing can be entered with `\(` and `\)`, so you can have both legato slurs and phrasing slurs at the same time.

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



For more information on

Fingering see Section 5.7.10 [Fingering instructions], page 97.

Articulations
see Section 5.7.8 [Articulations], page 95.

Slurs see Section 5.7.1 [Slurs], page 91.

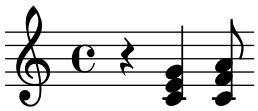
Phrasing slurs
see Section 5.7.2 [Phrasing slurs], page 92.

Dynamics see Section 5.7.13 [Dynamics], page 102.

2.10 Combining notes into chords

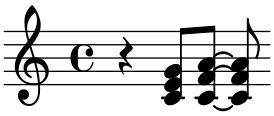
Chords can be made by surrounding pitches with angle brackets. Angle brackets are the symbols '<' and '>'.

```
r4 <c e g>4 <c f a>8
```

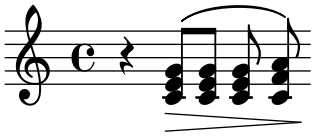


You can combine markings like beams and ties with chords. They must be placed outside the angled brackets

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



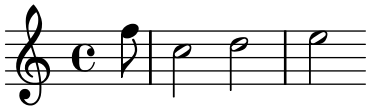
```
r4 <c e g>8\>( <c e g> <c e g> <c f a>\!)
```



2.11 Advanced rhythmic commands

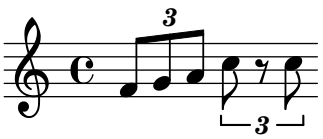
A pickup is entered with the keyword `\partial`. It is followed by a duration: `\partial 4` is a quarter note upstep and `\partial 8` an eighth note

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



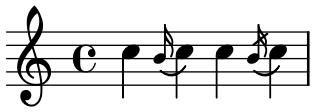
Tuplets are made with the `\times` keyword. It takes two arguments: a fraction and a piece of music. The duration of the piece of music is multiplied by the fraction. Triplets make notes occupy 2/3 of their notated duration, so a triplet has 2/3 as its fraction

```
\times 2/3 { f8 g a }
\times 2/3 { c r c }
```



Grace notes are also made by prefixing a music expression with the keyword `\appoggiatura` or `\acciaccatura`

```
c4 \appoggiatura b16 c4
c4 \acciaccatura b16 c4
```



For more information on

Grace notes
: see Section 5.7.11 [Grace notes], page 99,

Tuplets
: see Section 5.1.13 [Tuplets], page 72,

Pickups
: see Section 5.3.6 [Partial measures], page 80.

2.12 Commenting input files

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; after `%` the rest of the line is ignored. A block comment marks a whole section of music input. 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 is a special statement that is a kind of comment. The `\version` statement marks for which version of LilyPond the file was written. To mark a file for version 2.4.0, use

```
\version "2.4.0"
```

These annotations make future upgrades of LilyPond go more smoothly. Changes in the syntax are handled with a special program, '`convert-ly`' (see Section 10.1 [Invoking convert-ly], page 227), and it uses `\version` to determine what rules to apply.

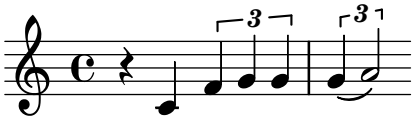
2.13 Printing lyrics

Lyrics are entered by separating each syllable with a space

```
I want to break free
```

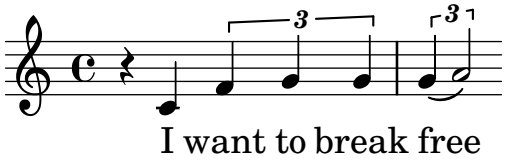
Consider the melody

```
\relative {
  r4 c \times 2/3 { f g g }
  \times 2/3 { g4( a2) }
}
```



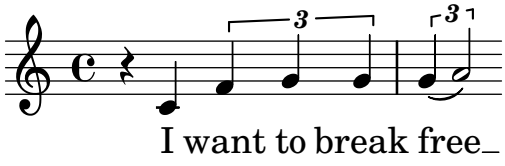
The lyrics can be set to these notes, combining both with the `\addlyrics` keyword

```
<<
  \relative {
    r4 c \times 2/3 { f g g }
    \times 2/3 { g4( a2) }
  }
  \addlyrics { I want to break free }
>>
```



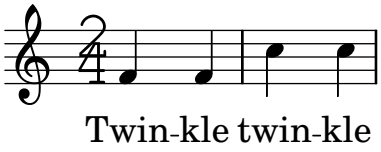
This melody ends on a melisma, a single syllable (‘free’) sung to more than one note. This is indicated with an *extender line*. It is entered as two underscores, i.e.,

```
{ I want to break free __ }
```



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

```
Twin -- kle twin -- kle
```

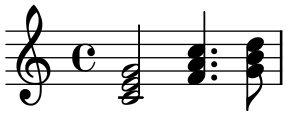


More options, like putting multiple lines of lyrics below a melody are discussed in Section 5.11 [Vocal music], page 116.

2.14 A lead sheet

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

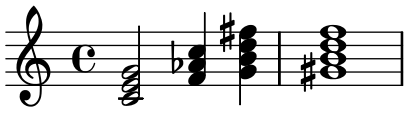
```
\chordmode { c2 f4. g8 }
```



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 }
```

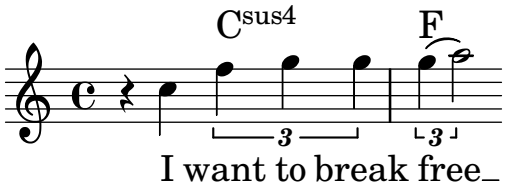


For lead sheets, chords are not printed on staves, but as names on a line for themselves. This is achieved by using `\chords` instead of `\chordmode`. This uses the same syntax as `\chordmode`, but renders the notes in a `ChordNames` context, with the following result.

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

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

```
<<
  \chords { chords }
  the melody
  \addlyrics { the text }
>>
}
```



A complete list of modifiers and other options for layout can be found in Section 5.1.5 [Chords], page 68.

2.15 Adding titles

Bibliographic information is entered in a separate block, the `\header` block. The name of the piece, its composer, etc., are entered as an assignment, within `\header { ... }`. The `\header` block is usually put at the top of the file. For example,

```
\header {
  title = "Miniature"
  composer = "Igor Stravinsky"
}

{ ... }
```

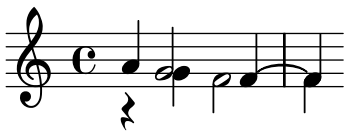
When the file is processed, the title and composer are printed above the music. More information on titling can be found in Section 7.5.11 [Creating titles], page 204.

2.16 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.

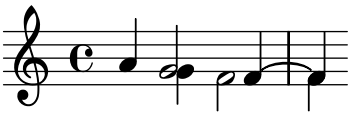
Entering such parts is done by entering each voice as a sequence (with `{...}`), and combining these simultaneously, separating the voices with `\\`

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



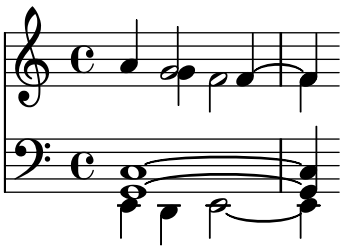
For polyphonic music typesetting, spacer rests can also be convenient; these are rests that do not print. They are useful for filling up voices that temporarily do not play. Here is the same example with a spacer rest (`s`) instead of a normal rest (`r`),

```
<< { 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 features of polyphonic typesetting are described in this manual in section Section 5.4 [Polyphony], page 84.

2.17 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 2.8 [More staves], page 18,

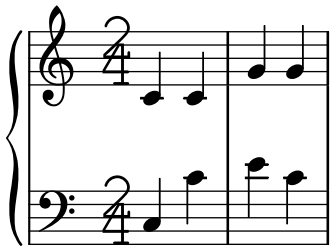
```
<< \new Staff { ... }
   \new Staff { ... } >>
```

but now this entire expression must be interpreted as a `PianoStaff`

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

Here is a small example

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



More information on formatting piano music is given in Section 5.10 [Piano music], page 111.

2.18 Organizing larger pieces

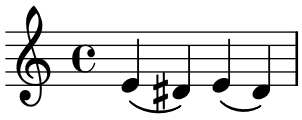
When all of the elements discussed earlier are combined to produce larger files, the `\score` blocks get a lot bigger, because the music expressions are longer, and, in the case of polyphonic pieces, more deeply nested. Such large expressions can become unwieldy.

By using variables, also known as identifiers, it is possible to break up complex music expressions. An identifier is assigned as follows

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

The contents of the music expression `namedMusic`, can be used later by preceding the name with a backslash, i.e., `\namedMusic`. In the next example, a two-note motive is repeated two times by using variable substitution

```
seufzer = {
  e'4( dis'4)
}
{ \seufzer \seufzer }
```



The name of an identifier should have alphabetic characters only; no numbers, underscores or dashes. The assignment should be outside of running music.

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
  linewidth = \width
}
{ c4^\name }
```

More information on the possible uses of identifiers is given in the technical manual, in Section 8.1.1 [Input variables and Scheme], page 209.

2.19 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
}
```

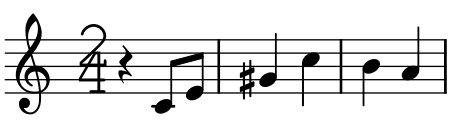
Then, an individual part is made by putting the following in a file

```
\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 downwards. 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



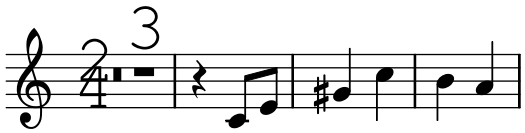
In ensemble pieces, one of the voices often does not play for many measures. This is denoted by a special rest, the multi-measure rest. It is entered with a capital '`R`' followed by a duration (1 for a whole note, 2 for a half note, etc.). By multiplying the duration, longer rests can be constructed.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

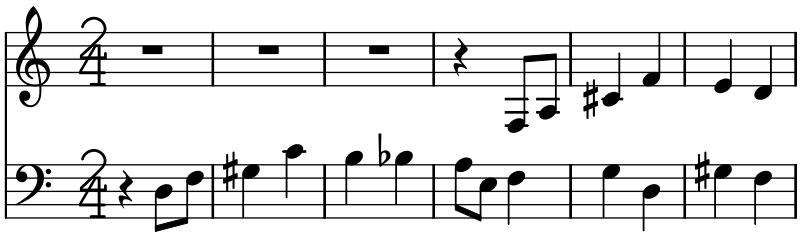
This command sets the property `skipBars` in the `Score` context to true (`##t`). Prepending the rest and this option to the music above, leads to the following result



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 5.15 [Orchestral music], page 133.

Setting run-time variables ('properties') is discussed in Section 7.1.2 [Changing context properties on the fly], page 174.

3 Example 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!

3.1 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?

The best answer is "however you want to do 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. That said, sometimes we make mistakes when writing files. If LilyPond can't understand your files, or produces output that you don't like, how do you fix the problem?

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

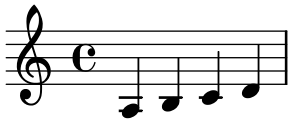
- Include `\version` numbers in every file. Note that all templates contain a `\version "2.4.0"` string. We highly recommend that you always include the `\version`, no matter how small your file is. Speaking from personal experience, it's quite frustrating to try to remember which version of LilyPond you were using a few years ago. `convert-ly` requires you to declare which version of LilyPond you used.
- Include checks: See Section 5.2.3 [Bar check], page 75 and Section 5.2.2 [Octave check], page 74. 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, every bar.
- One bar per line. 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, with either bar numbers (every so often) or references to musical themes ("second theme in violins", "fourth variation"). You may not need it when you're writing the piece for the first time, but if you want to go back and change something two or three years later, you won't know how your file is structured if you don't comment the file.

3.2 Single staff

3.2.1 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!

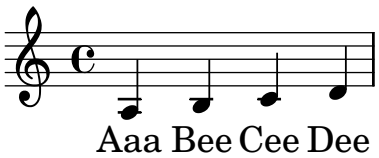
```
\version "2.4.0"
melody = \relative c' {
   \clef treble
   \key c \major
   \time 4/4

   a4 b c d
}

\score {
   \new Staff \melody
   \layout { }
   \midi { \tempo 4=60 }
}
```



3.2.2 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 "2.4.0"
melody = \relative c' {
   \clef treble
   \key c \major
   \time 4/4

   a4 b c d
}

text = \lyricmode {
   Aaa Bee Cee Dee
}

\score{
   <<
      \context Voice = one {
         \autoBeamOff
         \melody
      }
      \lyricsto "one" \new Lyrics \text
   >>
   \layout { }
   \midi { \tempo 4=60 }
}
```

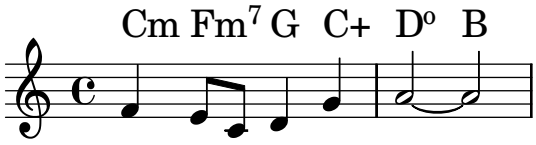


3.2.3 Notes and chords

Want to prepare a lead sheet with a melody and chords? Look no further!

```
\version "2.4.0"
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 {
   <<
      \context ChordNames {
         \set chordChanges = ##t
         \harmonies
      }
   \context Staff = one \melody
   >>

   \layout{ }
   \midi { \tempo 4=60}
}
```

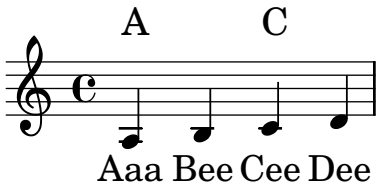


3.2.4 Notes, lyrics, and chords.

This template allows you to prepare a song with melody, words, and chords.

```
\version "2.4.0"
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 {
   <<
      \context ChordNames {
         \set chordChanges = ##t
         \harmonies
      }
   \context Voice = one {
      \autoBeamOff
      \melody
   }
   \lyricsto "one" \new Lyrics \text
```

```
   >>
   \layout { }
   \midi { \tempo 4=60 }
}
```

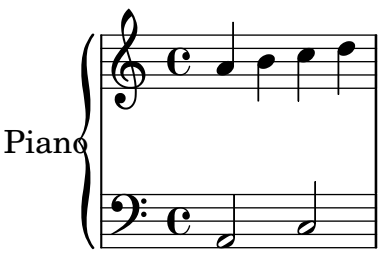


3.3 Piano templates

3.3.1 Solo piano

Here is a simple piano staff.

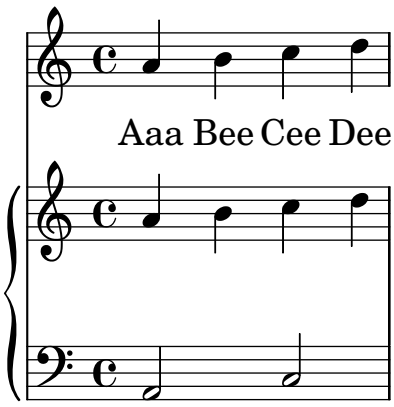
```
\version "2.4.0"
upper = \relative c'' {
   \clef treble
   \key c \major
   \time 4/4

   a b c d
}

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

   a2 c
}

\score {
   \context PianoStaff <<
      \set PianoStaff.instrument = "Piano  "
      \context Staff = upper \upper
      \context Staff = lower \lower
   >>
   \layout { }
   \midi { \tempo 4=60 }
}
```



3.3.2 Piano and melody with lyrics

Here is a typical song format: one staff with the melody and lyrics, with piano accompaniment underneath.

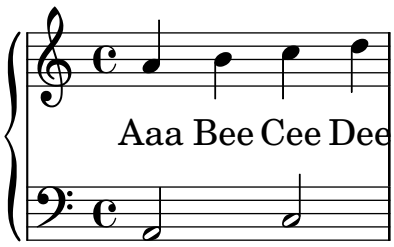
```
\version "2.4.0"
melody = \relative c'' {
   \clef treble
   \key c \major
   \time 4/4

   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 {
   <<
      \context Voice = mel {
          \autoBeamOff
          \melody
      }
      \lyricsto mel \new Lyrics \text

      \context PianoStaff <<
         \context Staff = upper \upper
         \context Staff = lower \lower
      >>
   >>
   \layout {
      \context { \RemoveEmptyStaffContext }
   }
   \midi { \tempo 4=60 }
}
```



3.3.3 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 "2.4.0"
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 {
  \context GrandStaff <<
    \context Staff = upper {
        \context Voice = singer \upper }
    \lyricsto "singer" \new Lyrics \text
    \context Staff = lower <<
      \clef bass
      \lower
    >>
  >>
  \layout {
    \context { \GrandStaff \accepts "Lyrics" }
    \context { \Lyrics \consists "Bar_engraver" }
  }
  \midi { \tempo 4=60 }
}
```

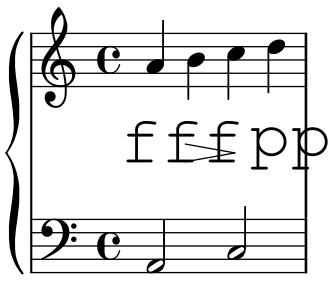


3.3.4 Piano centered dynamics

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 "2.4.0"
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 {
  \context PianoStaff <<
    \context Staff=upper \upper
    \context Dynamics=dynamics \dynamics
    \context Staff=lower <<
      \clef bass
      \lower
    >>
    \context Dynamics=pedal \pedal
  >>
  \layout {
    \context {
      \type "Engraver_group_engraver"
      \name Dynamics
      \alias Voice % So that \cresc works, for example.
```

```
      \consists "Output_property_engraver"

      minimumVerticalExtent = #'(-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
    }
  }
  \midi {
    \context {
      \type "Performer_group_performer"
      \name Dynamics
      \consists "Piano_pedal_performer"
      \consists "Span_dynamic_performer"
      \consists "Dynamic_performer"
    }
    \context {
      \PianoStaff
      \accepts Dynamics
    }
  }
}
```

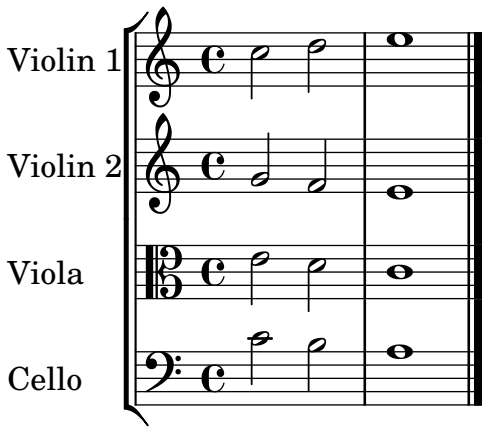


3.4 String quartet

3.4.1 String quartet

This template demonstrates a string quartet. It also uses a `\global` section for time and key signatures.

```
\version "2.4.0"

global= {
  \time 4/4
  \key c \major
}

violinOne = \new Voice { \relative c''{
  \set Staff.instrument = "Violin 1 "

  c2 d e1

\bar "|." }}
violinTwo = \new Voice { \relative c''{
  \set Staff.instrument = "Violin 2 "

  g2 f e1

\bar "|." }}
viola = \new Voice { \relative c' {
  \set Staff.instrument = "Viola "
  \clef alto

  e2 d c1

\bar "|." }}
cello = \new Voice { \relative c' {
  \set Staff.instrument = "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 { \tempo 4=60}
}
```



3.4.2 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 "2.4.0"

global= {
  \time 4/4
  \key c \major
}

Violinone = \new Voice { \relative c''{
  \set Staff.instrument = "Violin 1 "

  c2 d e1

\bar "|." }}   %*********************************
Violintwo = \new Voice { \relative c''{
  \set Staff.instrument = "Violin 2 "

  g2 f e1

\bar "|." }}   %*********************************
Viola = \new Voice { \relative c' {
  \set Staff.instrument = "Viola "
  \clef alto

  e2 d c1

\bar "|." }}   %*********************************
Cello = \new Voice { \relative c' {
  \set Staff.instrument = "Cello     "
  \clef bass

  c2 b a1
```

```
\bar "|."}}   %*********************************

music = {
  <<
    \tag #'(score vn1) \new Staff { << \global \Violinone >> }
    \tag #'(score vn2) \new Staff { << \global \Violintwo>> }
    \tag #'(score vla) \new Staff { << \global \Viola>> }
    \tag #'(score vlc) \new Staff { << \global \Cello>> }
  >>
}



%%%%% score.ly
\version "2.4.0"
\include "piece.ly"
#(set-global-staff-size 14)
\score {
  \new StaffGroup \keepWithTag #'score \music
  \layout { }
  \midi { \tempo 4 = 60 }
}


%%%%% vn1.ly
\version "2.4.0"
\include "piece.ly"
\score {
  \keepWithTag #'vn1 \music
  \layout { }
}


%%%%% vn2.ly
\version "2.4.0"
\include "piece.ly"
\score {
  \keepWithTag #'vn2 \music
  \layout { }
}


%%%%% vla.ly
\version "2.4.0"
\include "piece.ly"
\score {
  \keepWithTag #'vla \music
  \layout { }
}


%%%%% vlc.ly
```

```
\version "2.4.0"
\include "piece.ly"
\score {
  \keepWithTag #'vlc \music
  \layout { }
}
```

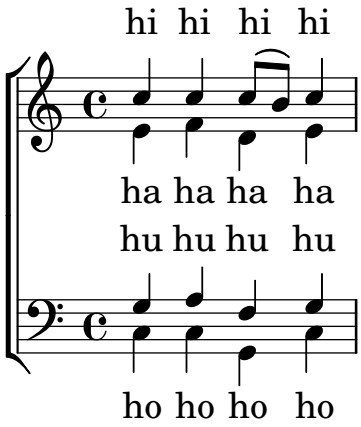
3.5 Vocal ensembles

3.5.1 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 "2.4.0"
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 {
   \context ChoirStaff <<
      \context Lyrics = sopranos { s1 }
      \context Staff = women <<
```

```
            \context Voice =
              sopranos { \voiceOne << \global \sopMusic >> }
            \context Voice =
              altos { \voiceTwo << \global \altoMusic >> }
         >>
         \context Lyrics = altos { s1 }
         \context Lyrics = tenors { s1 }
         \context Staff = men <<
            \clef bass
            \context Voice =
              tenors { \voiceOne <<\global \tenorMusic >> }
            \context Voice =
              basses { \voiceTwo <<\global \bassMusic >> }
         >>
         \context 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 minimumVerticalExtent = #'(-3 . 3)
         }
      }
   }
```



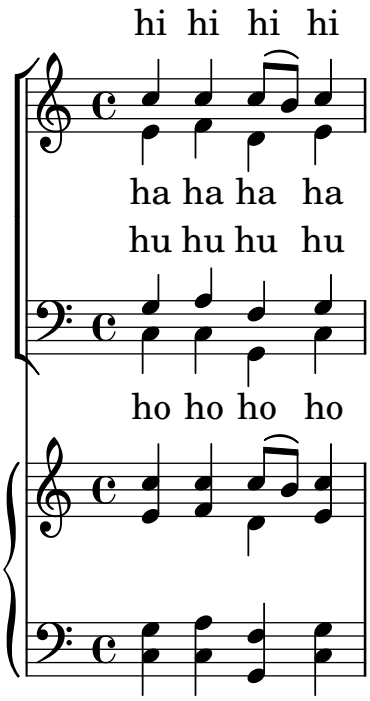
3.5.2 SATB vocal score and automatic piano reduction

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 "2.4.0"
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
}

\layout {
  \context {
         % a little smaller so lyrics
         % can be closer to the staff
         \Staff
          minimumVerticalExtent = #'(-3 . 3)
  }
}

<<
  \context ChoirStaff
  <<
    \context Lyrics = sopranos { s1 }
    \context Staff = women <<
      \context Voice =
        sopranos { \voiceOne << \global \sopMusic >> }
      \context Voice =
        altos { \voiceTwo << \global \altoMusic >> }
    >>
    \context Lyrics = altos { s1 }
    \context Lyrics = tenors { s1 }
```

```
    \context Staff = men <<
      \clef bass
      \context Voice =
        tenors { \voiceOne <<\global \tenorMusic >> }
      \context Voice =
        basses { \voiceTwo <<\global \bassMusic >> }
    >>
    \context 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 >>
    >>
  >>
>>
```



3.6 Ancient notation templates

3.6.1 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 where 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.

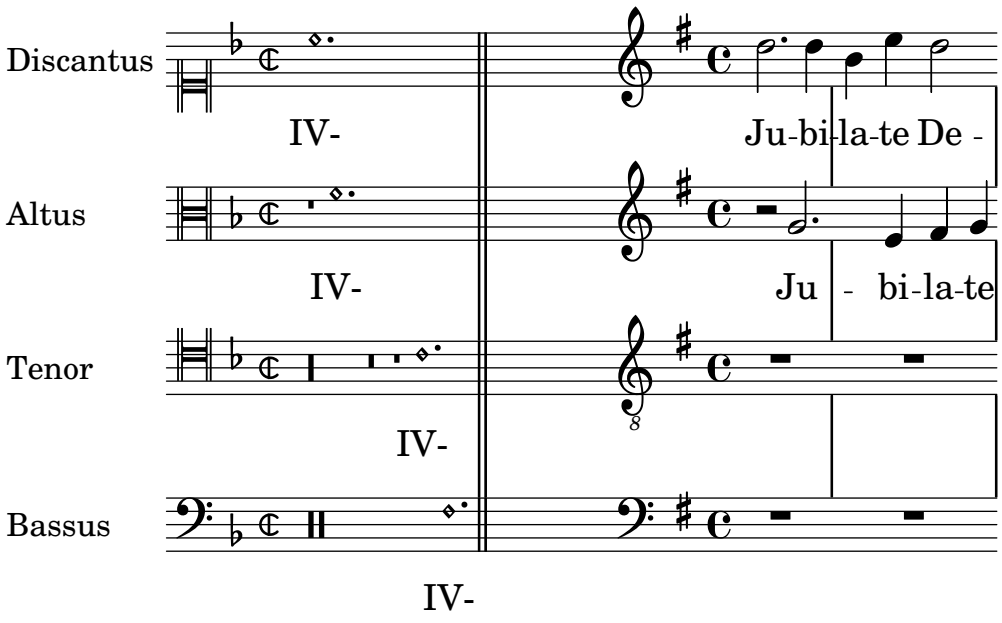
```
\version "2.4.0"

global = {
  % incipit
  \once \override Score.SystemStartBracket #'transparent = ##t
  \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
  \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*5

  % last bar contains a brevis (i.e., spans 2 bars);
  % therefore do not draw this particular bar
  \cadenzaOn
  \skip 1*2
  \cadenzaOff
```

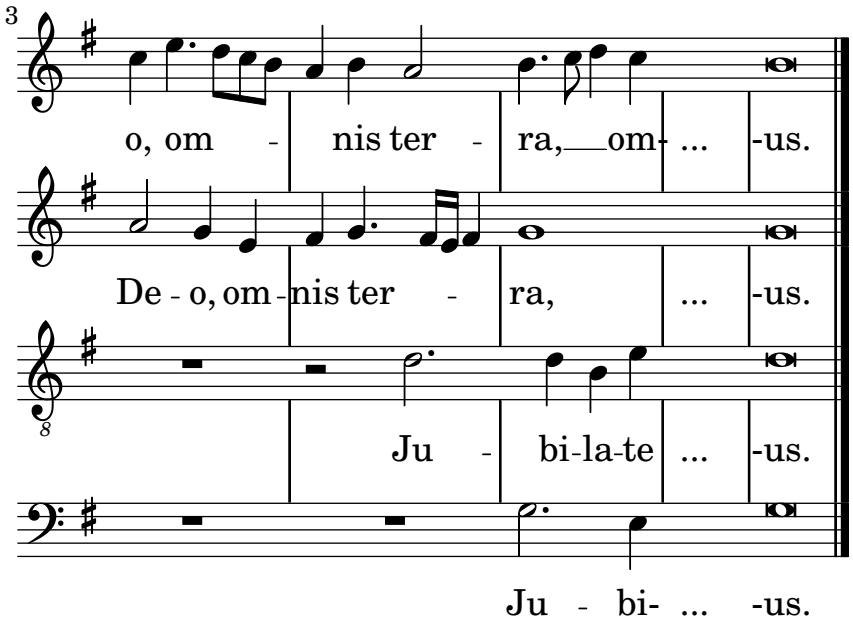
```
  % let finis bar go through all staves
  \override Staff.BarLine #'transparent = ##f

  % finis bar
  \bar "|."
}

discantusNotes = {
  \transpose c' c'' {
    \set Staff.instrument = "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.instrument = "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.instrument = "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.instrument = "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 {
  \context StaffGroup = choirStaff <<
    \context Voice =
      discantusNotes << \global \discantusNotes >>
    \context Lyrics =
      discantusLyrics \lyricsto discantusNotes { \discantusLyrics }
    \context Voice =
      altusNotes << \global \altusNotes >>
    \context Lyrics =
      altusLyrics \lyricsto altusNotes { \altusLyrics }
    \context Voice =
      tenorNotes << \global \tenorNotes >>
    \context Lyrics =
```

```
      tenorLyrics \lyricsto tenorNotes { \tenorLyrics }
    \context Voice =
      bassusNotes << \global \bassusNotes >>
    \context Lyrics =
      bassusLyrics \lyricsto bassusNotes { \bassusLyrics }
  >>
  \layout {
    \context {
      \Score
      \override BarLine #'transparent = ##t
      \remove "System_start_delimiter_engraver"
    }
    \context {
      \Voice
      \override Slur #'transparent = ##t
    }
  }
}
```





3.7 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 "2.4.0"
\header {
  title = "Song"
  subtitle = "(tune)"
  composer = "Me"
  meter = "moderato"
  piece = "Swing"
  tagline = "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.instrument = #"Trumpet"
  \clef treble
  \context Staff <<
    \trpt
  >>
}

% ------ Alto Saxophone ------
alto = \transpose c a \relative c' {
  \Key
  c1 c c
}
altoharmony = \transpose c' a {
  \jzchords
}
altosax = {
  \global
  \set Staff.instrument = #"Alto Sax"
  \clef treble
  \context Staff <<
    \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.instrument = #"Bari Sax"
  \clef treble
  \context Staff <<
    \bari
  >>
}

% ------ Trombone ------
tbone = \relative c {
  \Key
  c1 c c
}
tboneharmony = \chordmode {
  \jzchords
}
trombone = {
  \global
  \set Staff.instrument = #"Trombone"
  \clef bass
  \context Staff <<
    \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.instrument = #"Guitar"
  \clef treble
  \context Staff <<
    \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"
  \context Staff <<
    \context Voice = one \rhUpper
    \context Voice = two \rhLower
  >>
}
PianoLH = {
  \clef bass
  \global
  \set Staff.midiInstrument = "acoustic grand"
  \context Staff <<
    \context Voice = one \lhUpper
    \context Voice = two \lhLower
  >>
}

piano = {
  \context PianoStaff <<
    \set PianoStaff.instrument = #"Piano"
    \context Staff = upper \PianoRH
    \context Staff = lower \PianoLH
  >>
}

% ------ Bass Guitar ------
Bass = \relative c {
  \Key
  c1 c c
}
```

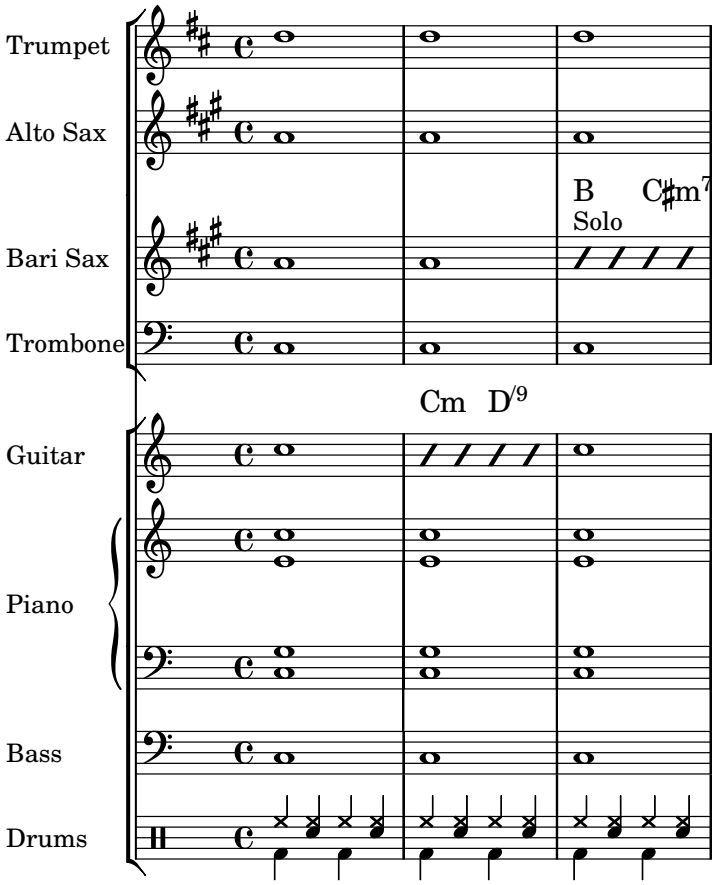
```
bass = {
  \global
  \set Staff.instrument = #"Bass"
  \clef bass
  \context Staff <<
    \Bass
  >>
}

% ------ 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.instrument = #"Drums"
    \new DrumVoice { \voiceOne \up }
    \new DrumVoice { \voiceTwo \down }
  >>
}

%%%%%%%% It All Goes Together Here %%%%%%%%%%%%%%%%%%%%%%

\score {
  <<
    \context StaffGroup = horns <<
      \context Staff = trumpet \trumpet
      \context Staff = altosax \altosax
      \context ChordNames = barichords \bariharmony
      \context Staff = barisax \barisax
      \context Staff = trombone \trombone
    >>

    \context StaffGroup = rhythm <<
      \context ChordNames = chords \gtrharmony
      \context Staff = guitar \guitar
      \context PianoStaff = piano \piano
      \context Staff = bass \bass
      \new DrumStaff { \drumContents }
    >>
  >>

  \layout {
```

```
    \context { \RemoveEmptyStaffContext }
    \context {
      \Score
      \override BarNumber #'padding = #3
      \override RehearsalMark #'padding = #2
      skipBars = ##t
    }
  }

  \midi { \tempo 4 = 75 }
}
```

Song
(tune)

Me

Swing



3.8 Other templates

3.8.1 All headers

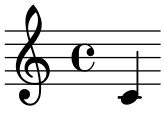
This template displays all available headers. Some of them are only used in the Mutopia project; they don't affect the printed output at all. They are used if you want the piece to be listed with

different information in the Mutopia database than you wish to have printed on the music. For example, Mutopia lists the composer of the famous D major violin concerto as TchaikovskyPI, whereas perhaps you wish to print "Petr Tchaikowski" on your music.

The ‘linewidth’ is for \header.

```
\version "2.4.0"
\header {
  dedication = "dedication"
  title = "Title"
  subtitle = "Subtitle"
  subsubtitle = "Subsubtitle"
  composer = "Composer (xxxx-yyyy)"
  opus = "Opus 0"
  piece = "Piece I"
  instrument = "Instrument"
  arranger = "Arranger"
  poet = "Poet"
  texttranslator = "Translator"
  copyright = "public domain"

  % These are headers used by the Mutopia Project
  % http://www.mutopiaproject.org/
  mutopiatitle = ""
  mutopiacomposer = ""
  mutopiapoet = ""
  mutopiainstrument = ""
  date = "composer's dates"
  source = "urtext "
  maintainer = "your name here"
  maintainerEmail = "your email here"
  maintainerWeb = "your home page"
  lastupdated = "2004/Aug/26"
}

\score {
  \header {
    piece = "piece1"
    opus = "opus1"
  }
  { c'4 }
}

\score {
  \header {
    piece = "piece2"
    opus = "opus2"
  }
  { c'4 }
}
```

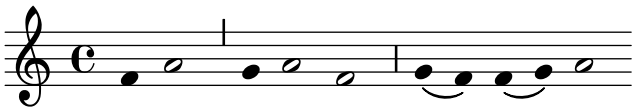
piece2 opus2



3.8.2 Gregorian template

This example demonstrates how to do modern transcriptions of Gregorian music. Gregorian music has no measure, no stems; it uses only half and quarter notes, and two types of barlines, a short one indicating a rest, and a second one indicating a breath mark.

```
barOne = { \once \override Staff.BarLine #'bar-size = #2
  \bar "|" }
barTwo = { \once \override Staff.BarLine #'extra-offset = #'(0 . 2)
  \once \override Staff.BarLine #'bar-size = #2
  \bar "|" }
chant = \relative c' {
  \set Score.timing = ##f
  \override Staff.Stem #'transparent = ##t

  f4 a2 \barTwo
  g4 a2 f2 \barOne
  g4( f) f( g) a2
}
\score {
  \chant
  \layout{ }
  \midi { \tempo 4=60 }
}
```

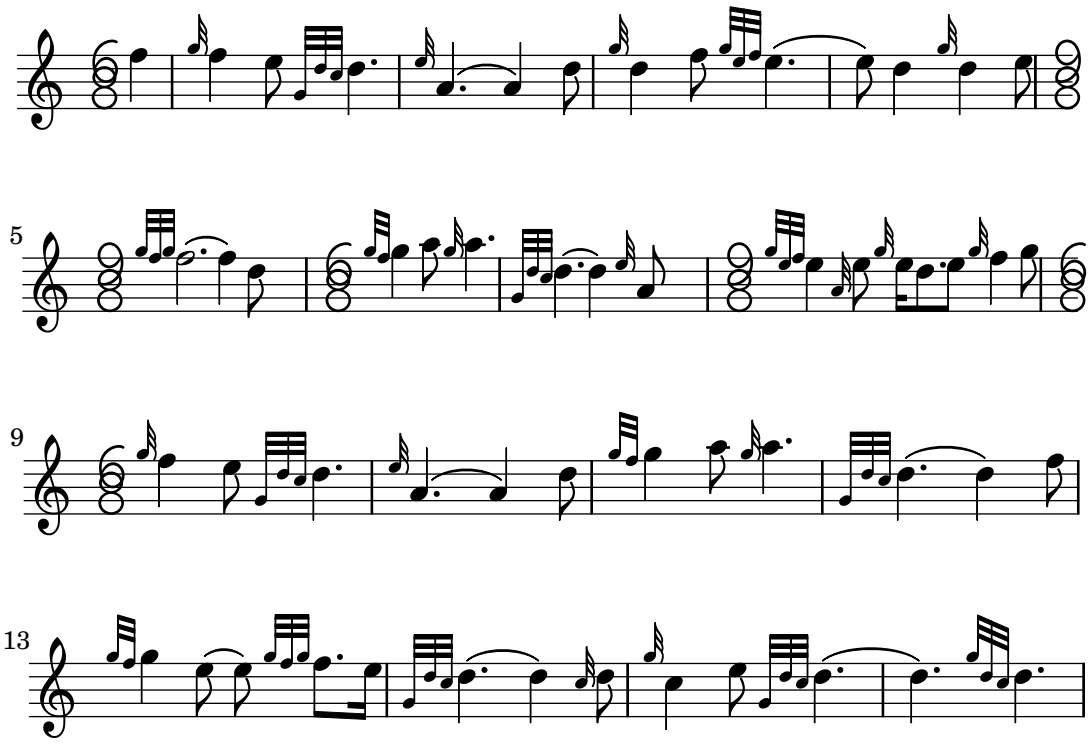


3.8.3 Bagpipe music

Here is an example of bagpipe music. It demonstrates a big strength of LilyPond, compared to graphical score editors: in LilyPond, you can very easily reuse small segments of music without writing them out completely. This template defines a large number of small segments (`taor`, `grip`, `thrd`, etc), which can be reused easily.

```
taor = { \grace { g32[ d' g e'] } }
grip = { \grace { g32[ b g ]    } }
thrd = { \grace { g32[ d' c']   } }
birl = { \grace { g32[ a g]     } }
gstd = { \grace { g'32[ d' g]   } }
fgg  = { \grace { f32[ g'32]    } }
dblb = { \grace { g'32[ b d']   } }
dblc = { \grace { g'32[ c' d']  } }
dble = { \grace { g'32[ e' f']  } }
dblf = { \grace { g'32[ f' g']  } }
```

```
dblg = { \grace { g'32[ f']     } }
dbla = { \grace { a'32[ g']     } }
lgg  = { \grace { g32  } }
lag  = { \grace { a32  } }
cg   = { \grace { c'32 } }
eg   = { \grace { e'32 } }
gg   = { \grace { g'32 } }
dg   = { \grace { d'32 } }
hag  = { \grace { a'32 } }
gefg = { \grace { g'32[ e' f']  } }
efg  = { \grace { e'32[ f']     } }
gdcg = { \grace { g'32[ d' c']  } }
gcdg = { \grace { g'32[ c' d']  } }

\transpose a a' {
  #(add-grace-property 'Voice 'Stem 'length 6)
  \time 6/8 \partial 4
  \tieUp
  \slurUp

  f'4 |
  \gg f'4 e'8 \thrd d'4. |
  \eg a4.(a4) d'8 |
  \gg d'4 f'8 \dble e'4. ( | \noBreak
  e'8) d'4 \gg d'4 e'8 |

  \break
  \time 9/8
  \dblf f'2.( f'4) d'8 |
  \time 6/8
  \dblg g'4 a'8 \gg a'4. |
  \thrd d'4.( d'4) \eg a8 |
  \time 9/8
  \dble e'4 \lag e'8 \gg e'16[ d'8. e'8] \gg f'4 g'8 |

  \break
  \time 6/8
  \gg f'4 e'8 \thrd d'4. |
  \eg a4.( a4) d'8 |
  \dblg g'4 a'8 \gg a'4. |
  \thrd d'4.( d'4) f'8 |

  \break
  \dblg g'4 e'8( e'8) \dblf f'8.[ e'16] |
  \thrd d'4.( d'4) \cg d'8 |
  \gg c'4 e'8 \thrd d'4.( |
  d'4.) \gdcg d'4.
}
```



3.9 Lilypond-book templates

These templates are for use with `lilypond-book`. If you're not familiar with this program, please refer to Chapter 9 [LilyPond-book], page 218.

3.9.1 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}
```

3.9.2 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,raggedright]
a4 b c d
@end lilypond

More Texinfo text

@lilypond[verbatim,fragment,raggedright]
d4 c b a
@end lilypond

@bye
```

4 Running LilyPond

This chapter details the technicalities of running LilyPond.

4.1 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]

4.2 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 function `ly:set-option` allows access to some internal variables. Use `-e '(ly:option-usage)'` for more information.

`-f,--format=`*format*
: which formats should be written. Choices are `svg`, `ps`, `pdf`, `png`, `tex`, `dvi`.

`-b,--backend=`*format*
: the output format to use for the back-end. Choices 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.

  `ps` for PostScript

  `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)

  `scm` for a dump of the raw, internal Scheme-based drawing commands.

`-h,--help`
: Show a summary of usage.

`--include, -I=`*directory*
: Add *directory* to the search path for input files.

[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.

`-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., `-f tex`.

`--png`
: Generate pictures of each page, in PNG format. This implies `--ps`.

`--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, the `--safe` **MUST** be passed. This 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.

`-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**!)

4.3 Environment variables

For processing both the TEX and the PostScript output, the appropriate environment variables must be set. The following scripts do this:

- '`buildscripts/out/lilypond-profile`' (for SH shells)
- '`buildscripts/out/lilypond-login`' (for C-shells)

They should normally be sourced as part of the login process. If these scripts are not run from the system wide login process, then you must run them yourself.

If you use sh, bash, or a similar shell, then add the following to your '`.profile`':

```
. /the/path/to/lilypond-profile
```

If you use csh, tcsh or a similar shell, then add the following to your '`~/.login`':

```
source /the/path/to/lilypond-login
```

Of course, in both cases, you should substitute the proper location of either script.

These scripts set the following variables:

`TEXMF` To make sure that TEX and lilypond find data files (among others '`.tex`', '`.mf`', and '`.tfm`'), you have to set `TEXMF` to point to the lilypond data file tree. A typical setting would be

```
{/usr/share/lilypond/2.4.0,{!!/usr/share/texmf}}
```

The binary itself recognizes the following environment variables:

`LILYPONDPREFIX`
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.

4.4 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 occured, 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.

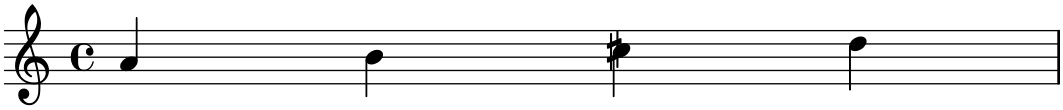
4.5 Reporting bugs

If you have input that results in a crash or an erroneous output, then that is a bug. We try to respond to bug-reports promptly, and fix them as soon as possible. Help us by sending a defective input file, so we can reproduce the problem. Make it small, so we can easily debug the problem. Don't forget to tell which version of LilyPond you use! Send the report to `bug-lilypond@gnu.org`.

When you've found a bug, have a look at our bug database (`http://lilypond.org/doc/v2.3/bugs/`) to see if it has already been reported. You could also try to do a few searches on the mailing list for the bug. Sometimes the bug will have already been reported and a fix or workaround is already known.

Here is an example of a good bug report:

```
It seems that placement of accidentals is broken.  In the
following example, the accidental touches the note head.

Using Mac OSX 10.3.5, fink package lilypond-unstable

\version "2.3.22"
\relative c''{
   a4 b cis d
}
```



4.6 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 (`http://www.vim.org`), a '`vimrc`' is supplied, along with syntax coloring tools. For more information, refer to the installation instructions.

JEdit
: The jEdit (`http://www.jedit.org/`) editor has a LilyPond plugin. This plugin includes a DVI viewer, integrated help and viewing via GhostScript. It can be installed by doing ⟨Plugins > Plugin Manager⟩, and selecting `LilyTool` from the ⟨Install⟩ tab.

All these editors can be made to jump into the input file to the source of a symbol in the graphical output. See Appendix D [Point and click], page 249.

4.7 Invoking lilypond-latex

Before LilyPond 2.4, the `lilypond` program only generated music notation. Titles and page layout was done in a separate wrapper program. For compatibility with older files, this wrapper program has been retained as `lilypond-latex`. It uses the LilyPond program and LaTeX to create a nicely titled piece of sheet music. Use of this program is only necessary if the input file contains special LaTeX options or formatting codes in markup texts.

The `lilypond-latex` wrapper is invoked from the command-line as follows

`lilypond-latex [`*`option`*`]... `*`file`*`...`

To have `lilypond-latex` read from stdin, use a dash (`-`) for *file*. The program supports the following options.

`-k,--keep`
: Keep the temporary directory with all output files. The temporary directory is created in the current directory as `lilypond.dir`.

`-h,--help`
: Print usage help.

`-I,--include=`*`dir`*
: Add *dir* to LilyPond's include path.

`-o,--output=`*`file`*
: Generate output to *file*. The extension of *file* is ignored.

`--png`
: Also generate pictures of each page, in PNG format.

`--preview`
: Also generate a picture of the first system of the score.

`-s,--set=`*`key=val`*
: Add *key*= *val* to the settings, overriding those specified in the files. Possible keys: `language`, `latexheaders`, `latexpackages`, `latexoptions`, `papersize`, `linewidth`, `orientation`, `textheight`.

`-v,--version`
: Show version information.

`-V,--verbose`
: Be verbose. This prints out commands as they are executed, and more information about the formatting process is printed.

`--debug`
: Print even more information. This is useful when generating bug reports.

`-w,--warranty`
: Show the warranty with which GNU LilyPond comes. (It comes with **NO WARRANTY**!)

4.7.1 Additional parameters

The `lilypond` program responds to several parameters specified in a `\layout` section of the input file. They can be overridden by supplying a `--set` command line option.

`language`
: Specify LaTeX language: the `babel` package will be included. Default: unset.

  Read from the `\header` block.

`latexheaders`
: Specify additional LaTeX header files. Normally read from the `\header` block. Default value: empty.

`latexpackages`
: Specify additional LaTeX package files. This works cumulative, so you can add multiple packages using multiple `-s=latexpackages` options. Normally read from the `\header` block. Default value: `geometry`.

`latexoptions`
: Specify additional options for the LaTeX `\documentclass`. You can put any valid value here. This was designed to allow `lilypond` to produce output for double-sided paper, with balanced margins and page numbers on alternating sides. To achieve this specify `twoside`.

`orientation`
: Set orientation. Choices are `portrait` or `landscape`. Is read from the `\layout` block, if set.

`textheight`
: The vertical extension of the music on the page. It is normally calculated automatically, based on the paper size.

`linewidth`
: The music line width. It is normally read from the `\layout` block.

`papersize`
: The paper size (as a name, e.g., `a4`). It is normally read from the `\layout` block.

`fontenc`
: The font encoding, should be set identical to the `font-encoding` property in the score.

5 Notation manual

This chapter describes all the different types of notation supported by LilyPond. It is intended as a reference for users who are already somewhat familiar with LilyPond.

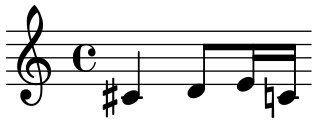
5.1 Note entry

This section is about basic notation elements like notes, rests, and related constructs, such as stems, tuplets and ties.

5.1.1 Notes

A note is printed by specifying its pitch and then its duration,

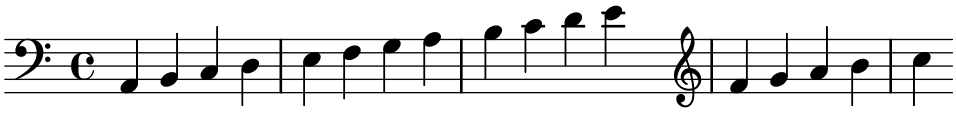
```
{ cis'4 d'8 e'16 c'16 }
```



5.1.2 Pitches

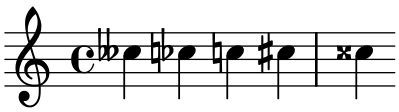
The most common syntax for pitch entry is used for standard notes and `\chordmode` modes. In these modes, pitches may be designated by names. The notes are specified by the letters `a` through `g`. The octave is formed with notes ranging from `c` to `b`. The pitch `c` is an octave below middle C and the letters span the octave above that C

```
\clef bass
a,4 b, c d e f g a b c' d' e' \clef treble f' g' a' b' c''
```



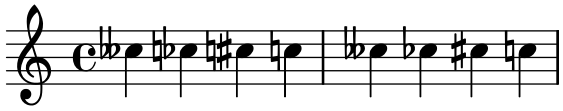
A sharp is formed by adding `-is` to the end of a pitch name and a flat is formed by adding `-es`. Double sharps and double flats are obtained by adding `-isis` or `-eses`. These names are the Dutch note names. In Dutch, `aes` is contracted to `as`, but both forms are accepted. Similarly, both `es` and `ees` are accepted

```
ceses4
ces
c
cis
cisis
```



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

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

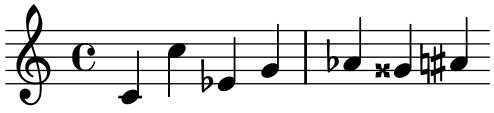


There are predefined sets of note names for various other languages. To use them, include the language specific init file. For example: `\include "english.ly"`. The available language files and the note names they define are

	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

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

```
c’ c’’ es’ g’ as’ gisis’ ais’
```



Predefined commands

Notes can be hidden and unhidden with the following commands

`\hideNotes`, `\unHideNotes`.

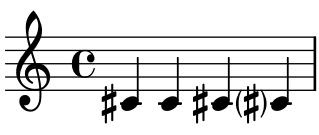
See also

Program reference: `NoteEvent`, and `NoteHead`.

5.1.3 Chromatic alterations

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.

```
cis’ cis’ cis’! cis’?
```



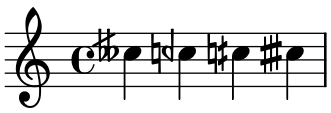
See also

The automatic production of accidentals can be tuned in many ways. For more information, refer to Section 5.6.1 [Automatic accidentals], page 89.

5.1.4 Micro tones

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

```
{ ceseh ceh cih cisih }
```



Micro tones are also exported to the MIDI file

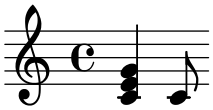
Bugs

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

5.1.5 Chords

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

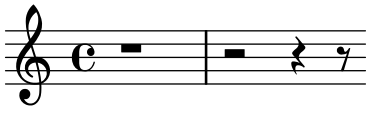
```
<c e g>4 <c>8
```



5.1.6 Rests

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

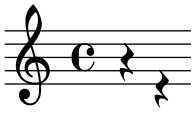
```
r1 r2 r4 r8
```



Whole bar rests, centered in middle of the bar, must be done with multi-measure rests. They are discussed in Section 5.15.8 [Multi measure rests], page 139.

A rest's vertical position may be explicitly specified by entering a note with the `\rest` keyword appended, the rest will be placed at the note's place. This makes manual formatting in polyphonic music easier. Automatic rest collision formatting will leave these rests alone

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



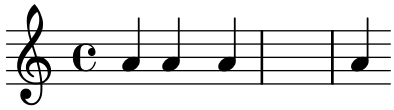
See also

Program reference: `RestEvent`, and `Rest`.

5.1.7 Skips

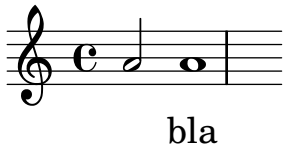
An invisible rest (also called a 'skip') can be entered like a note with note name '`s`' or with `\skip` *`duration`*

```
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 a1 }
  \new Lyrics \lyricmode { \skip 2 bla1 }
>>
```



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.

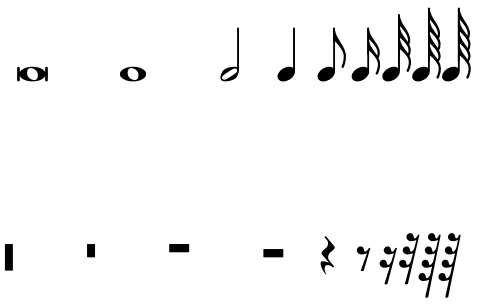
See also

Program reference: `SkipEvent`, `SkipMusic`.

5.1.8 Durations

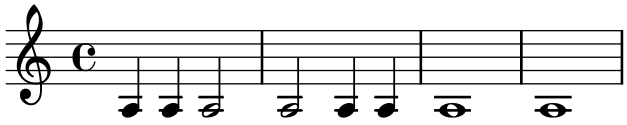
In Note, Chord, and Lyrics mode, durations are designated by numbers and dots: durations are entered as their reciprocal values. For example, a quarter note is entered using a `4` (since it is a 1/4 note), while a half note is entered using a `2` (since it is a 1/2 note). For notes longer than a whole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

```
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.

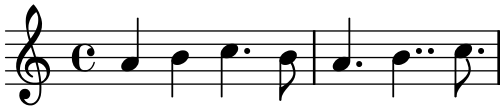
```
{ a a a2 a a4 a a1 a }
```



5.1.9 Augmentation dots

To obtain dotted note lenghts, 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`.

5.1.10 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 5.1.13 [Tuplets], page 72

5.1.11 Stems

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

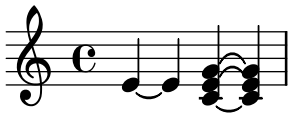
Predefined commands

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

5.1.12 Ties

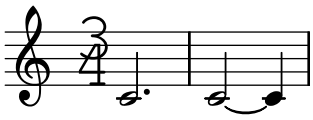
A tie connects two adjacent note heads of the same pitch. The tie in effect extends the length of a note. Ties should not be confused with slurs, which indicate articulation, or phrasing slurs, which indicate musical phrasing. A tie is entered using the tilde symbol '~'

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

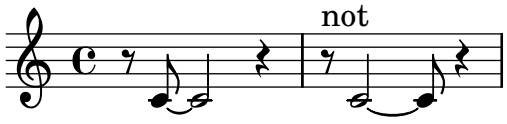


When a tie is applied to a chord, all note heads whose pitches match are connected. When no note heads match, no ties will be created.

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, eg.



If you need to tie a lot of notes over bars, it may be easier to use automatic note splitting (see Section 5.2.5 [Automatic note splitting], page 75). This mechanism automatically splits long notes, and ties them across bar lines.

Predefined commands

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

See also

In this manual: Section 5.2.5 [Automatic note splitting], page 75.

Program reference: `TieEvent`, `Tie`.

Bugs

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

Formatting of ties is a difficult subject. The results are often not optimal.

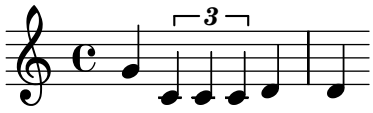
5.1.13 Tuplets

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

```
\times fraction musicexpr
```

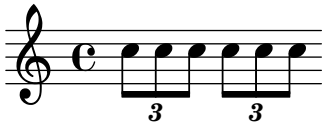
The duration of *musicexpr* will be multiplied by the fraction. The fraction's denominator will be printed over the notes, optionally with a bracket. The most common tuplet is the triplet in which 3 notes have the length of 2, so the notes are 2/3 of their written length

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



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 { c'8 c c c c c }
```



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

Predefined commands

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

See also

User manual: Section 7.1.2 [Changing context properties on the fly], page 174 for the `\set` command.

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

Examples: '`input/regression/tuplet-nest.ly`'.

Bugs

Nested tuplets are not formatted automatically. In this case, outer tuplet brackets should be moved manually, which is demonstrated in '`input/regression/tuplet-nest.ly`'.

5.2 Easier music entry

This section deals with tricks and features of the input language that were added solely to help entering music and finding and correcting mistakes. There are also external tools that make debugging easier. See Appendix D [Point and click], page 249 for more information.

It is also possible to enter and edit music using other programs, such as GUI interfaces or MIDI sequencers. Refer to the LilyPond website for more information.

5.2.1 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

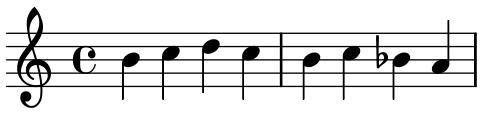
```
\relative musicexpr
```

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 diminshed fifth, even though the fourth is seven semitones while the fifth is only 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.

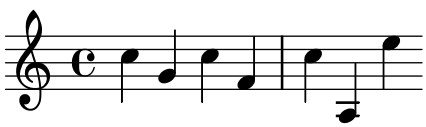
Here is the relative mode shown in action

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



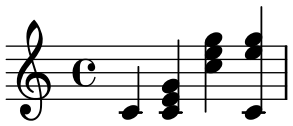
Octave changing marks are used for intervals greater than a fourth

```
\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 the `\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`.

5.2.2 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 a syntax that is separate from the notes. The syntax

```
\octave pitch
```

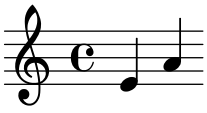
This checks that *pitch* (without quotes) yields *pitch* (with quotes) in \relative mode. If not, a warning is printed, and the octave is corrected.

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 a `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
}
```



5.2.3 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 |
```

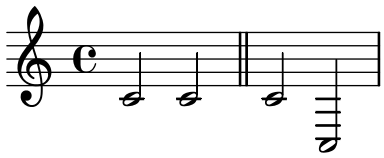
Bar checks can also be used in lyrics, for example

```
\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. To speed up this process, the `skipTypesetting` feature may be used. It is described in the next section.

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

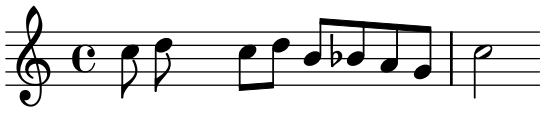
```
pipeSymbol = \bar "||"

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



5.2.4 Skipping corrected music

The property `Score.skipTypesetting` can be used to switch on and off typesetting completely during the interpretation phase. When typesetting is switched off, the music is processed much more quickly. This can be used to skip over the parts of a score that have already been checked for errors

```
\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 }
```

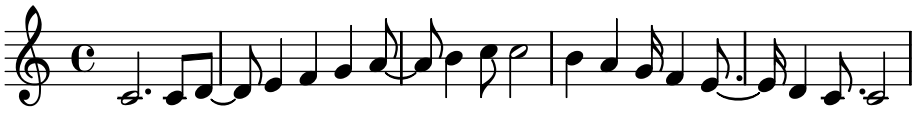


In polyphonic music, `Score.skipTypesetting` will affect all voices and staves, saving even more time.

5.2.5 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.

Bugs

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

See also

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

Program reference: `Completion_heads_engraver`.

5.3 Staff notation

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

5.3.1 Staff symbol

Notes, dynamic signs, etc., are grouped with a set of horizontal lines, into a staff (plural ‘staves’). In our system, these lines are drawn using a separate layout object called staff symbol.

See also

Program reference: `StaffSymbol`.

Examples: ‘`input/test/staff-lines.ly`’, ‘`input/test/staff-size.ly`’.

Bugs

If a staff is ended halfway a piece, the staff symbol may not end exactly on the bar line.

5.3.2 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

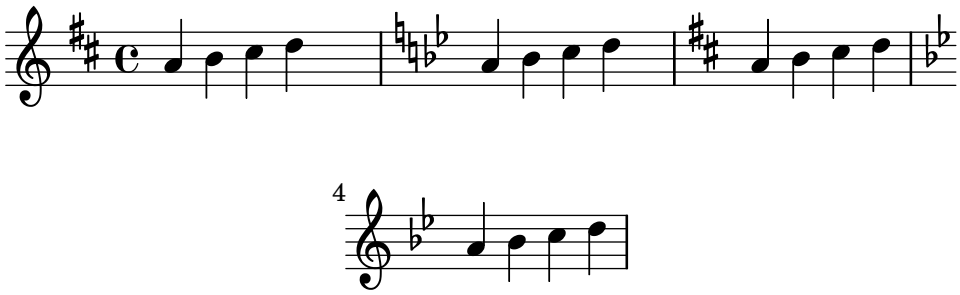
`\key` *`pitch type`*

Here, *type* should be `\major` or `\minor` to get *pitch*-major or *pitch*-minor, respectively. The standard mode names `\ionian`, `\locrian`, `\aeolian`, `\mixolydian`, `\lydian`, `\phrygian`, and `\dorian` are also defined.

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

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
}
```



Accidentals and key signatures often confuse new users, because unaltered notes get natural signs depending on the key signature. For more information, see Section 2.3 [More about pitches], page 12.

See also

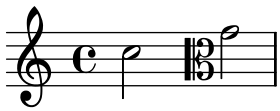
Program reference: `KeyChangeEvent`, `KeyCancellation` and `KeySignature`.

5.3.3 Clef

The clef indicates which lines of the staff correspond to which pitches.

The clef can be set with the `\clef` command

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



Supported clef-names include

`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

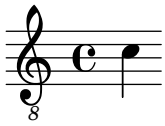
`subbass` F clef on 5th line

`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
```



This command 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.

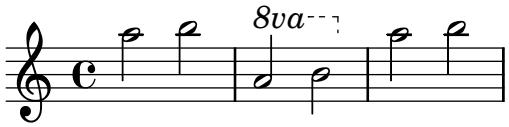
See also

Program reference: `Clef`.

5.3.4 Ottava brackets

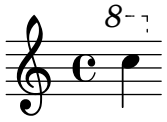
'Ottava' brackets introduce an extra transposition of an octave for the staff. They are created by invoking the function `set-octavation`

```
\relative c''' {
  a2 b
  #(set-octavation 1)
  a b
  #(set-octavation 0)
  a b
}
```



The `set-octavation` function also takes -1 (for 8va bassa) and 2 (for 15ma) as arguments. Internally the function sets the properties `ottavation` (e.g., to `"8va"`) 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

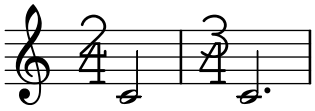
`set-octavation` will get confused when clef changes happen during an octavation bracket.

5.3.5 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.

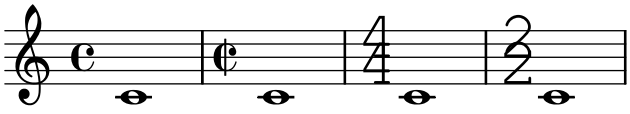
The time signature is set or changed by the `\time` command

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



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
```



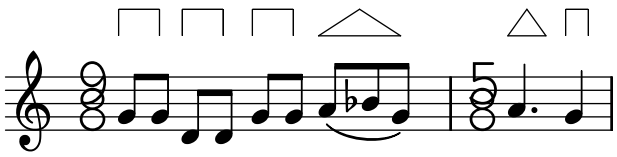
There are many more options for its layout. See Section 5.16.6 [Ancient time signatures], page 151 for more examples.

This command 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_engraver`.

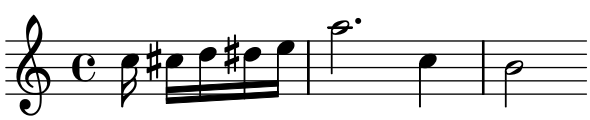
Bugs

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

5.3.6 Partial measures

Partial measures, for example in upsteps, are entered using the `\partial` command

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



The syntax for this command is

`\partial` *`duration`*

This is internally translated into

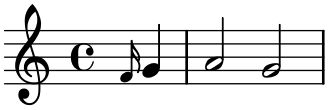
`\set Timing.measurePosition = -`*`length of duration`*

The property `measurePosition` contains a rational number indicating how much of the measure has passed at this point.

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
}
```



5.3.7 Unmetered music

Bar lines and bar numbers are calculated automatically. For unmetered music (cadenzas, for example), this is not desirable. By setting `Score.timing` to false, this automatic timing can be switched off. Empty bar lines,

```
\bar ""
```

indicate where line breaks can occur.

Predefined commands

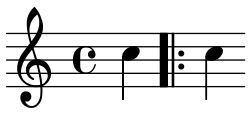
`\cadenzaOn`, `\cadenzaOff`.

5.3.8 Bar lines

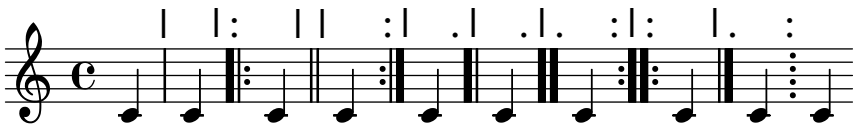
Bar lines delimit measures, but are also used to indicate repeats. Normally, they are inserted automatically. Line breaks may only happen on bar lines.

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

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



The following bar types are available



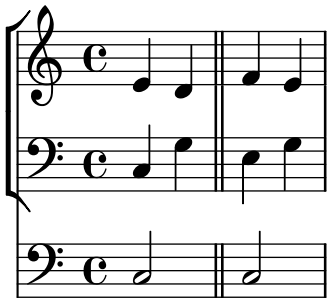
For allowing line breaks, there is a special command,

```
\bar ""
```

This will insert an invisible bar line, and allow line breaks at this point.

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`

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



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 5.8 [Repeats], page 103.

See also

In this manual: Section 5.8 [Repeats], page 103, Section 5.15.1 [System start delimiters], page 133.

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

Examples: '`input/test/bar-lines.ly`',

5.3.9 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.

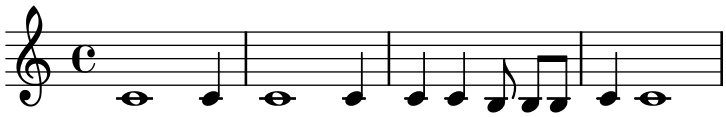
`measureLength`
: The length of the measures in the current time signature. For a 4/4 time this is 1, and for 6/8 it is 3/4.

`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.

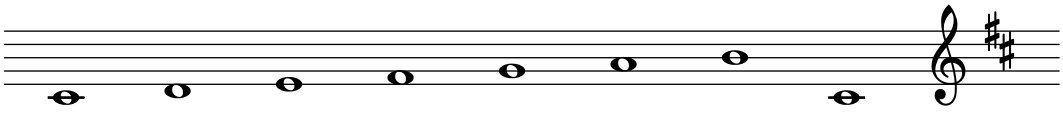
```
\set Score.measureLength = #(ly:make-moment 5 4)
c1 c4
c1 c4
c4 c4
\set Score.measurePosition = #(ly:make-moment 7 8)
b8 b b
c4 c1
```

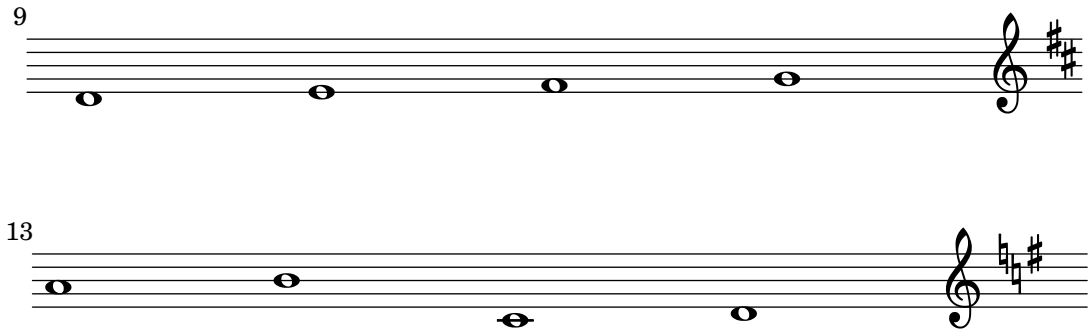


5.3.10 Controlling formatting of prefatory matter

This example demonstrates how to place prefatory matter (such as the clef and key signature) at the end of a line.

```
\transpose c c' {
  \override Staff.Clef
    #'break-visibility = #end-of-line-visible
  \override Staff.KeySignature
    #'break-visibility = #end-of-line-visible
  \set Staff.explicitClefVisibility = #end-of-line-visible
  \set Staff.explicitKeySignatureVisibility = #end-of-line-visible

  % We want the time sig to take space, otherwise there is not
  % enough white at the start of the line.

  \override Staff.TimeSignature #'transparent = ##t
  \set Score.defaultBarType = #"empty"

  c1 d e f g a b c
  \key d \major
  \break

  % see above.
  \time 4/4

  d e fis g a b cis d
  \key g \major
  \break
  \time 4/4
}
```





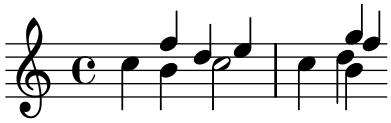
5.4 Polyphony

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

5.4.1 Writing polyphonic music

The easiest way to enter fragments with more than one voice on a staff is to split chords using the separator \\. You can use it for small, short-lived voices or for single chords

```
\context Staff \relative c'' {
  c4 << { f d e } \\ { b c2 } >>
  c4 << g' \\ b, \\ f' \\ d >>
}
```



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. Specifically,

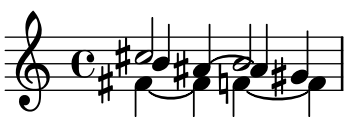
```
<< \upper \\ \lower >>
```

is equivalent to

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

This can also be done by instantiating `Voice` contexts by hand, and using `\voiceOne`, up to `\voiceFour` to assign a stem directions and horizontal shift for each part

```
\relative c''
\context Staff <<
  \new Voice { \voiceOne cis2 b }
  \new Voice { \voiceThree b4 ais ~ ais4 gis4 }
  \new Voice { \voiceTwo fis4~ fis4 f ~ f } >>
```



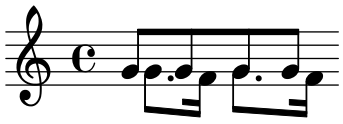
[1] Polyphonic voices are sometimes called “layers” in other notation packages

The command `\oneVoice` will revert back to the normal setting.

Defining voices (either with the separator `\\` or with `\voiceXXX`) will set the direction of stems, slurs, ties, articulations, text annotations, augmentation dots of dotted notes, and fingerings. `\voiceOne` makes these objects point upwards, while `\voiceTwo` makes them point down.

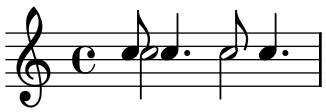
Normally, note heads with a different number of dots are not merged, but when the object property `merge-differently-dotted` is set in the `NoteCollision` object, they are merged

```
\context 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`

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



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

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

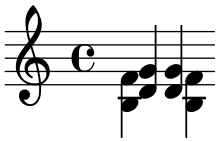
Predefined commands

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

`\shiftOn`, `\shiftOnn`, `\shiftOnnn`, `\shiftOff`: these commands specify in what 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 5.17.2 [Clusters], page 165).

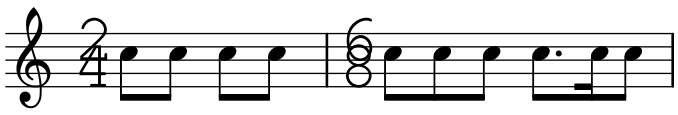
5.5 Beaming

Beams are used to group short notes into chunks that are aligned with the metrum. LilyPond normally inserts beams automatically, but if you wish you may control them manually or changed how beams are automatically grouped.

5.5.1 Automatic beams

LilyPond inserts beams automatically

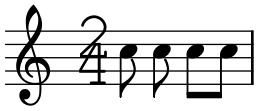
```
\time 2/4 c8 c c c \time 6/8 c c c c8. c16 c8
```



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.

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

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



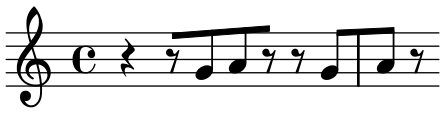
See also

Program reference: `Beam`.

5.5.2 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
}
```

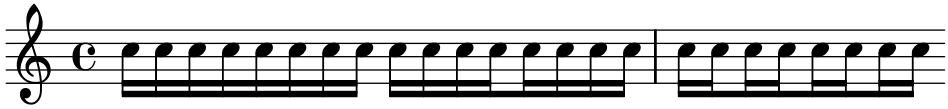


Normally, beaming patterns within a beam are determined automatically. If necessary, the properties `stemLeftBeamCount` and `stemRightBeamCount` can be used to override the defaults. If either property is set, its value will be used only once, and then it is erased

```
{
   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]
```



Normally, line breaks are forbidden when beams cross bar lines. This behavior can be changed by setting `allowBeamBreak`.

See also

User manual: Section 7.1.2 [Changing context properties on the fly], page 174 for the `\set` command

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 5.15.10 [Hiding staves], page 142.

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

5.5.3 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 defaults for `autoBeamSettings` are defined in '`scm/auto-beam.scm`'.

The value of `autoBeamSettings` is changed with three functions,

```
#(override-auto-beam-setting
   '(be p q n m) a b
   [context])
#(score-override-auto-beam-setting
   '(be p q n m) a b)
#(revert-auto-beam-setting '(be p q n m)
   [context])
```

Here, *be* is the symbol `begin` or `end`, and *context* is an optional context (default: `'Voice`). It determines whether the rule applies to begin or end-points. The quantity p/q refers to the length of the beamed notes (and '`* *`' designates notes of any length), n/M refers to a time signature (wildcards '`* *`' may be entered to designate all time signatures), a/b is a duration. By default, this command changes settings for the current voice. It is also possible to adjust settings at higher contexts, by adding a *context* argument. `score-override-auto-beam-setting` is equal to `override-auto-beam-setting` with the argument *context* set to `'Score`.

For example, if automatic beams should end on every quarter note, use the following

```
#(override-auto-beam-setting '(end * * * *) 1 4 'Staff)
```

Since the duration of a quarter note is 1/4 of a whole note, it is entered as `(ly:make-moment 1 4)`.

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)
```

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).

Rules can also be restricted to specific time signatures. A rule that should only be applied in N/M time signature is formed by replacing the second asterisks by N and M. For example, a rule for 6/8 time exclusively looks like

```
#(override-auto-beam-setting '(begin * * 6 8) ...)
```

If a rule should be to applied only to certain types of beams, use the first pair of asterisks. Beams are classified according to the shortest note they contain. For a beam ending rule that only applies to beams with 32nd notes (and no shorter notes), use `(end 1 32 * *)`.

If beams are used to indicate melismata in songs, then automatic beaming should be switched off. This is done by setting `autoBeaming` to `#f`.

Predefined commands

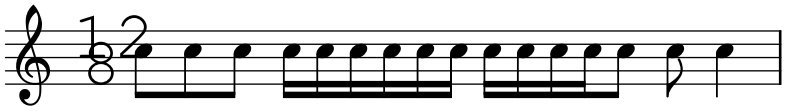
`\autoBeamOff`, `\autoBeamOn`.

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.

The rules for ending a beam depend on the shortest note in a beam. So, while it is possible to have different ending rules for eight beams and sixteenth beams, a beam that contains both eight and sixteenth notes will use the rules for the sixteenth beam.

In the example below, the autobeamer makes eighth beams and sixteenth end at three eighths. The third beam can only be corrected by specifying manual beaming.

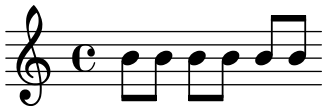


It is not possible to specify beaming parameters that act differently in different parts of a measure. This means that it is not possible to use automatic beaming in irregular meters such as `5/8`.

5.5.4 Beam formatting

When a beam falls in the middle of the staff, the beams point normally down. However, this behaviour can be altered with the `neutral-direction` property.

```
{
  b8[ b]
  \override Beam #'neutral-direction = #-1
  b[ b]
  \override Beam #'neutral-direction = #1
  b[ b]
}
```



5.6 Accidentals

This section describes how to change the way that accidentals are inserted automatically before notes.

5.6.1 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.

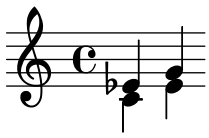
The following accidental styles are supported

`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

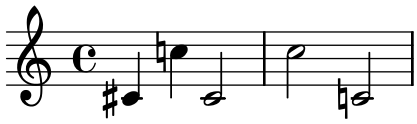
```
\context Staff <<
  #(set-accidental-style 'voice)
  <<
    { es g } \\
    { c, e }
>> >>
```



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

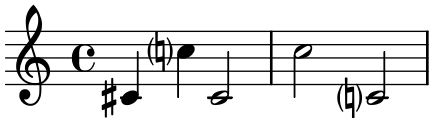
```
#(set-accidental-style 'modern)
cis' c'' cis'2 | c'' c'
```



`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`

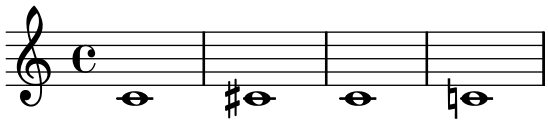
This rule is the same as `modern-voice`, but with the extra accidentals (the ones not typeset by `voice`) typeset as cautionaries. Even though all accidentals typeset by `default` *are* typeset by this variable, some of them are typeset as cautionaries.

`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.

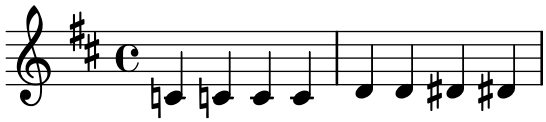
`no-reset` This is the same as `default` but with accidentals lasting "forever" and not only until the next measure

```
#(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 happened once 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.

5.7 Expressive marks

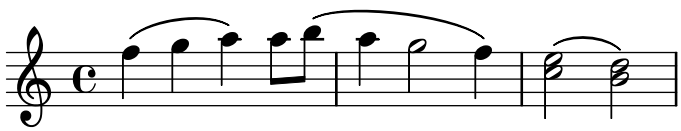
Expressive marks help musicians to bring more to the music than simple notes and rhythms.

5.7.1 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)
```

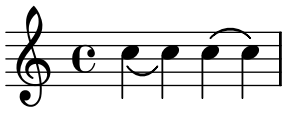


The direction of a slur can be set with the generic commands

```
\override Slur #'direction = #UP
\slurUp            % shortcut for the previous line
```

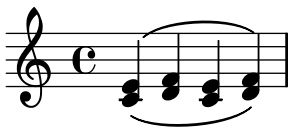
However, there is a convenient shorthand for forcing slur directions. By adding _ or ^ before the opening parentheses, the direction is also set. For example,

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



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`, `\slurDotted`, `\slurSolid`.

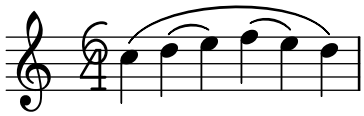
See also

Program reference: internals document, `Slur`, and `SlurEvent`.

5.7.2 Phrasing slurs

A phrasing slur (or phrasing mark) connects chords 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`.

The commands `\slurUp`, `\slurDown`, and `\slurNeutral` will only affect normal slurs and not phrasing slurs.

Predefined commands

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

See also

Program reference: see also `PhrasingSlur`, and `PhrasingSlurEvent`.

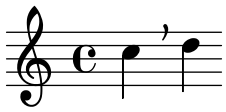
Bugs

Putting phrasing slurs over rests leads to spurious warnings.

5.7.3 Breath marks

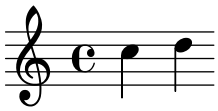
Breath marks are entered using `\breathe`

```
c'4 \breathe d4
```



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`, `BreathingSignEvent`.

Examples: '`input/regression/breathing-sign.ly`'.

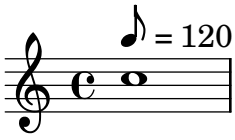
5.7.4 Metronome marks

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
```



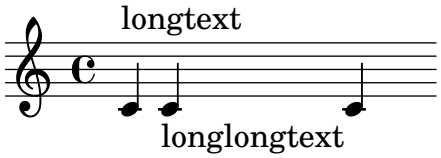
See also

Program reference: `MetronomeChangeEvent`.

5.7.5 Text scripts

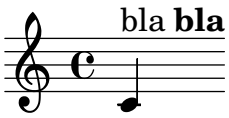
It is possible to place arbitrary strings of text or markup text (see Section 7.4 [Text markup], page 188) above or below notes by using a string `c^"text"`. By default, these indications do not influence the note spacing, but by using the command `\fatText`, the widths will be taken into account

```
c4^"longtext" \fatText c4_"longlongtext" c4
```



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 7.4 [Text markup], page 188.

Predefined commands

`\fatText`, `\emptyText`.

See also

In this manual: Section 7.4 [Text markup], page 188.

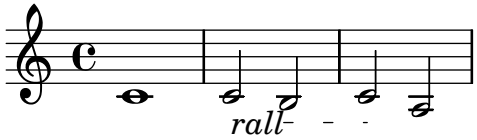
Program reference: `TextScriptEvent`, `TextScript`.

5.7.6 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
\override TextSpanner #'direction = #-1
\override TextSpanner #'edge-text = #'("rall " . "")
c2\startTextSpan b c\stopTextSpan a
```



See also

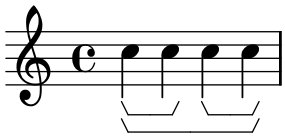
Internals `TextSpanEvent`, `TextSpanner`.

Examples: '`input/regression/text-spanner.ly`'.

5.7.7 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"
}}}
```



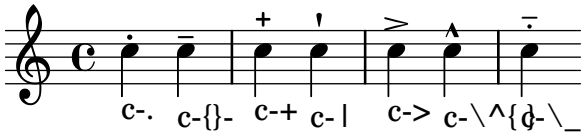
See also

Program reference: `HorizontalBracket`, `NoteGroupingEvent`.

Examples: '`input/regression/note-group-bracket.ly`'.

5.7.8 Articulations

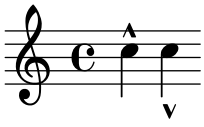
A variety of symbols can appear above and below notes to indicate different characteristics of the performance. They are added to a note by adding a dash and the character signifying the articulation. They are demonstrated here



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

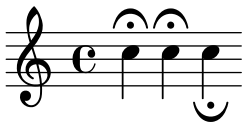
The script is automatically placed, but the direction can be forced as well. Like other pieces of LilyPond code, `_` will place them below the staff, and `^` will place them above.

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

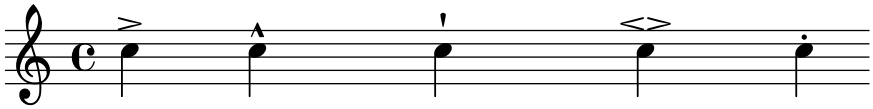


Other symbols can be added using the syntax *note*`\`*name*. Again, they can be forced up or down using `^` and `_`, e.g.,

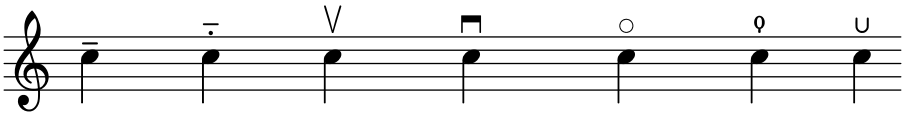
```
c\fermata c^\fermata c_\fermata
```



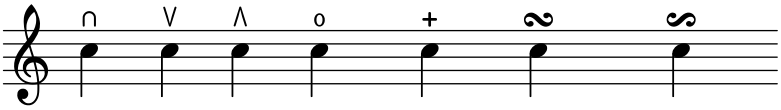
Here is a chart showing all scripts available,



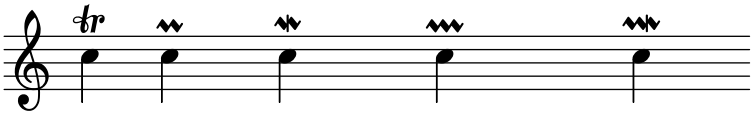
accent marcato staccatissimo espressivo staccato



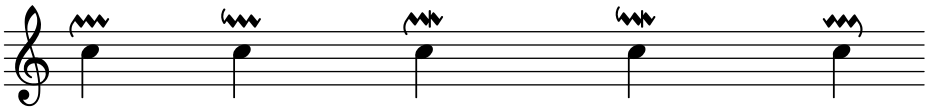
tenuto portato upbow downbow flageolet thumb lheel



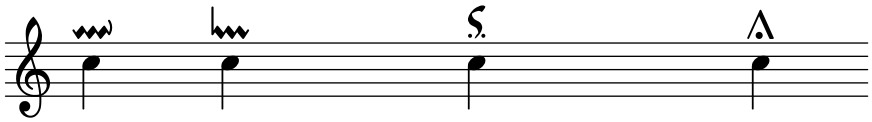
rheel ltoe rtoe open stopped turn reverseturn



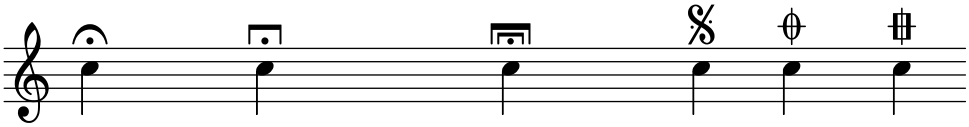
trill prall mordent prallprall prallmordent



upprall downprall upmordent downmordent pralldown



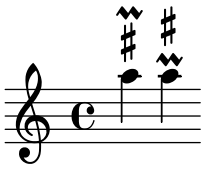
prallup lineprall signumcongruentiae shortfermata



fermata longfermata verylongfermata segno coda varcoda

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: `ScriptEvent`, and `Script`.

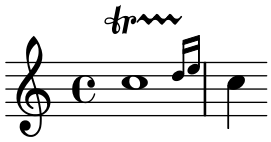
Bugs

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

5.7.9 Running trills

Long running trills are made with `\startTrillSpan` and `\stopTrillSpan`,

```
\new Voice {
  << { c1 \startTrillSpan }
     { s2. \grace { d16[\stopTrillSpan e] } } >>
  c4 }
```



Predefined commands

`\startTrillSpan`, `\stopTrillSpan`.

See also

Program reference: `TrillSpanner`, `TrillSpanEvent`.

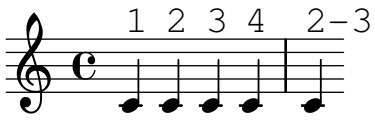
5.7.10 Fingering instructions

Fingering instructions can be entered using

`note-digit`

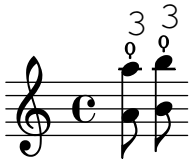
For finger changes, use markup texts

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



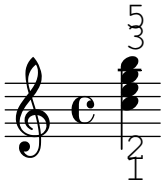
You can use the thumb-script to indicate that a note should be played with the thumb (e.g., in cello music)

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



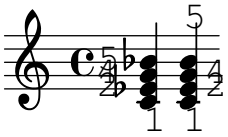
Fingerings for chords can also be added to individual notes of the chord by adding them after the pitches

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



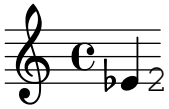
In this case, setting `fingeringOrientations` will put fingerings next to note heads

```
\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

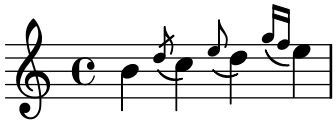
Program reference: `FingerEvent`, and `Fingering`.

Examples: '`input/regression/finger-chords.ly`'.

5.7.11 Grace notes

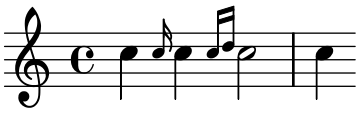
Grace notes are ornaments that are written out. The most common ones are acciaccatura, which should be played as very short. It is denoted by a slurred small note with a slashed stem. The appoggiatura is a grace note that takes a fixed fraction of the main note, and is denoted as a slurred note in small 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
```



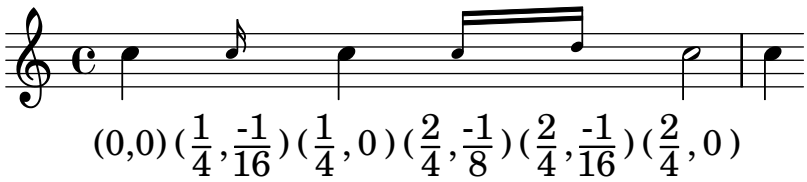
Both are special forms of the `\grace` command. By prefixing this keyword to a music expression, a new one is formed, which will be printed in a smaller font and takes up no logical time in a measure.

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



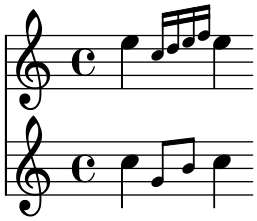
Unlike `\acciaccatura` and `\appoggiatura`, the `\grace` command does not start a slur.

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



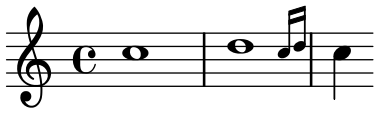
The placement of grace notes is synchronized between different staves. In the following example, there are two sixteenth grace notes for every eighth grace note

```
<< \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
```



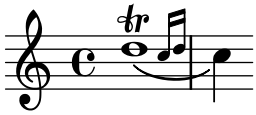
This will put the grace notes after a "space" lasting 3/4 of the length of the main note. The fraction 3/4 can be changed by setting `afterGraceFraction`, ie.

```
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

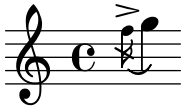
```
\context Voice {
  << { d1^\trill_( }
     { s2 \grace { c16[ d] } } >>
  c4)
}
```



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 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`'.

See also

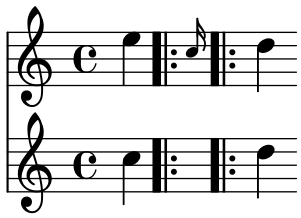
Program reference: `GraceMusic`.

Bugs

A score that starts with a `\grace` section needs an explicit `\context 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, for the above example

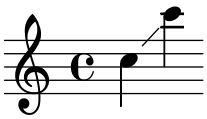
```
\new Staff { c4 \bar "|:" \grace s16 d4 }
```

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.

5.7.12 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

```
c\glissando c'
```



See also

Program reference: `Glissando`, and `GlissandoEvent`.

Example files: '`input/regression/glissando.ly`'.

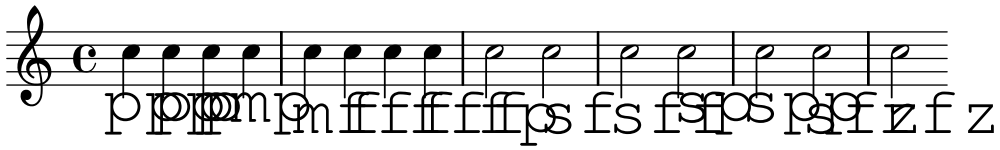
Bugs

Printing text over the line (such as *gliss.*) is not supported.

5.7.13 Dynamics

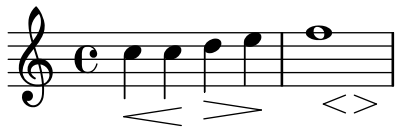
Absolute dynamic marks are specified using a command after a note `c4\ff`. The available dynamic marks are `\ppp`, `\pp`, `\p`, `\mp`, `\mf`, `\f`, `\ff`, `\fff`, `\ffff`, `\fp`, `\sf`, `\sff`, `\sp`, `\spp`, `\sfz`, and `\rfz`

```
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 `\!`. A decrescendo is started with `\>` and also terminated with `\!`. 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\! } >>
```

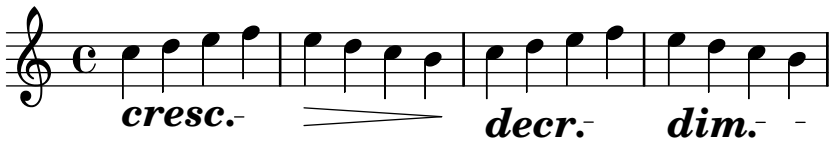


This may give rise to very short hairpins. Use `minimum-length` in `Voice.Hairpin` to lengthen them, for example

```
\override Staff.Hairpin #'minimum-length = #5
```

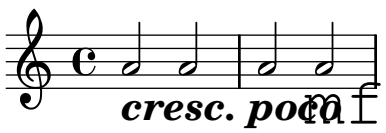
You can also use a text saying *cresc.* instead of hairpins. Here is an example how to do it

```
\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 7.4.4 [New dynamic marks], page 196.

Predefined commands

`\dynamicUp`, `\dynamicDown`, `\dynamicNeutral`.

See also

Program reference: `CrescendoEvent`, `DecrescendoEvent`, and `AbsoluteDynamicEvent`.

Dynamics are `DynamicText` and `Hairpin` objects. Vertical positioning of these symbols is handled by the `DynamicLineSpanner` object.

5.8 Repeats

Repetition is a central concept in music, and multiple notations exist for repetitions.

5.8.1 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.

5.8.2 Repeat syntax

LilyPond has one syntactic construct for specifying different types of repeats. The syntax is

```
\repeat variant repeatcount repeatbody
```

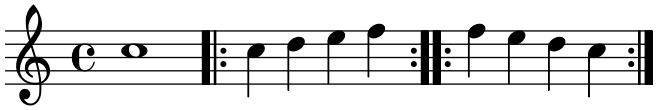
If you have alternative endings, you may add

```
\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.

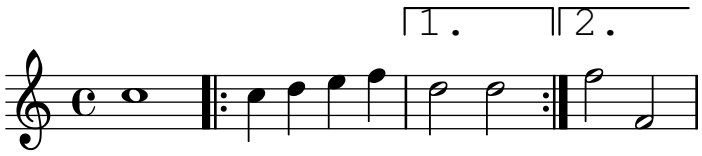
Standard repeats are used like this

```
c1
\repeat volta 2 { c4 d e f }
\repeat volta 2 { f e d c }
```

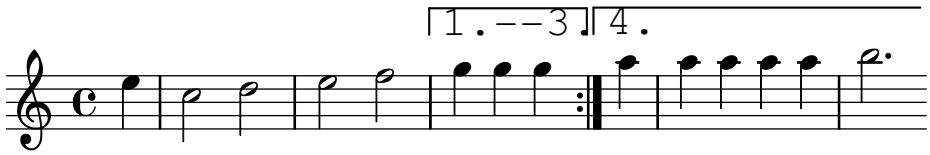


With alternative endings

```
c1
\repeat volta 2 {c4 d e f}
\alternative { {d2 d} {f f,} }
```

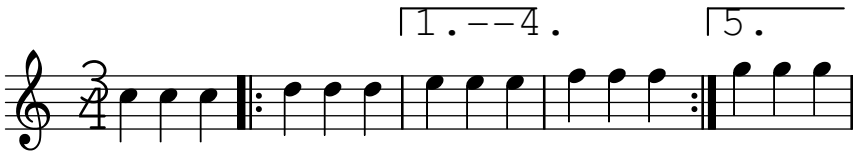



```
\context Staff {
  \partial 4
  \repeat volta 4 { e | c2 d2 | e2 f2 | }
  \alternative { { g4 g g } { a | a a a a | b2. } }
}
```



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
```

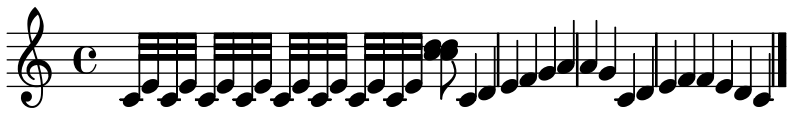
is ambiguous, since it is is not clear to which `\repeat` the `\alternative` belongs. This ambiguity is resolved by always having the `\alternative` belong to the inner `\repeat`. For clarity, it is advisable to use braces in such situations.

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.

5.8.3 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 functions 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, then 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 { .. }
}
```

5.8.4 Manual repeat commands

The property `repeatCommands` can be used to control the layout of repeats. Its value is a Scheme list of repeat commands.

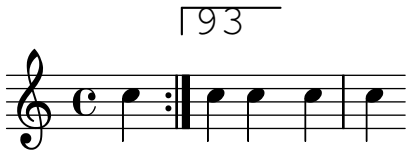
`start-repeat`
: Print a `|:` bar line.

`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 7.4 [Text markup], page 188. 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
```



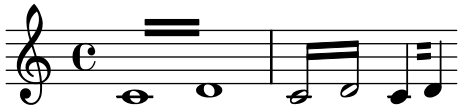
See also

Program reference: `VoltaBracket`, `RepeatedMusic`, `VoltaRepeatedMusic`, `UnfoldedRepeatedMusic`, and `FoldedRepeatedMusic`.

5.8.5 Tremolo repeats

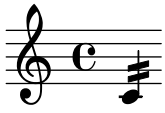
To place tremolo marks between notes, use `\repeat` with tremolo style

```
\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 5.8.6 [Tremolo subdivisions], page 107.

See also

In this manual: Section 5.8.6 [Tremolo subdivisions], page 107, Section 5.8 [Repeats], page 103.

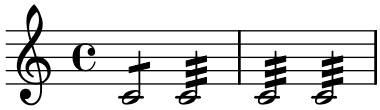
Program reference: tremolo beams are `Beam` objects. Single stem tremolos are `StemTremolo` objects. The music expression is `TremoloEvent`.

Example files: '`input/regression/chord-tremolo.ly`', '`input/regression/stem-tremolo.ly`'.

5.8.6 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

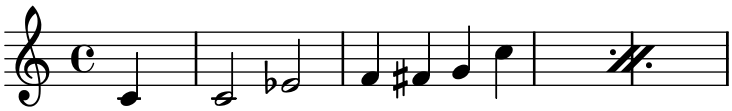
In this manual: Section 5.8.5 [Tremolo repeats], page 106.

Elsewhere: `StemTremolo`, `TremoloEvent`.

5.8.7 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

```
\new Voice \relative c' {
  \repeat "percent" 4 { c4 }
  \repeat "percent" 2 { c2 es2 f4 fis4 g4 c4 }
}
```



See also

Program reference: `RepeatSlash`, `PercentRepeat`, `PercentRepeatedMusic`, and `DoublePercentRepeat`.

5.9 Rhythmic music

Rhythmic music is primarily used for percussion and drum notation, but it can also be used to show the rhythms of melodies.

5.9.1 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

```
\context RhythmicStaff {
  \time 4/4
  c4 e8 f g2 | r4 g r2 | g1:32 | r1 |
```

```
}
```

See also

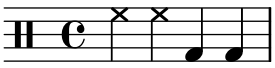
Program reference: `RhythmicStaff`.

Examples: ‘`input/regression/rhythmic-staff.ly`’.

5.9.2 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

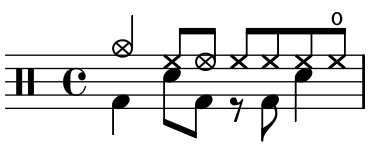
Program reference: `DrumNoteEvent`.

5.9.3 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.

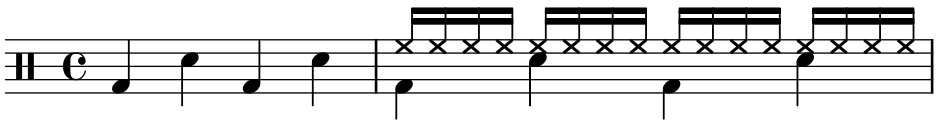
To typeset the music, the notes must be interpreted in a `DrumStaff` and `DrumVoice` contexts

```
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 }
  >>
```



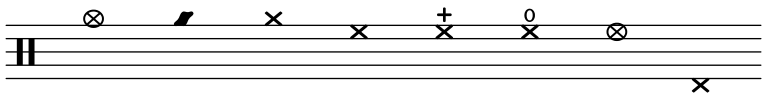
The above example shows verbose polyphonic notation. The short polyphonic notation, described in Section 5.4 [Polyphony], page 84, can also be used if the `DrumVoices` are instantiated by hand first. For example,

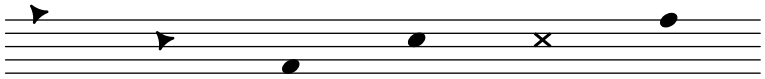
```
\new DrumStaff <<
  \context DrumVoice = "1" { s1 *2 }
  \context DrumVoice = "2" { s1 *2 }
  \drummode {
    bd4 sn4 bd4 sn4
    <<
      { \repeat unfold 16 hh16 }
      \\
      { bd4 sn4 bd4 sn4 }
    >>
  }
>>
```

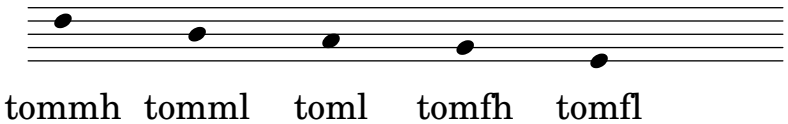


There are also other layout possibilities. To use these, set the property `drumStyleTable` in context `DrumVoice`. The following variables have been predefined

`drums-style`
: This is the default. It typesets a typical drum kit on a five-line staff

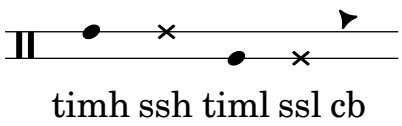




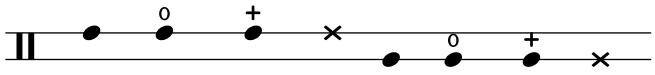


The drum scheme supports six different toms. When there are fewer toms, simply select the toms that produce the desired result, i.e., to get toms on the three middle lines you use `tommh`, `tomml`, and `tomfh`.

`timbales-style`
: This typesets timbales on a two line staff

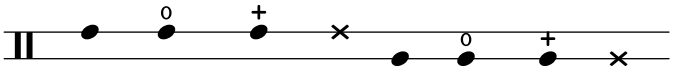


`congas-style`
: This typesets congas on a two line staff



cgh cgho cghm ssh cgl cglo cglm ssl

`bongos-style`
: This typesets bongos on a two line staff



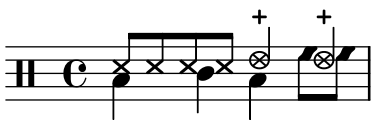
boh boho bohm ssh bol bolo bolm ssl

`percussion-style`
: To typeset all kinds of simple percussion on one line staves.

tri trio trim gui guis guil cb cl tamb cab mar hc

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.

5.10 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 3.3 [Piano templates], page 32.

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`'.

5.10.1 Automatic staff changes

Voices can be made to switch automatically between the top and the bottom staff. The syntax for this is

```
\autochange ...music...
```

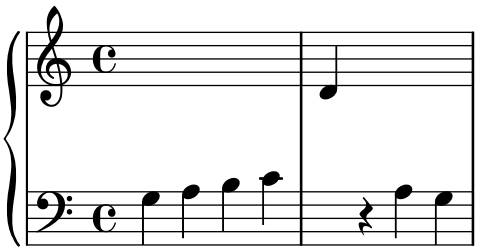
This will create two staves inside the current PianoStaff, called `up` and `down`. The lower staff will be in bass clef by default.

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

```
\context PianoStaff
  \autochange \relative c'
  {
    g4 a b c d r4 a g
  }
```



See also

In this manual: Section 5.10.2 [Manual staff switches], page 112.

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`.

Internally, the `\partcombine` interprets both arguments as `Voice`s 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.

5.10.2 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,

```
<<
  \context Staff = up {
    \skip 1 * 10  % keep staff alive
    }
  \context Staff = down {
    \skip 1 * 10  % idem
    }
>>
```

and the `Voice` is inserted afterwards

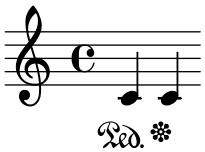
```
\context Staff = down
  \new Voice { ... \change Staff = up ... }
```

5.10.3 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

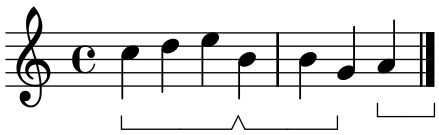
```
c'4\sustainDown c'4\sustainUp
```



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.

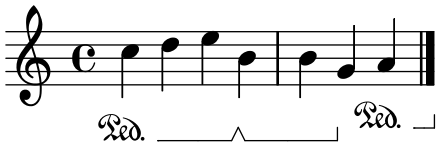
Pedals can also be indicated by a sequence of brackets, by setting the `pedalSustainStyle` property to bracket objects

```
\set Staff.pedalSustainStyle = #'bracket
c\sustainDown d e
b\sustainUp\sustainDown
b g \sustainUp a \sustainDown \bar "|."
```



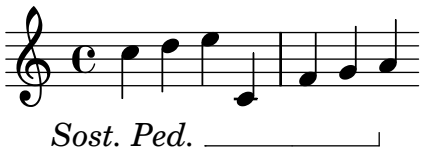
A third style of pedal notation is a mixture of text and brackets, obtained by setting the `pedalSustainStyle` property to `mixed`

```
\set Staff.pedalSustainStyle = #'mixed
c\sustainDown d e
b\sustainUp\sustainDown
b g \sustainUp a \sustainDown \bar "|."
```



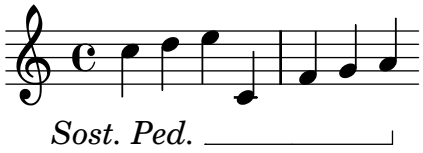
The default '*Ped.' style for sustain and damper pedals corresponds to style `#'text`. The sostenuto pedal uses `mixed` style by default.

```
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

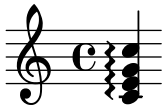
```
\override Staff.PianoPedalBracket #'shorten-pair = #'(0 . -1.0)
c\sostenutoDown d e c, f g a\sostenutoUp
```



5.10.4 Arpeggio

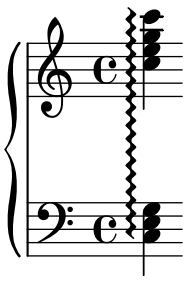
You can specify an arpeggio sign on a chord by attaching an `\arpeggio` to a chord

```
<c e g c>\arpeggio
```



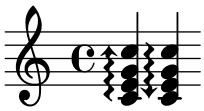
When an arpeggio crosses staves, you attach an arpeggio to the chords in both staves, and set `PianoStaff.connectArpeggios`

```
\context PianoStaff <<
  \set PianoStaff.connectArpeggios = ##t
  \new Staff { <c' e g c>\arpeggio }
  \new Staff { \clef bass <c,, e g>\arpeggio }
>>
```



The direction of the arpeggio is sometimes denoted by adding an arrowhead to the wiggly line

```
\context Voice {
  \arpeggioUp
  <c e g c>\arpeggio
  \arpeggioDown
  <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
```

Predefined commands

`\arpeggio`, `\arpeggioUp`, `\arpeggioDown`, `\arpeggioNeutral`, `\arpeggioBracket`.

See also

Program reference: `ArpeggioEvent`, `Arpeggio`.

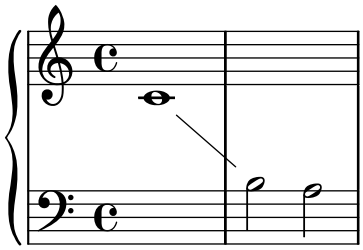
Bugs

It is not possible to mix connected arpeggios and unconnected arpeggios in one `PianoStaff` at the same point in time.

5.10.5 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 `PianoStaff.followVoice` to true

```
\context PianoStaff <<
  \set PianoStaff.followVoice = ##t
  \context Staff \context Voice {
    c1
    \change Staff=two
    b2 a
  }
 \context Staff=two { \clef bass \skip 1*2 }
>>
```



See also

Program reference: `VoiceFollower`.

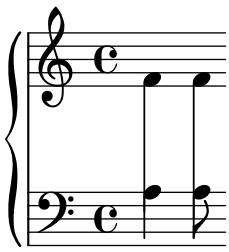
Predefined commands

`\showStaffSwitch`, `\hideStaffSwitch`.

5.10.6 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
\context PianoStaff <<
  \new Staff {
    \stemDown \stemExtend
    f'4
    \stemExtend \noFlag
    f'8
  }
  \new Staff {
    \clef bass
    a4 a8
  }
>>
```



5.11 Vocal music

There are three different issues when printing 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 printed as text, not as notes.
- Song texts must be aligned with the notes of their melody.

The simplest solution for printing music uses the `\addlyrics` function to solve all these problems at once. However, these three functions can be controlled separately, which is necessary for complex vocal music.

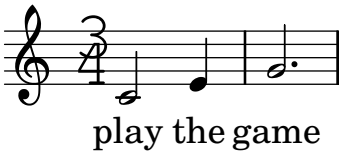
5.11.1 Setting simple songs

The easiest way to add lyrics to a melody is to append

```
\addlyrics { the lyrics }
```

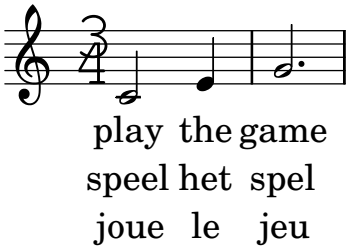
to a melody. Here is an example,

```
\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 `\addlyrics` command is actually just a convienient way to write a more complicated LilyPond structure that sets up the lyrics. You should use `\addlyrics` unless you need to do fancy things, in which case you should investigate `\lyricsto` or `\lyricmode`.

```
{ MUSIC }
\addlyrics { LYRICS }
```

is the same as

```
\context Voice = blah { music }
\lyricsto "blah" \new lyrics { LYRICS }
```

Bugs

`\addlyrics` cannot handle polyphony.

5.11.2 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 }
```

A word lyrics mode 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
```

Any `_` character that appears in an unquoted word is converted to a space. This provides a mechanism for introducing spaces into words without using quotes. Quoted words can also be used in Lyrics mode to specify words that cannot be written with the above rules. The following example incorporates double quotes

```
\lyricmode { He said: "\"Let" my peo ple "go\"" }
```

This example is slightly academic, since it gives better looking results using single quotes, ‘‘ and ’’

```
\lyricmode { He said: ‘‘Let my peo ple go’’ }
```

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 `^`.

See also

Program reference: events `LyricEvent`, and `LyricText`.

Bugs

The definition of lyrics mode is too complex.

5.11.3 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 `__`.

See also

Program reference: `HyphenEvent`, `ExtenderEvent`, `LyricHyphen`, and `LyricExtender`

5.11.4 The Lyrics context

Lyrics are printed by interpreting them in a `Lyrics` context

```
\context 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

```
\lyricsto name \new Lyrics ...
```

This aligns the lyrics to the notes of the `Voice` context called *name*, which has to 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.

For different or more complex orderings, the best way is to setup the hierarchy of staves and lyrics first, e.g.,

```
\context ChoirStaff <<
  \context Lyrics = sopranoLyrics { s1 }
  \context Voice = soprano { music }
  \context Lyrics = tenorLyrics { s1 }
  \context Voice = tenor { music }
>>
```

and then combine the appropriate melodies and lyric lines

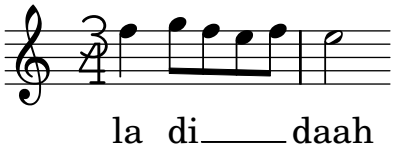
```
\lyricsto "soprano" \context Lyrics = sopranoLyrics
  the lyrics
```

The final input would resemble

```
<<\context ChoirStaff << setup the music >>
  \lyricsto "soprano" etc
  \lyricsto "alto" etc
  etc
>>
```

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.,

```
<<
  \context Voice = "lala" {
    \time 3/4
    f4 g8
    \melisma
    f e f
    \melismaEnd
    e2
  }
  \lyricsto "lala" \new Lyrics {
    la di __ daah
  }
>>
```



In addition, notes are considered a melisma if they are manually beamed, and automatic beaming (see Section 5.5.3 [Setting automatic beam behavior], page 88) is switched off.

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.
```

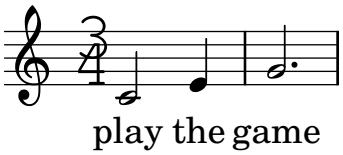
The alignment to a melody can be specified with the `associatedVoice` property,

```
\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,

```
<< \context Voice = melody {
     \time 3/4
     c2 e4 g2.
  }
  \new Lyrics \lyricmode {
    \set associatedVoice = #"melody"
    play2 the4 game2.
  } >>
```



A complete example of a SATB score setup is in section Section 3.5 [Vocal ensembles], page 40.

Predefined commands

`\melisma`, `\melismaEnd`

See also

Program reference: `LyricCombineMusic`, `Lyrics`, `Melisma_translator`.

‘`input/regression/lyric-combine-new.ly`’.

Bugs

Melismata are not detected automatically, and extender lines must be inserted by hand.

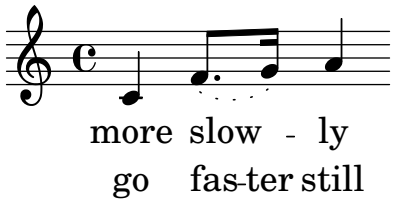
5.11.5 Flexibility in alignment

Often, different stanzas of one song are put to one melody in slightly differing ways. Such variations can still be captured with `\lyricsto`.

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 has one tricky aspect. The setting for `ignoreMelismata` must be set one syllable *before* the non-melismatic syllable in the text, as shown here,

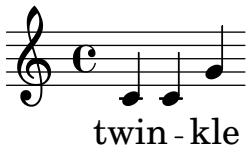
```
<<
  \relative \context Voice = "lahlah" {
    \set Staff.autoBeaming = ##f
    c4
    \slurDotted
    f8.[( g16])
    a4
  }
  \new Lyrics \lyricsto "lahlah" {
    more slow -- ly
  }
  \new Lyrics \lyricsto "lahlah" {
    \set ignoreMelismata = ##t % applies to "fas"
    go fas -- ter
    \unset ignoreMelismata
    still
  }
>>
```



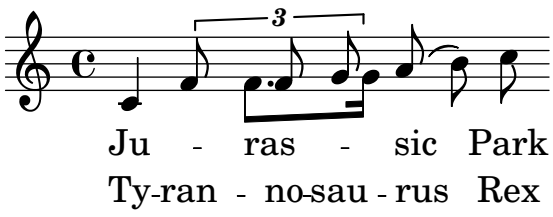
The `ignoreMelismata` applies to the syllable “fas”, so it should be entered before “go”.

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
}
```



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.

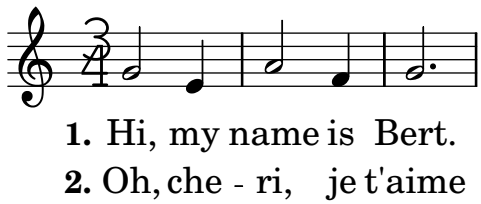
```
\new Lyrics \lyricsto "lahlah" {
  \set associatedVoice = alternative % applies to "ran"
  Ty --
  ran --
  no --
  \set associatedVoice = lahlah % applies to "rus"
  sau -- rus Rex
}
```

The underlay is switched back to the starting situation by assigning `lahlah` to `associatedVoice`.

5.11.6 More stanzas

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
}
```

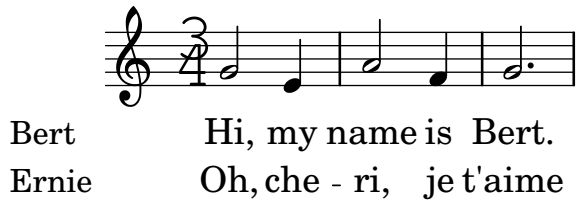


These numbers are put just before the start of first syllable.

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 `vocNam`.

```
\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
}
```



See also

Program reference: Layout objects `LyricText` and `VocalName`. Music expressions `LyricEvent`.

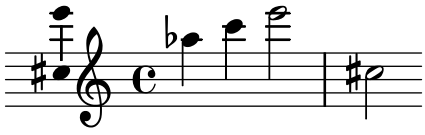
5.11.7 Ambitus

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
  }
}
```

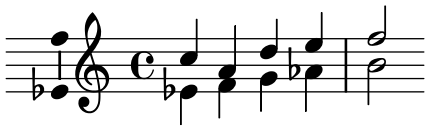
This results in the following output



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-callbacks
      = #(list (lambda (grob axis) -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-callbacks
  = #(list (lambda (grob axis) -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`'.

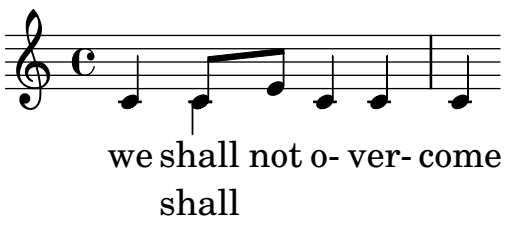
Bugs

There is no collision handling in the case of multiple per-voice ambitus.

5.11.8 Other vocal issue

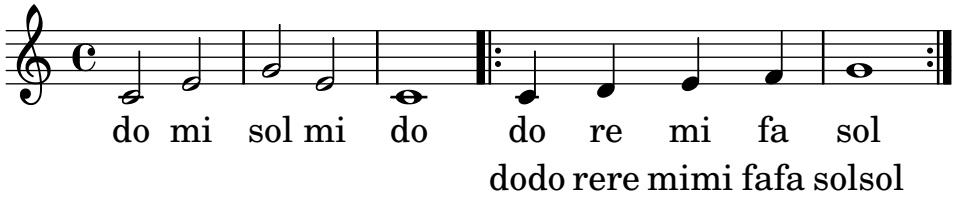
You can display alternate (or divisi) lyrics by naming voice contexts and attaching lyrics to those specific contexts.

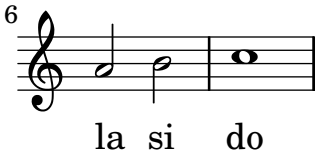
```
\score{ <<
  \context Voice = "melody" {
    \relative c' {
      c4
      <<
        { \voiceOne c8 e }
        \context Voice = splitpart { \voiceTwo c4 }
      >>
      \oneVoice c4 c | c
    }
  }
  \new Lyrics \lyricsto "melody" { we shall not o- ver- come }
  \new Lyrics \lyricsto "splitpart" { shall }
>> }
```



You can use this trick to display different lyrics for a repeated section.

```
\score{ <<
  \context Voice = melody \relative c' {
    c2 e | g e | c1 |
    \context Voice = verse \repeat volta 2 {c4 d e f | g1 | }
    a2 b | c1}
  \lyricsto melody  \context Lyrics = mainlyrics \lyricmode {
    do mi sol mi do
    la si do }
  \lyricsto verse \context Lyrics = mainlyrics \lyricmode {
   do re mi fa sol }
  \lyricsto verse \context Lyrics = repeatlyrics \lyricmode {
   dodo rere mimi fafa solsol }
>>
}
```





5.12 Other instrument specific notation

This section includes extra information for writing string music, and may include extra information for other instruments in the future.

5.12.1 Harmonic notes

Artificial harmonics are notated with a different notehead style. They are entered by marking the harmonic pitch with `\harmonic`.

```
<c' g'\harmonic>4
```

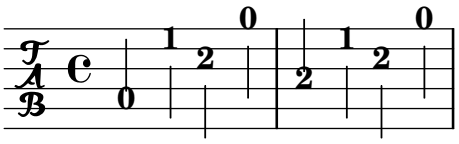
5.13 Tablatures

Tablature notation is used for notating music for plucked string instruments. Pitches are not denoted with note heads, but by indicating on which string and fret a note must be played. LilyPond offers limited support for tablature.

5.13.1 Tablatures basic

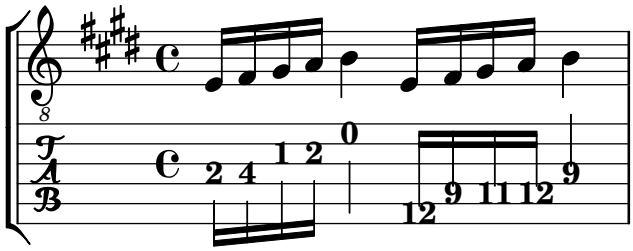
The string number associated to a note is given as a backslash followed by a number, e.g., `c4\3` for a C quarter on the third string. By default, string 1 is the highest one, and the tuning defaults to the standard guitar tuning (with 6 strings). The notes are printed as tablature, by using `TabStaff` and `TabVoice` contexts

```
\context 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
```



See also

Program reference: `TabStaff`, `TabVoice`, and `StringNumberEvent`.

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.

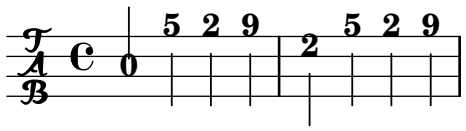
5.13.2 Non-guitar tablatures

You can change the number of strings, by setting the number of lines in the `TabStaff`.

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. In the next example, `stringTunings` is set for the pitches e, a, d, and g

```
\context TabStaff <<
  \set TabStaff.stringTunings = #'(-5 -10 -15 -20)
  {
    a,4 c' a e' e c' a e'
  }
```

```
>>
```



Bugs

No guitar special effects have been implemented.

See also

Program reference: `Tab_note_heads_engraver`.

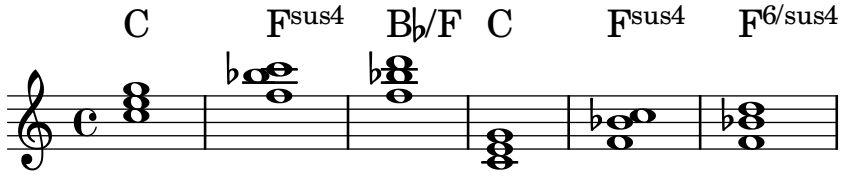
5.14 Popular music

This section discusses issues that arise when writing popular music.

5.14.1 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'>
}

<< \context ChordNames \twoWays
   \context Voice \twoWays >>
```

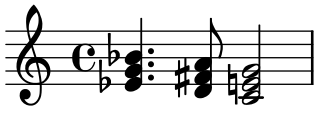


This example also shows that the chord printing routines do not try to be intelligent. The last chord (`f bes d`) is not interpreted as an inversion.

5.14.2 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

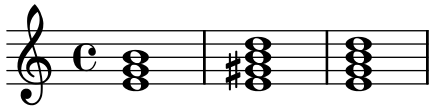
```
\chordmode { es4. d8 c2 }
```



The mode is introduced by the keyword `\chordmode`.

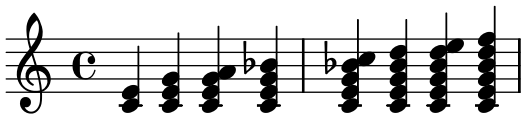
Other chords may be entered by suffixing a colon and introducing a modifier (which may include a number if desired)

```
\chordmode { e1:m e1:7 e1:m7 }
```



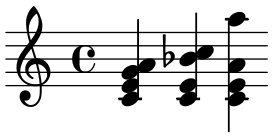
The first number following the root is taken to be the 'type' of the chord, thirds are added to the root until it reaches the specified number

```
\chordmode { c:3 c:5 c:6 c:7 c:8 c:9 c:10 c:11 }
```



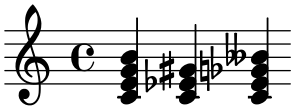
More complex chords may also be constructed adding separate steps to a chord. Additions are added after the number following the colon and are separated by dots

```
\chordmode { c:5.6 c:3.7.8 c:3.6.13 }
```



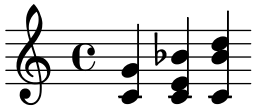
Chord steps can be altered by suffixing a `-` or `+` sign to the number

```
\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.

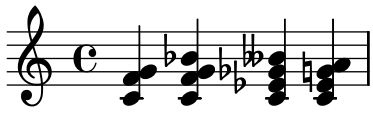
`aug` The augmented chord. This modifier raises the 5th step.

`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.

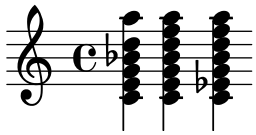
Modifiers can be mixed with additions

```
\chordmode { c:sus4 c:7sus4 c:dim7 c:m6 }
```



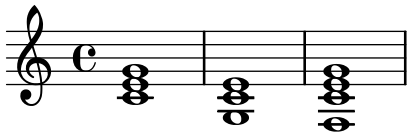
Since an unaltered 11 does not sound good when combined with an unaltered 3, the 11 is removed in this case (unless it is added explicitly)

```
\chordmode { c:13 c:13.11 c:m13 }
```



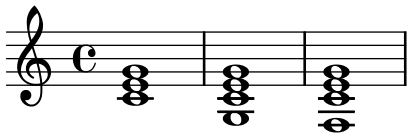
An inversion (putting one pitch of the chord on the bottom), as well as bass notes, can be specified by appending */pitch* to the chord

```
\chordmode { c1 c/g c/f }
```



A bass note can be added instead transposed out of the chord, by using `/+`*pitch*.

```
\chordmode { c1 c/+g c/+f }
```

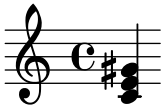


Chords is a mode similar to `\lyricmode`, etc. Most of the commands continue to work, for example, `r` and `\skip` can be used to insert rests and spaces, and property commands may be used to change various settings.

Bugs

Each step can only be present in a chord once. The following simply produces the augmented chord, since `5+` is interpreted last

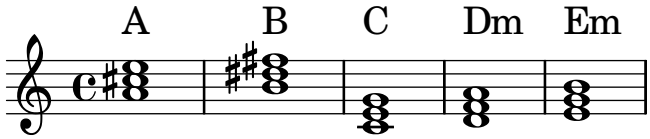
```
\chordmode { c:5.5-.5+ }
```



5.14.3 Printing chord names

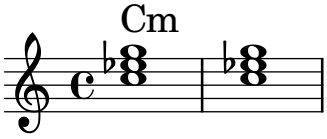
For displaying printed chord names, use the `ChordNames` context. The chords may be entered either using the notation described above, or directly using `<` and `>`

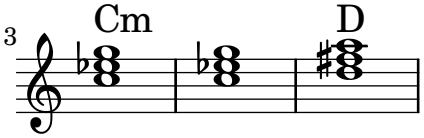
```
harmonies = {
  \chordmode {a1 b c} <d' f' a'> <e' g' b'>
}
<<
  \context ChordNames \harmonies
  \context Staff \harmonies
>>
```



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
}
<<
  \context ChordNames {
    \set chordChanges = ##t
    \harmonies }
  \context Staff \transpose c c' \harmonies
>>
```

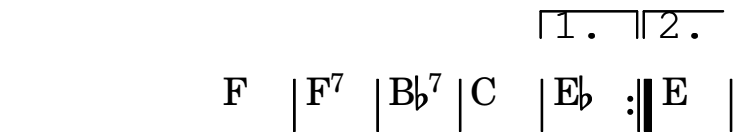




The previous examples all show chords over a staff. This is not necessary. Chords may also be printed separately. It may be necessary to add `Volta_engraver` and `Bar_engraver` for showing repeats.

```
\new ChordNames \with {
  \override BarLine #'bar-size = #4
  voltaOnThisStaff = ##t
  \consists Bar_engraver
  \consists "Volta_engraver"
}
\repeat volta 2 \chordmode {
```

```
  f1:maj f:7 bes:7
  c:maj
} \alternative {
  es e
}
```



The default chord name layout is a system for Jazz music, proposed by Klaus Ignatzek (see Appendix A [Literature list], page 232). 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`'.

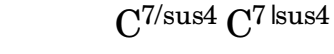
`majorSevenSymbol`

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.,

```
\context ChordNames \chordmode {
  c:7sus4
  \set chordNameSeparator
    = \markup { \typewriter "|" }
  c:7sus4
}
```

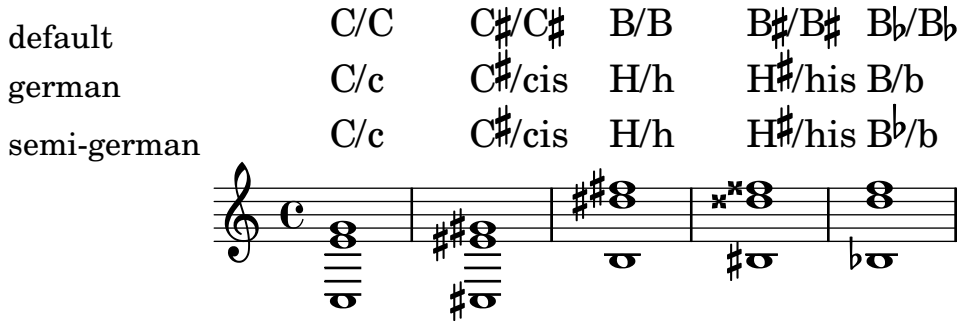


`chordRootNamer`

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.

The predefined variables `\germanChords`, `\semiGermanChords` 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 235. Turning on these styles is described in the input file '`input/test/chord-names-jazz.ly`'.

Predefined commands

`\germanChords`, `\semiGermanChords`.

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

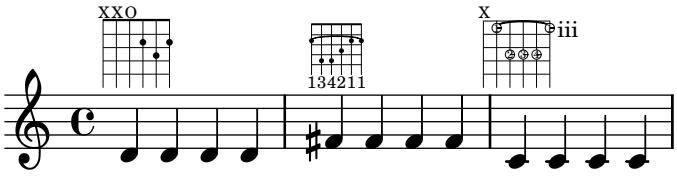
Chord names are determined solely from the list of pitches. Chord inversions are not identified, and neither are added bass notes. This may result in strange chord names when chords are entered with the `< .. >` syntax.

5.14.4 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

```
\context 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 7.4.3 [Overview of text markup commands], page 190.

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`’

5.14.5 Improvisation

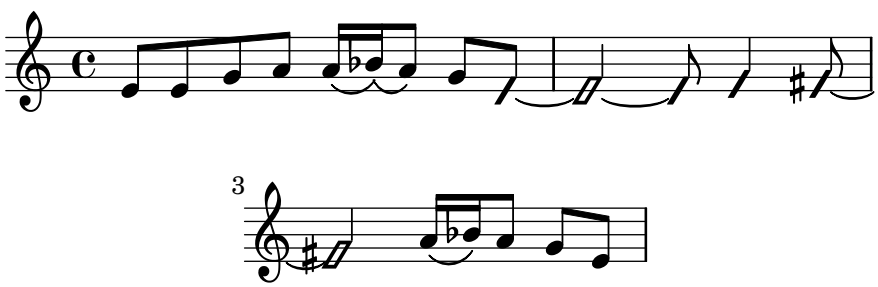
Improvisation is sometimes denoted with slashed note heads. Such note heads can be created by adding a `Pitch_squash_engraver` to the `Staff` or `Voice` context. Then, the following command

```
\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
}
```



5.15 Orchestral music

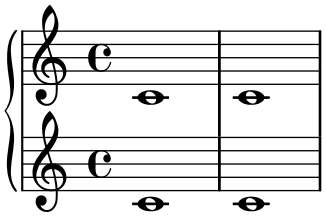
Orchestral music involves some special notation, both in the full score and the individual parts. This section explains how to tackle some common problems in orchestral music.

5.15.1 System start delimiters

Polyphonic scores consist of many staves. These staves can be constructed in three different ways

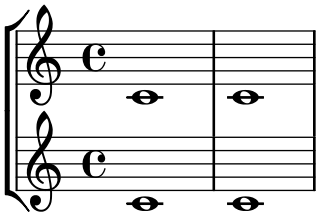
- The group is started with a brace at the left, and bar lines are connected. This is done with the `GrandStaff` context.

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



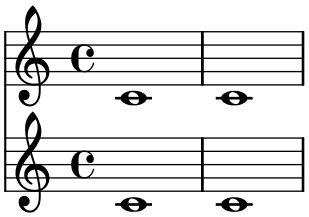
- The group is started with a bracket, and bar lines are connected. This is done with the `StaffGroup` context

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



- The group is started with a 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

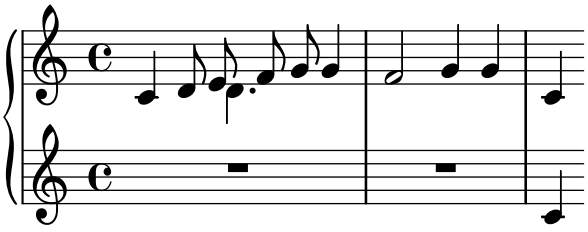
The bar lines at the start of each system are `SystemStartBar`, `SystemStartBrace`, and `SystemStartBracket`. Only one of these types is created in every context, and that type is determined by the property `systemStartDelimiter`.

5.15.2 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.

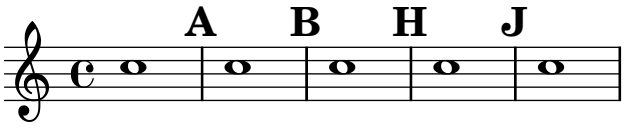
```
cadenza = \relative c' {
  c4 d8 << { e f g } \\ { d4. } >>
  g4 f2 g4 g
}

\new GrandStaff <<
  \new Staff { \cadenza c'4 }
  \new Staff {
    #(ly:export (mmrest-of-length cadenza))
    c'4
  }
>>
```



5.15.3 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
```

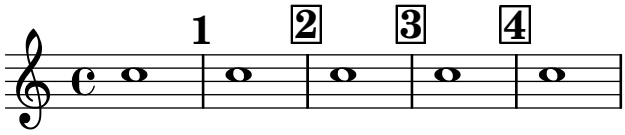


(The letter 'I' is skipped in accordance with engraving traditions.)

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.

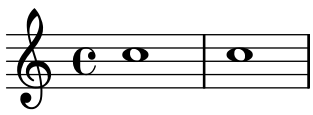
```
\set Score.markFormatter = #format-mark-numbers
c1 \mark \default
c1 \mark \default
\set Score.markFormatter = #format-mark-box-numbers
c1 \mark \default
c1 \mark \default
c1
```



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.

The `\mark` command can also be used to put signs like coda, segno, and fermata on a bar line. Use `\markup` to access the appropriate symbol

```
c1 \mark \markup { \musicglyph #"scripts-ufermata" }
c1
```



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. To print the mark at the end of the current line, use

```
\override Score.RehearsalMark
  #'break-visibility = #begin-of-line-invisible
```

See also

Program reference: `MarkEvent`, `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.

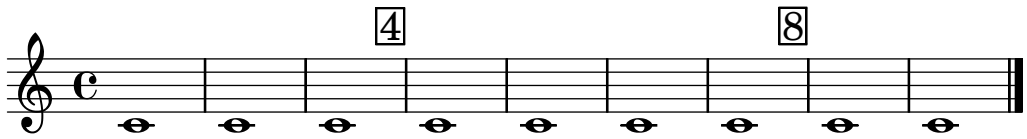
Examples: '`input/regression/rehearsal-mark-letter.ly`',

'`input/regression/rehearsal-mark-number.ly`'.

5.15.4 Bar numbers

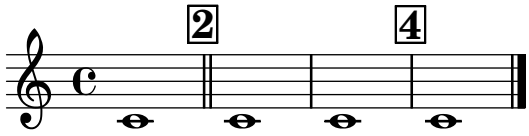
Bar numbers are printed by default at the start of the line. The number itself is stored in the `currentBarNumber` property, which is normally updated automatically for every measure.

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`'



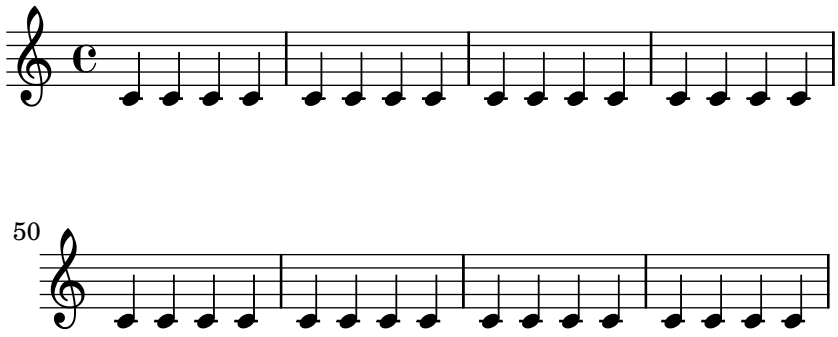
Bar numbers can be typeset manually by tweaking the `markFormatter` property

```
\relative c' {
  \set Score.markFormatter
    = #(lambda (mark context)
      (make-bold-markup
        (make-box-markup
          (number->string (ly:context-property context
                                               'currentBarNumber)))))

  c1 \bar "||" \mark \default c1 c1 \mark \default c1 \bar "|."
}
```



Bar numbers can be manually changed by setting the `Staff.currentBarNumber` property

```
\relative c' {
  \repeat unfold 4 {c4 c c c} \break
  \set Score.currentBarNumber = #50
  \repeat unfold 4 {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

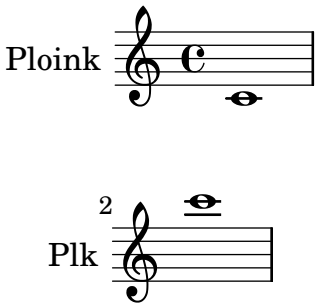
Bar numbers can collide with the `StaffGroup` bracket, if there is one at the top. To solve this, the `padding` property of `BarNumber` can be used to position the number correctly.

5.15.5 Instrument names

In an orchestral score, instrument names are printed at the left side of the staves.

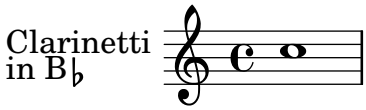
This can be achieved by setting `Staff.instrument` and `Staff.instr`. This will print a string before the start of the staff. For the first staff, `instrument` is used, for the following ones, `instr` is used.

```
\set Staff.instrument = "Ploink "
\set Staff.instr = "Plk "
c1
\break
c''
```



You can also use markup texts to construct more complicated instrument names, for example

```
\set Staff.instrument = \markup {
  \column { "Clarinetti"
            \line { "in B" \smaller \flat } } }
c''1
```



For longer instrument names, it may be useful to increase the `indent` setting in the `\layout` block.

See also

Program reference: `InstrumentName`.

Bugs

When you put a name on a grand staff or piano staff, the width of the brace is not taken into account. You must add extra spaces to the end of the name to avoid a collision.

5.15.6 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

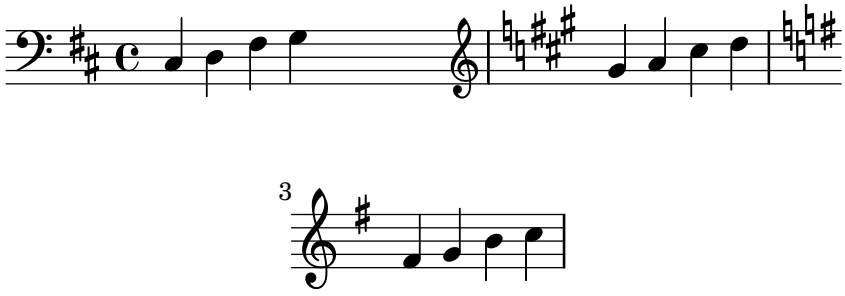
```
\transpose d e ...
```

Consider a part written for violin (a C instrument). If this part is to be played on the A clarinet, 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 }
\context Staff {
  \clef "F" \mus
  \clef "G"
  \transpose c g' \mus
  \transpose c f' \mus
}
```



See also

Program reference: `TransposedMusic`, and `UntransposableMusic`.

Bugs

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

5.15.7 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.

The pitch to use for `\transposition` should correspond to the transposition of the notes. 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'
```

```
  ...
}
```

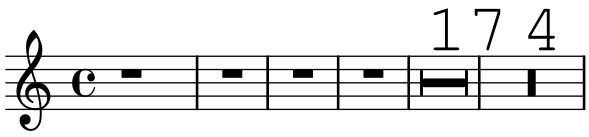
The command `\transposition` should be used when the music is entered from a (transposed) orchestral part. For example, in classical horn parts, the tuning of the instrument is often changed during a piece. When copying the notes from the part, use `\transposition`, e.g.,

```
\transposition d'
c'4^"in D"
...
\transposition g'
c'4^"in G"
...
```

5.15.8 Multi measure rests

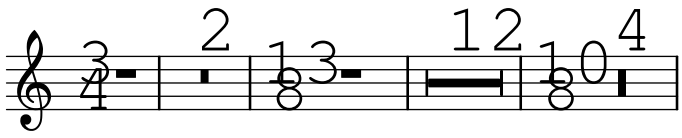
Multi-measure rests are entered using 'R'. It is specifically meant for full bar rests and for entering parts: the rest can expand to fill a score with rests, or it can be printed as a single multi-measure rest. This expansion is controlled by the property `Score.skipBars`. If this is set to true, empty measures will not be expanded, and the appropriate number is added automatically

```
\time 4/4 r1 | R1 | R1*2
\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

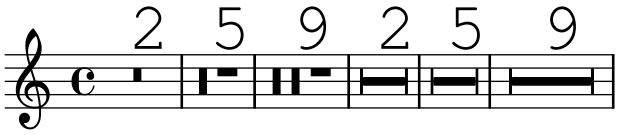
```
\set Score.skipBars = ##t
\time 3/4
R2. | R2.*2
\time 13/8
R1*13/8
R1*13/8*12 |
\time 10/8 R4*5*4 |
```



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.

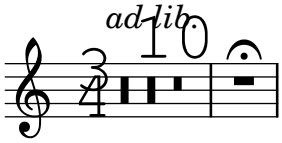
If there are only a few measures of rest, LilyPond prints "church rests" (a series of rectangles) in the staff. To replace that with a simple rest, use `MultiMeasureRest.expand-limit`.

```
\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 (see Section 7.4 [Text markup], page 188). A variable (`\fermataMarkup`) is provided for adding fermatas

```
\set Score.skipBars = ##t
\time 3/4
R2.*10^\markup { \italic "ad lib." }
R2.^\fermataMarkup
```



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: `MultiMeasureRestEvent`, `MultiMeasureTextEvent`, `MultiMeasureRestMusicGroup`, and `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.

5.15.9 Automatic part combining

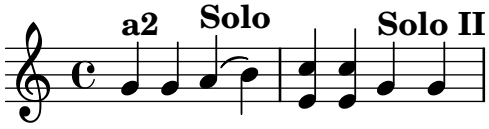
Automatic part combining is used to merge two parts of music onto a staff. It is aimed at typesetting orchestral scores. When the two parts are identical for a period of time, only one is shown. In places where the two parts differ, they are typeset as separate voices, and stem directions are set automatically. 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

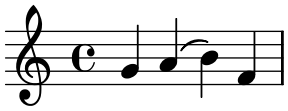
```
\new Staff \partcombine
  \relative g' { g g a( b) c c r r }
  \relative g' { g g r4 r e e g g }
```



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'.

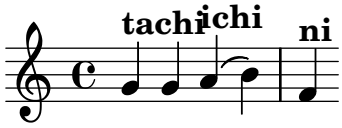
If you just want the merging parts, and not the textual markings, you may set the property `printPartCombineTexts` to false

```
\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`, `SoloOneEvent`, and `SoloTwoEvent`, and `UnisonoEvent`.

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`.

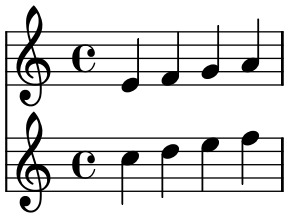
Internally, the `\partcombine` interprets both arguments as `Voice`s named `one` and `two`, and then decides when the parts can be combined. Consequently, if the arguments switch to differently named `Voice` contexts, the events in those will be ignored.

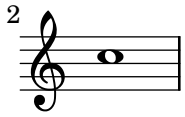
5.15.10 Hiding staves

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 false in `RemoveEmptyVerticalGroup`.

Another application is making ossia sections, i.e., alternative melodies on a separate piece of staff, with help of a Frenched staff. See '`input/test/ossia.ly`' for an example.

5.15.11 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,

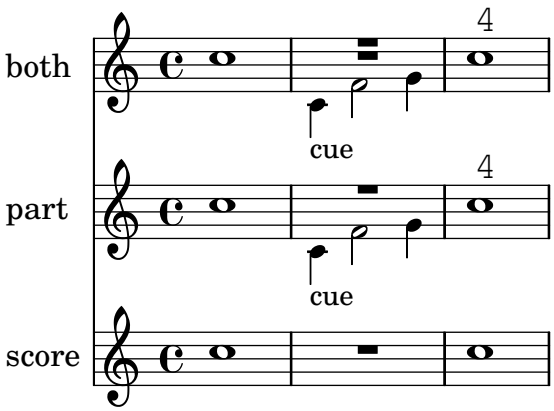
```
c1-\tag #'part ^4
```

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 argument of the `\tag` command should be a symbol, or a list of symbols, for example,

```
\tag #'(original-part 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.

5.15.12 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.

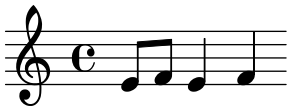
```
\quoteDuring #"clarinet" { s2. }
```

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 `\addquoted` 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.

See also

In this manual: Section 5.15.7 [Instrument transpositions], page 138.

Examples: ‘`input/regression/quote.ly`’ ‘`input/regression/quote-transposition.ly`’

Program reference: `QuoteMusic`.

5.15.13 Formatting cue notes

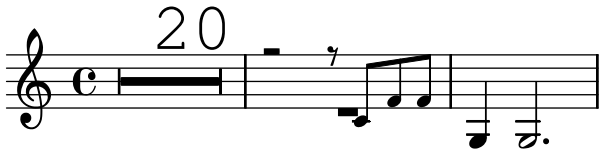
The previous section deals with inserting notes from another voice. There is a more advanced music function called `\cueDuring`, which makes formatting cue notes easier.

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 = #5.5
  \override Beam #'thickness = #0.384
  \override Beam #'space-function =
    #(lambda (beam mult) (* 0.8 (Beam::space_function beam mult)))
}

\addquote clarinet \relative {
  R1*20
  r2 r8 c f f
}

\new Staff \relative  <<

  % setup a context for  cue  notes.
  \context 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.

5.16 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 5.16.1 [Ancient note heads], page 147,
- Section 5.16.2 [Ancient accidentals], page 147,
- Section 5.16.3 [Ancient rests], page 148,
- Section 5.16.4 [Ancient clefs], page 148,
- Section 5.16.5 [Ancient flags], page 150,
- Section 5.16.6 [Ancient time signatures], page 151.

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 5.7.8 [Articulations], page 95, specific articulation signs for ancient notation are provided.

- Section 5.16.7 [Ancient articulations], page 152

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 5.16.8 [Custodes], page 152,
- Section 5.16.9 [Divisiones], page 153,
- Section 5.16.10 [Ligatures], page 154.

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 5.16.11 [Gregorian Chant contexts], page 160,
- Section 5.16.12 [Mensural contexts], page 160.

There is limited support for figured bass notation which came up during the baroque period.

- Section 5.16.13 [Figured bass], page 161

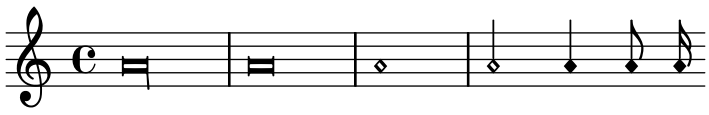
Here are all suptopics at a glance:

5.16.1 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` or `mensural`. 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 finally produces note heads that mimic the look of note heads in historic printings of the 16th century.

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 5.16.10 [Ligatures], page 154 for how ligature engravers work.

See also

Examples: '`input/regression/note-head-style.ly`' gives an overview over all available note head styles.

5.16.2 Ancient accidentals

Use the `style` property of grob `Accidental` to select ancient accidentals. Supported styles are `mensural`, `vaticana`, `hufnagel`, and `medicaea`.

vaticana medicaea hufnagel mensural

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 5.1.2 [Pitches], page 66, Section 5.1.3 [Chromatic alterations], page 67 and Section 5.6 [Accidentals], page 89 give a general introduction of the use of accidentals. Section 5.3.2 [Key signature], page 76 gives a general introduction of the use of key signatures.

Program reference: `KeySignature`.

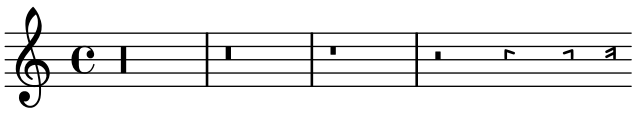
Examples: '`input/test/ancient-accidentals.ly`'.

5.16.3 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 5.16.9 [Divisiones], page 153.

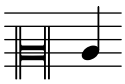
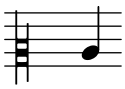
See also

In this manual: Section 5.1.6 [Rests], page 68 gives a general introduction into the use of rests.

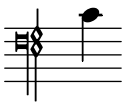
5.16.4 Ancient clefs

LilyPond supports a variety of clefs, many of them ancient.

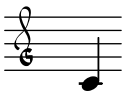
The following table shows all ancient clefs that are supported via the `\clef` command. Some of the clefs use the same glyph, but differ only with respect to the line they are printed on. In such cases, a trailing number in the name is used to enumerate these clefs. Still, you can manually force a clef glyph to be typeset on an arbitrary line, as described in Section 5.3.3 [Clef], page 77. 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`	

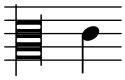
petrucci style mensural F clef `petrucci-f`



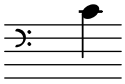
petrucci style mensural G clef `petrucci-g`



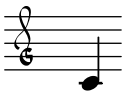
historic style mensural C clef `mensural-c1`, `mensural-c2`, `mensural-c3`, `mensural-c4`



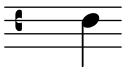
historic style mensural F clef `mensural-f`



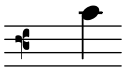
historic style mensural G clef `mensural-g`



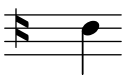
Editio Vaticana style do clef `vaticana-do1`, `vaticana-do2`, `vaticana-do3`



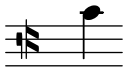
Editio Vaticana style fa clef `vaticana-fa1`, `vaticana-fa2`



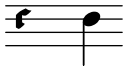
Editio Medicaea style do clef `medicaea-do1`, `medicaea-do2`, `medicaea-do3`



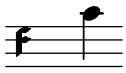
Editio Medicaea style fa clef	`medicaea-fa1`, `medicaea-fa2`



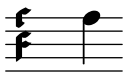
historic style hufnagel do clef	`hufnagel-do1`, `hufnagel-do2`, `hufnagel-do3`



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 5.3.3 [Clef], page 77.

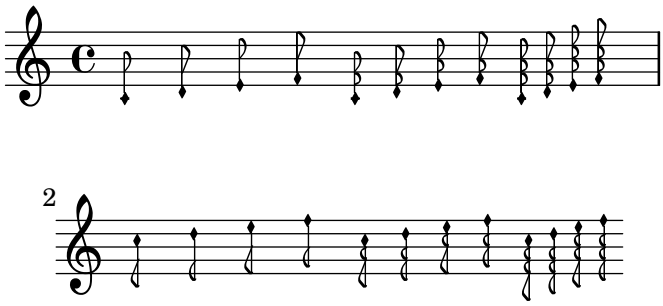
Bugs

The mensural g clef is mapped to the Petrucci g clef.

5.16.5 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.

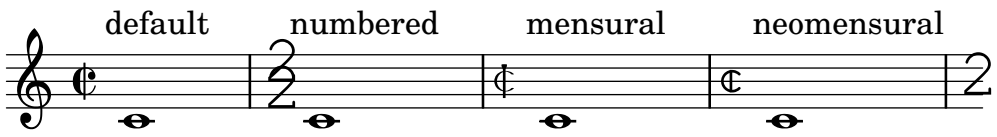
Vertically aligning each flag with a staff line assumes that stems always end either exactly on or exactly in the middle between two staff lines. This may not always be true when using advanced layout features of classical notation (which however are typically out of scope for mensural notation).

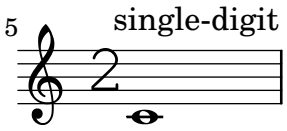
5.16.6 Ancient time signatures

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 5.3.5 [Time signature], page 79 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 }
```

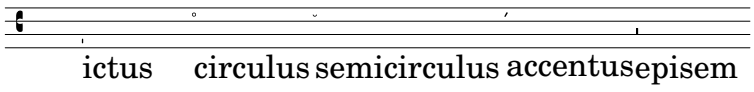
This sets `breveTP` to 3/2 times 2 = 3 times a whole note.

The `old6/8alt` symbol (an alternate symbol for 6/8) is not addressable with `\time`. Use a `\markup` instead

5.16.7 Ancient articulations

In addition to the standard articulation signs described in section Section 5.7.8 [Articulations], page 95, articulation signs for ancient notation are provided. These are specifically designed for use with notation in Editio Vaticana style.

```
\include "gregorian-init.ly"
\score {
  \context VaticanaVoice {
    \override TextScript #'font-family = #'typewriter
    \override TextScript #'font-shape = #'upright
    \override Script #'padding = #-0.1
    a4\ictus_"ictus" s1
    a4\circulus_"circulus" s1
    a4\semicirculus_"semicirculus" s1 s
    a4\accentus_"accentus" s1
    \[ a4_"episem" \episemInitium \pes b \flexa a \episemFinis \]
  }
}
```



Bugs

Some articulations are vertically placed too closely to the correpsonding note heads.

5.16.8 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.

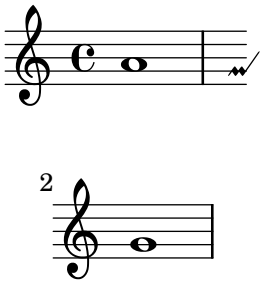
Custodes were frequently used in music notation until the 17th century. Nowadays, they have survived only in a few particular forms of musical notation such as contemporary editions

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

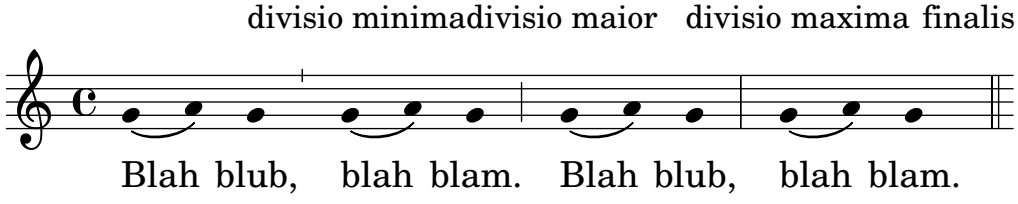
Program reference: `Custos`.

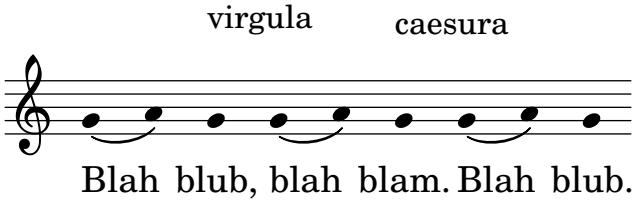
Examples: '`input/regression/custos.ly`'.

5.16.9 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 5.7.3 [Breath marks], page 93. 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 5.7.3 [Breath marks], page 93.

Program reference: `BreathingSign`, `BreathingSignEvent`.

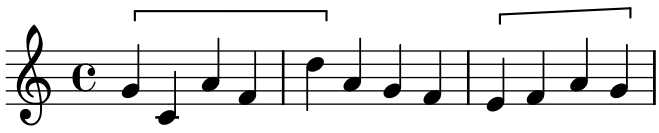
Examples: '`input/test/divisiones.ly`'.

5.16.10 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.

Augmentum dots within ligatures are not handled correctly.

5.16.10.1 White mensural ligatures

There is limited support for white mensural ligatures.

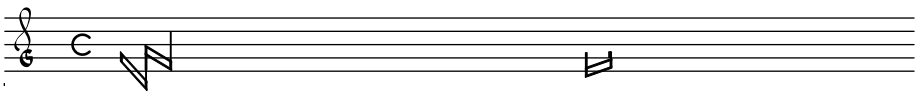
To engrave white mensural ligatures, in the layout block put the `Mensural_ligature_engraver` into the `Voice` context, and remove the `Ligature_bracket_engraver`, like this

```
\layout {
  \context {
    \Voice
    \remove Ligature_bracket_engraver
    \consists Mensural_ligature_engraver
  }
}
```

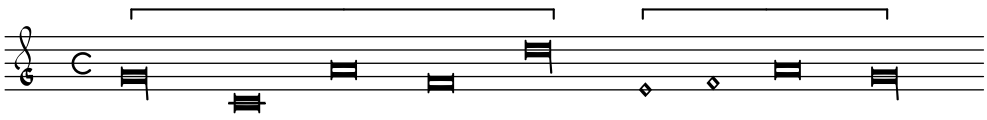
There is no additional input language to describe the shape of a white mensural ligature. The shape is rather determined solely from the pitch and duration of the enclosed notes. While this approach may take a new user a while to get accustomed to, it has the great advantage that the full musical information 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"
\[ g\longa c\breve a\breve f\breve d'\longa \]
s4
\[ e1 f1 a\breve g\longa \]
```



Without replacing `Ligature_bracket_engraver` with `Mensural_ligature_engraver`, the same music transcribes to the following



Bugs

The implementation is experimental. It may output strange warnings, incorrect results, and might even crash on more complex ligatures.

5.16.10.2 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

k l m n

6. Podatus vel Pes

o p q r

7. Pes Quassus

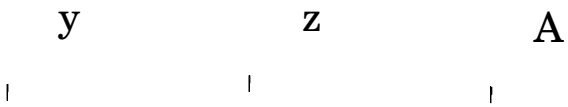
s t

8. Quilisma Pes

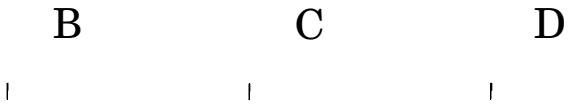
u v

9. Podatus Initio Debilis

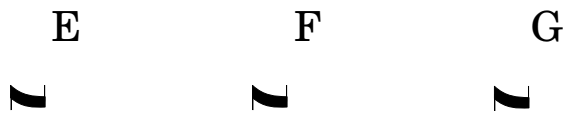
10. Torculus



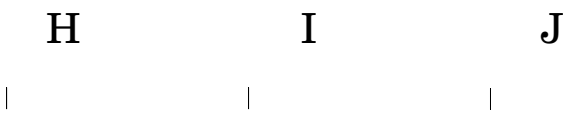
11. Torculus Initio Debilis



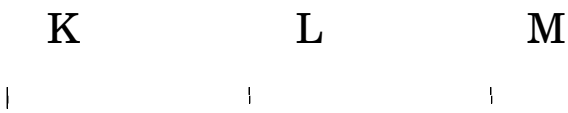
12. Porrectus



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	`\[ \stropha b \stropha b \stropha a \]`

Predefined commands

The following head prefixes are supported

`\virga`, `\stropha`, `\inclinatum`, `\auctum`, `\descendens`, `\ascendens`, `\oriscus`, `\quilisma`, `\deminutum`.

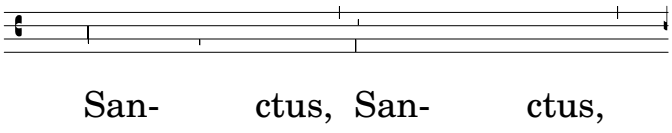
Head prefixes can be accumulated, though restrictions apply. For example, either `\descendens` or `\ascendens` can be applied to a head, but not both to the same head.

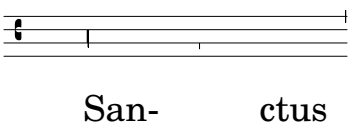
Two adjacent heads can be tied together with the `\pes` and `\flexa` infix commands for a rising and falling line of melody, respectively.

5.16.11 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 {
  <<
    \context VaticanaVoice = "cantus" {
      \override Score.BarNumber #'transparent = ##t {
        \[ c'\melisma c' \flexa a \]
        \[ a \flexa \deminutum g\melismaEnd \]
        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
      }
    }
    \lyricsto "cantus" \new Lyrics {
      San- ctus, San- ctus, San- ctus
    }
  >>
}
```



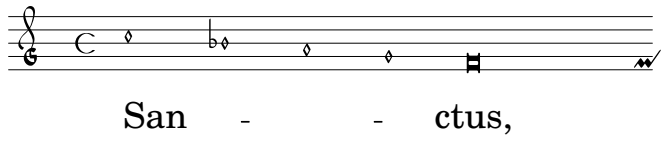


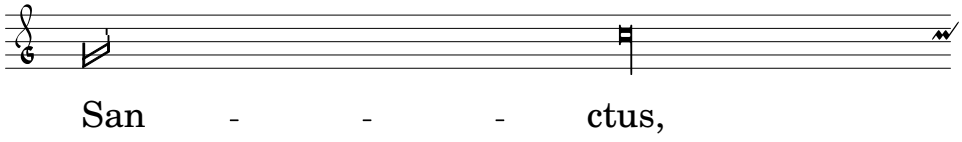
5.16.12 Mensural contexts

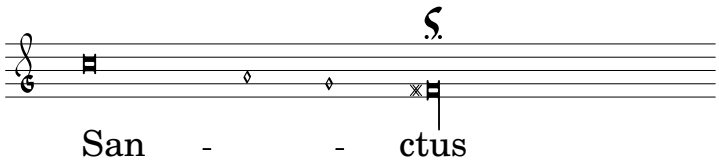
The predefined `MensuralVoiceContext` and `MensuralStaffContext` can be used to engrave a piece in mensural style. These contexts initialize all relevant context properties and grob properties to proper values, so you can immediately go ahead entering the chant, as the following excerpt demonstrates

```
\score {
  <<
    \context 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
      }
    }
    \lyricsto "discantus" \new Lyrics {
      San -- ctus, San -- ctus, San -- ctus
    }
  >>
}
```



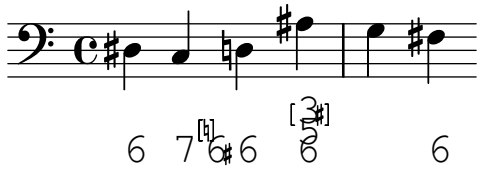




5.16.13 Figured bass

LilyPond has limited support for figured bass

```
<<
  \context Voice { \clef bass dis4 c d ais g fis}
  \context FiguredBass \figuremode {
    < 6 >4 < 7 >8 < 6+ [_!] >
    < 6 >4 <6 5 [3+] >
    < _ >4 < 6 >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

```
<4- 6+ 7!>
```

Spaces or dashes may be inserted by using `_`. Brackets are introduced with `[` and `]`

```
< [4 6] 8 [_! 12] >
```

Although the support for figured bass may superficially resemble chord support, it works 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`.

See also

Program reference: `BassFigureEvent` music, `BassFigure` object, and `FiguredBass` context.

Bugs

Slash notation for alterations is not supported.

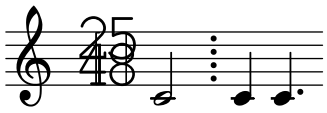
5.17 Contemporary notation

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 232). In general, the use of new, innovative notation makes a piece harder to understand and perform and its use should therefore be avoided. For this reason, support for contemporary notation in LilyPond is limited.

5.17.1 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.

```
% create 2/4 + 5/8
tsMarkup =\markup {
  \number {
    \column { "2" "4" }
    \musicglyph #"scripts-stopped"
    \bracket \column { "5" "8" }
  }
}

{
  \override Staff.TimeSignature #'print-function = #Text_interface::print
  \override Staff.TimeSignature #'text = #tsMarkup
  \time 3/2
  c'2 \bar ":" c'4 c'4.
}
```

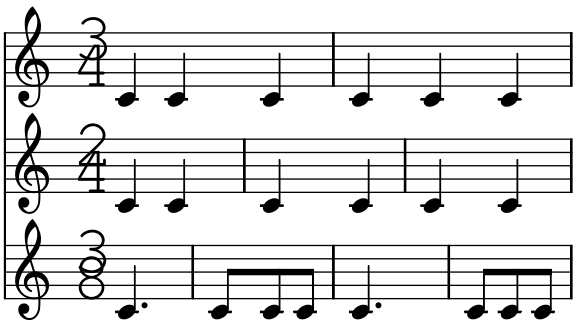


Each staff can also have its own time signature. This is done by moving the `Timing_engraver` to the `Staff` context.

```
\layout {
  \context { \Score \remove "Timing_engraver" }
  \context { \Staff \consists "Timing_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
  }
>>
```

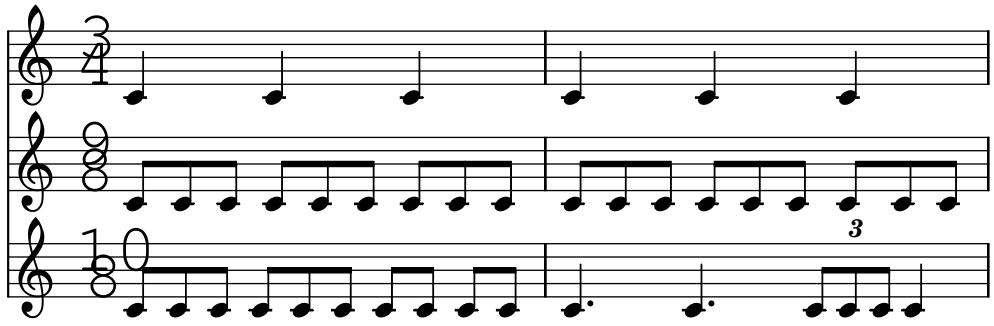


A different form of polymetric notation is where note lengths have different values across staves.

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 similar to `\times`, but does not create a tuplet bracket.

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
    }
  }
>> }
```



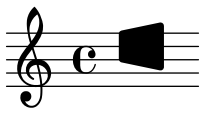
Bugs

When using different time signatures in parallel, the spacing is aligned vertically, but bar lines distort the regular spacing.

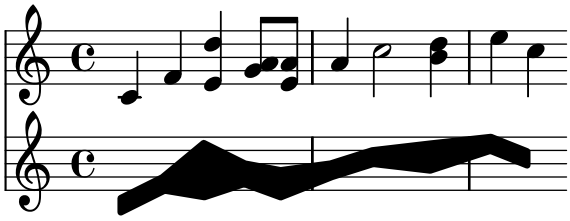
5.17.2 Clusters

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`, and `ClusterNoteEvent`.

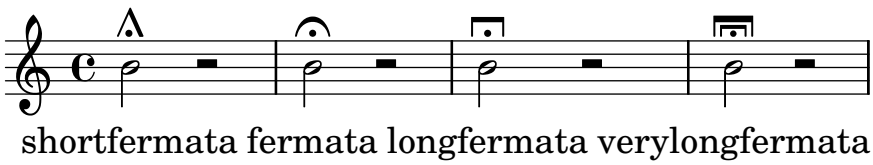
Examples: '`input/regression/cluster.ly`'.

Bugs

Music expressions like `<< { g8 e8 } a4 >>` are not printed accurately. Use `<g a>8 <e a>8` instead.

5.17.3 Special fermatas

In contemporary music notation, special fermata symbols denote breaks of differing lengths. The following fermatas are supported

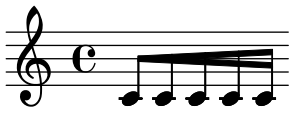


See Section 5.7.8 [Articulations], page 95 for general instructions how to apply scripts such as fermatas to notes.

5.17.4 Feathered beams

Feathered beams are not supported natively, but they can be faked by forcing two beams to overlap. Here is an example,

```
\new Staff <<
  \new Voice
  {
    \stemUp
    \once \override Voice.Beam #'positions = #'(0 . 0.5)
    c8[ c c c c ]
  }
  \new Voice {
    \stemUp
    \once \override Voice.Beam #'positions = #'(0 . -0.5)
    c[ c c c c]
  }
>>
```



5.18 Educational use

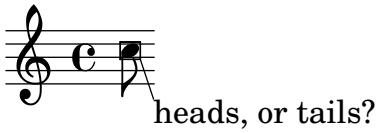
With the amount of control that LilyPond offers, one can make great teaching tools in addition to great musical scores.

5.18.1 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.

```
\context Voice {
  \applyoutput
    #(add-balloon-text 'NoteHead "heads, or tails?"
    '(1 . -3))
  c8
}
```



The function `add-balloon-text` takes the name of a grob, the label to print, and the position where to put the label relative to the object. In the above example, the text "heads or tails?" ends 3 spaces below and 1 space to the right of the marked head.

See also

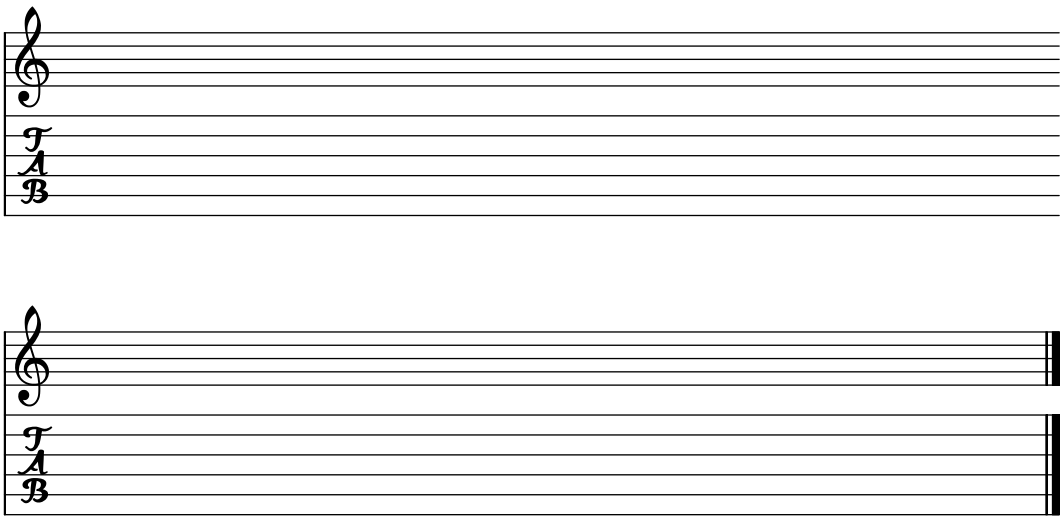
Program reference: `text-balloon-interface`.

Examples: '`input/regression/balloon.ly`'.

5.18.2 Blank music sheet

A blank music sheet can be produced also by using invisible notes, and removing `Bar_number_engraver`.

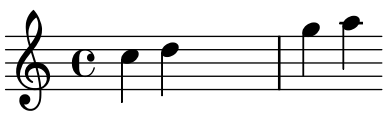
```
emptymusic = {
  \repeat unfold 2 % Change this for more lines.
  { s1\break }
  \bar "|."
}
\new Score \with {
  \override TimeSignature #'transparent = ##t
  defaultBarType = #""
  \remove Bar_number_engraver
} <<
  \context Staff \emptymusic
  \context TabStaff \emptymusic
>>
```



5.18.3 Hidden notes

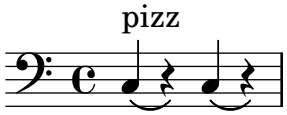
Hidden (or invisible or transparent) notes can be useful in preparing theory or composition exercises.

```
c4 d4
\hideNotes
e4 f4
\unHideNotes
g4 a
```



Hidden notes are also great for performing weird tricks. For example, slurs cannot be attached to rests or spacer rests, but you may wish to include that in your score – string instruments use this notation when doing pizzicato to indicate that the note should ring for as long as possible.

```
\clef bass
<< {
  c4^"pizz"( \hideNotes c)
  \unHideNotes c( \hideNotes c)
} {
  s4 r s r
} >>
```

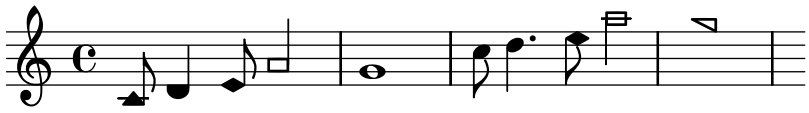


5.18.4 Shaped note heads

In shaped 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.

Shaped note heads can be produced by setting `\aikenHeads` or `\sacredHarpHeads`, depending on the style desired.

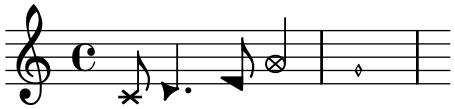
```
\aikenHeads
c8 d4 e8 a2 g1
\sacredHarpHeads
c8 d4. e8 a2 g1
```



Shapes are determined on the step in the scale, where the base of the scale is determined by the `\key` command

Shaped 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, eg.,

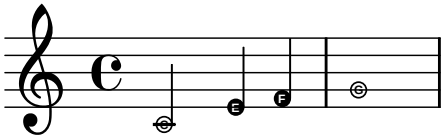
```
\set shapeNoteStyles  = ##(cross triangle fa #f mensural xcircle diamond)
c8 d4. e8 a2 g1
```



5.18.5 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 7.5.1 [Setting global staff size], page 197.

Predefined commands

`\setEasyHeads`

6 Sound

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 (`http://timidity.sourceforge.net/`).

6.1 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 { \tempo 4=72 }
}
```

The tempo is specified using the `\tempo` command. In this example the tempo of quarter notes is set to 72 beats per minute.

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 { \tempo 4=72 }
  \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`.

6.2 MIDI block

The MIDI block is analogous to the layout block, but it is somewhat simpler. The `\midi` block can contain

- a `\tempo` definition, and
- context definitions.

A number followed by a period is interpreted as a real number, so for setting the tempo for dotted notes, an extra space should be inserted, for example

```
\midi { \tempo 4 . = 120 }
```

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`'.

6.3 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 236.

```
\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.

7 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 (`http://lilypond.org/doc/Documentation/user/out-www/lilypond-internals/`), but is also included with the LilyPond documentation package.

There are three areas where the default settings may be changed:

- 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.
- Global layout: changing the appearance of the spacing, line breaks, and page dimensions.

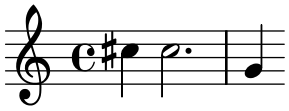
Then there are separate systems for typesetting text (like *ritardando*) and selecting different fonts. This chapter also discusses these.

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]

7.1 Interpretation contexts

When music is printed, a lot of notational elements must be added to the input, which is often bare bones. 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 what pitches need 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 so-called Contexts. Examples of context are `Voice`, `Staff`, and `Score`. They are hierarchical, for example, a `Staff` can contain many `Voice`s, 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. So, the synchronization of bar lines is handled at `Score` context. The `Voice` may introduce an accidental and then the `Staff` context maintains the rule to show or suppress the accidental for the remainder of the measure.

[1] Appendix B [Scheme tutorial], page 233 contains a short tutorial on entering numbers, lists, strings, and symbols in Scheme.

For simple scores, contexts are created implicitly, and you need not be aware of them. For larger pieces, such as piano music, 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.

7.1.1 Creating contexts

For scores with only one voice and one staff, correct contexts are created automatically. For more complex scores, it is necessary to create them by hand. There are three commands that do this.

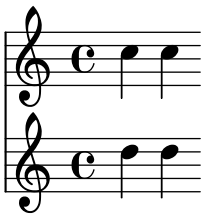
The easiest command is `\new`, and it also the quickest to type. It is prepended to a music expression, for example

```
\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 }
>>
```



Like `\new`, the `\context` command also directs a music expression to a context object, but gives the context an extra 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, it 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,

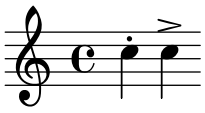
```
\new Lyrics \lyricsto "tenor" lyrics
```

Another possibility 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 8.3.2 [Running a function on all layout objects], page 217). Without an explicit `\context`, it is usually applied to `Voice`

```
\applyoutput #function   % apply to Voice
```

To have it interpreted at the `Score` or `Staff` level use these forms

```
\context Score \applyoutput #function
\context Staff \applyoutput #function
```

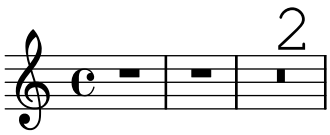
7.1.2 Changing context properties on the fly

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,

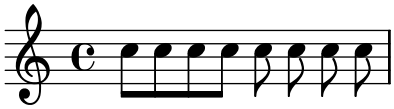
```
R1*2
\set Score.skipBars = ##t
R1*2
```



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.

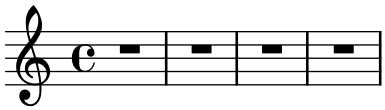
If the *context* argument is left out, then the current bottom-most context (typically `ChordNames`, `Voice`, or `Lyrics`) is used. In this example,

```
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 `Voice`s 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
```

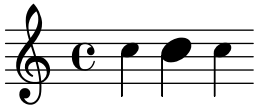
Like `\set`, the *context* argument does not have to be specified for a bottom context, so the two statements

```
\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.

7.1.3 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`.

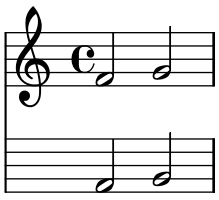
For a full a description of each plug-in, see Program reference ⇒ Translation ⇒ Engravers. Every context described in Program reference ⇒ Translation ⇒ Context. lists the engravers used for that context.

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 like this,

```
\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
  }
>>
```

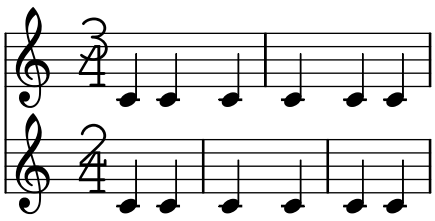


In the second staff there are no time signature or clef symbols. This is a rather crude method of making objects disappear since it will affect the entire staff. The spacing is adversely influenced too. A more sophisticated method of blanking objects is shown in Section 7.2.1 [Common tweaks], page 181.

The next example shows a practical application. Bar lines and time signatures are normally synchronized across the score. This is done by the `Timing_engraver`. This plug-in keeps an administration of time signature, location within the measure, etc. By moving the `Timing_engraver` engraver from `Score` to `Staff` context, we can have a score where each staff has its own time signature.

```
\new Score \with {
  \remove "Timing_engraver"
} <<
```

```
  \new Staff \with {
    \consists "Timing_engraver"
  } {
      \time 3/4
      c4 c c c c c
  }
  \new Staff \with {
    \consists "Timing_engraver"
  } {
       \time 2/4
       c4 c c c c c
  }
>>
```



7.1.4 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
```

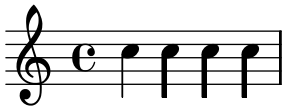
Here *name* is the name of a graphical object, like `Stem` or `NoteHead`, and *property* is an internal variable of the formatting system ('grob property' or 'layout property'). The latter is a symbol, so it must be quoted. The subsection Section 7.2.2 [Constructing a tweak], page 182 explains what to 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:

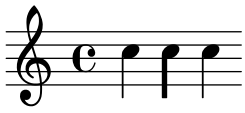
```
c4
\override Staff.Stem #'thickness = #4.0
c4
c4
c4
```



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 it to default to `Voice`, and 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, like 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. Therefore 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
```

See also

Internals: `OverrideProperty`, `RevertProperty`, `PropertySet`, `All-backend-properties`, 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.

7.1.5 Changing context default settings

The adjustments of the previous subsections (Section 7.1.2 [Changing context properties on the fly], page 174, Section 7.1.3 [Modifying context plug-ins], page 176, and Section 7.1.4 [Layout tunings within contexts], page 177) 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"
  }
}
```

Here

```
\Staff
```

takes the existing definition for context `Staff` from the identifier `\Staff`.

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 one `\context` definition by referring to that variable.

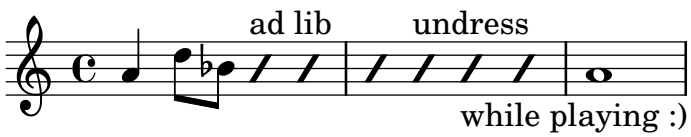
The `\RemoveEmptyStaffContext` will override your current `\Staff` variable. If you wish to change the defaults for a staff that uses `\RemoveEmptyStaffContext`, you must do so after calling `\RemoveemptyStaffContext`, ie

```
\layout {
  \context {
    \RemoveEmptyStaffContext

    \override Stem #'thickness = #4.0
  }
}
```

7.1.6 Defining new contexts

Specific contexts, like `Staff` and `Voice`, are made of simple building blocks, and it is possible to compose engraver plug-ins in different combinations, thereby creating new types of contexts.

The next example shows how to build a different type of `Voice` context from scratch. It will be similar to `Voice`, but prints centered slash noteheads only. It can be used to indicate improvisation in Jazz pieces,



These settings are again done within a `\context` block inside a `\layout` block,

```
\layout {
  \context {
    ...
  }
}
```

In the following discussion, the example input shown should go on the ... in the previous fragment.

First, the context gets a name. Instead of `Voice` it will be called `ImproVoice`,

```
\name ImproVoice
```

Since it is similar to the `Voice`, we want commands that work on (existing) `Voice`s to remain working. This is achieved by giving the new context an alias `Voice`,

```
\alias Voice
```

The context will print notes, and instructive texts

```
\consists Note_heads_engraver
\consists Text_engraver
```

but only 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, without a 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_engraver`,

```
\type "Engraver_group_engraver"
```

Put together, we get

```
\context {
  \name ImproVoice
  \type "Engraver_group_engraver"
  \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 `Voice`s. Therefore, we modify the `Staff` definition with the `\accepts` command,[2]

```
\context {
  \Staff
  \accepts ImproVoice
}
```

Putting both into a `\layout` block, like

[2] The opposite of `\accepts` is `\denies`, which is sometimes needed when reusing existing context definitions.

```
\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
}
```

7.2 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. First, we will give a few versatile commands that are sufficient for many situations. The next section will discuss the general use of `\override`.

7.2.1 Common tweaks

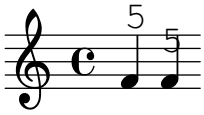
Some overrides are so common that predefined commands are provided as short-cuts, for example, `\slurUp` and `\stemDown`. These commands are described in Chapter 5 [Notation manual], page 66, under the sections for slurs and stems respectively.

The exact tuning possibilities for each type of layout object are documented in the program reference of the respective object. However, many layout objects share properties, which can be used to apply generic tweaks. We mention a few of these:

- The `extra-offset` property, which has a pair of numbers as value, moves objects around in the printout. The first number controls left-right movement; a positive number will move the object to the right. The second number controls up-down movement; a positive number will move it higher. The units of these offsets are staff-spaces. The `extra-offset` property is a low-level feature: the formatting engine is completely oblivious to these offsets.

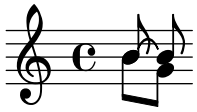
  In the following example, the second fingering is moved a little to the left, and 1.8 staff space downwards:

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



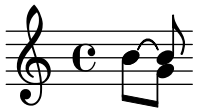
- Setting the `transparent` property will cause an object to be printed in 'invisible ink': the object is not printed, but all its other behavior is retained. The object still takes up space, it takes part in collisions, and slurs, ties, and beams can be attached to it.

  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,



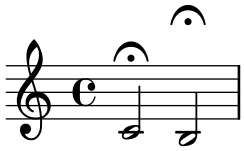
  and blanking the first up-stem in that voice, the tie appears to cross voices:

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



- The `padding` property for objects with `side-position-interface` can be set to increase the distance between symbols that are printed above or below notes. We only give an example; a more elaborate explanation is in Section 7.2.2 [Constructing a tweak], page 182:

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



More specific overrides are also possible. The next section discusses in depth how to figure out these statements for yourself.

7.2.2 Constructing a tweak

The general procedure of changing output, that is, entering a command like

```
\override Voice.Stem #'thickness = #3.0
```

means that we have to determine these bits of information:

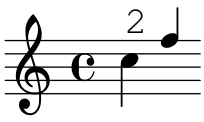
- the context: here `Voice`.
- the layout object: here `Stem`.
- the layout property: here `thickness`
- a sensible value: here `3.0`

We demonstrate how to glean this information from the notation manual and the program reference.

7.2.3 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 5.7.10 [Fingering instructions], page 97), you will notice that there is written:

> **See also**
>
> Program reference: `FingerEvent` and `Fingering`.

This fragment points to two parts of the program reference: a page on `FingerEvent` and one on `Fingering`.

The page on `FingerEvent` describes the properties of the music expression for the input `-2`. The page contains many links forward. For example, it says

> Accepted by: `Fingering_engraver`,

That link brings us to the documentation for the Engraver, the plug-in, which says

> This engraver creates the following layout objects: `Fingering`.

In other words, once the `FingerEvent`s are interpreted, the `Fingering_engraver` plug-in will process them. The `Fingering_engraver` is also listed to create `Fingering` objects,

Lo and behold, that is also the second bit of information listed under **See also** in the Notation manual. By clicking around in the program reference, we can follow the flow of information within the program, either forward (like we did here), or backwards, following links like this:

- `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.

The program reference can also be browsed like a normal document. It contains a chapter on `Music definitions` on `Translation`, and the `Backend`. Every chapter lists all the definitions used, and all properties that may be tuned.

7.2.4 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 *grob*, which is short for Graphical Object.

The page for `Fingering` lists the definitions for the `Fingering` object. For example, the page says

> `padding` (dimension, in staff space):
>
> `0.6`

which means that the number will be kept at a distance of at least 0.6 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 super- 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 are.

We have been talking of *the* `Fingering` object, but actually it does not amount to much. The initialization file '`scm/define-grobs.scm`' shows the soul of the 'object',

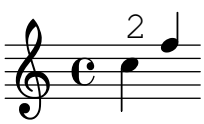
```
(Fingering
  . ((print-function . ,Text_interface::print)
     (padding . 0.6)
     (staff-padding . 0.6)
     (self-alignment-X . 0)
     (self-alignment-Y . 0)
     (script-priority . 100)
     (font-size . -5)
     (meta . ((interfaces . (finger-interface font-interface
                             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.

7.2.5 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)

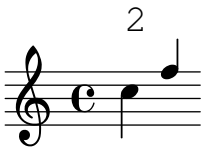
Add this much extra space between objects that are next to each other.

By increasing the value of `padding`, we can move away the fingering. 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`

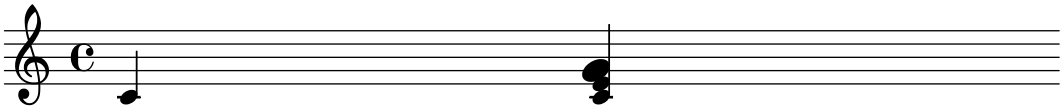
7.2.6 Difficult tweaks

There are two classes of difficult adjustments. First, when there are several of the same objects at one point, and you want to adjust only one. For example, if you want to change only one note head in a chord.

In this case, the `\applyoutput` function must be used. The next example defines a Scheme function `set-position-font-size` that sets the `font-size` property, but only on objects that have `note-head-interface` and are at the right Y-position.

```
#(define ((set-position-font-size pos size) grob origin current)
  (let*
      ((interfaces (ly:grob-property grob 'interfaces))
       (position (ly:grob-property grob 'staff-position)))
   (if (and
        ; is this a note head?
        (memq 'note-head-interface interfaces)

        ; is the Y coordinate right?
        (= pos position))

      ; then do it.
      (set! (ly:grob-property grob 'font-size) size))))

\relative {
```

```
  c
  \applyoutput #(set-position-font-size -2 4)
  <c e g>
}
```



A similar technique can be used for accidentals. In that case, the function should check for `accidental-interface`.

Another 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 `\override`s.

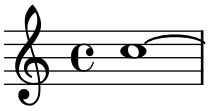
In other words, an `\override` always affects all pieces of a broken spanner. To change only one part of a spanner at a line break, it is necessary to hook into the formatting process. The `after-line-breaking-callback` property contains the Scheme procedure that is called after the line breaks have been determined, and layout objects have been split over different systems.

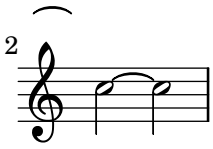
In the following example, we define a procedure `my-callback`. This procedure

- 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-callback =
  #my-callback
  c1 ~ \break c2 ~ c
}
```





When applying this trick, the new `after-line-breaking-callback` should also call the old `after-line-breaking-callback`, if there is one. For example, if using this with `Slur`, `Slur::after_line_breaking` should also be called.

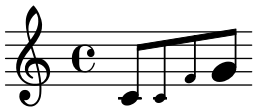
7.3 Fonts

This section details the ways that the font can be changed.

7.3.1 Selecting font sizes

The easiest method of setting the font size of any context, is by setting the `fontSize` property.

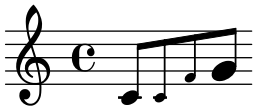
```
c8
\set fontSize = #-4
c f
\set fontSize = #3
g
```



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.

```
c8
\override NoteHead #'font-size = #-4
c f
\override NoteHead #'font-size = #3
g
```



LilyPond has fonts in different design sizes. The music fonts for smaller sizes are chubbier, while the text fonts are relatively wider. 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` mechanism does not work for fonts selected through `font-name`. These may be scaled with `font-magnification`. 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`.

7.3.2 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 and TEX's Computer Modern text fonts.

- `font-encoding` is a symbol that sets layout of the glyphs. This should only be set to select different types of non-text fonts, eg.

  `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.

The font used for printing a object can be selected by setting `font-name`, e.g.,

```
\override Staff.TimeSignature
    #'font-name = #"cmr17"
```

Any font can be used, as long as it is available to TEX. Possible fonts include foreign fonts or fonts that do not belong to the Computer Modern font family. The size of fonts selected in this way can be changed with the `font-magnification` property. For example, `2.0` blows up all letters by a factor 2 in both directions.

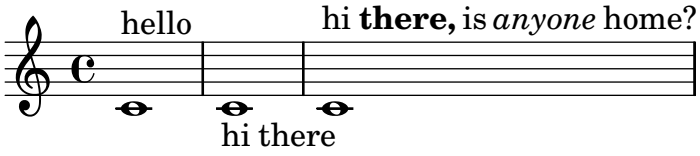
See also

Init files: '`ly/declarations-init.ly`' contains hints how new fonts may be added to LilyPond.

7.4 Text markup

The internal mechanism to typeset texts is accessed with the keyword `\markup`. Within markup mode, you can enter texts similar to lyrics. They are simply entered, while commands use the backslash `\`.

```
c1^\markup { hello }
c1_\markup { hi there }
c1^\markup { hi \bold there, is \italic anyone home? }
```



The markup in the example demonstrates font switching commands. The command `\bold` and `\italic` apply to the first following word only; enclose a set of texts with braces to apply a command to more words:

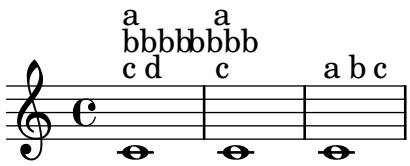
```
\markup { \bold { hi there } }
```

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 } }
```



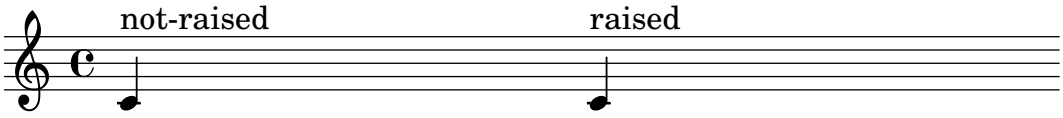
Markups can be stored in variables and these variables may be attached to notes, like

```
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.

Similarly, for moving whole texts over notes with `\raise`, use the following trick:

```
{
  c'^\markup { \raise #0.5 not-raised }
  c'^\markup { "" \raise #0.5 raised }
}
```



On the second note, the text `raised` is moved relative to the empty string `""` which is not visible. Alternatively, complete objects can be moved with layout properties such as `padding` and `extra-offset`.

See also

Init files: '`scm/new-markup.scm`'.

Bugs

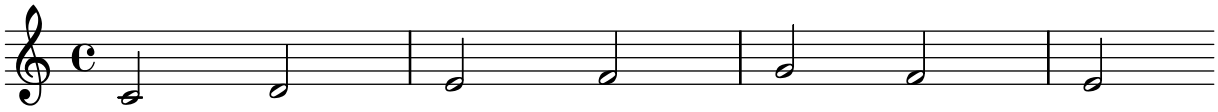
Kerning or generation of ligatures is only done when the TEX backend is used. In this case, LilyPond does not account for them so texts will be spaced slightly too wide.

Syntax errors for markup mode are confusing.

7.4.1 Text encoding

LilyPond uses the Pango library to format multi-lingual texts. This means that any text, be it title, lyric text, or musical instruction contaning non-ASCII characters should be entered as Unicode.

Depending on the fonts installed, the following fragment shows Hebrew and Cyrillic lyrics,



Жълтата дюля беше щастлива, че пухът, който

The TEX backend does not handle encoding specially at all. Strings in the input are put in the output as-is. Extents of text items in the TEX backend, are determined by reading a file created via the '`texstr`' backend,

```
lilypond -b texstr input/les-nereides.ly
latex les-nereides.texstr
```

The last command produces '`les-nereides.textmetrics`', which is read when you execute

```
lilypond -b tex input/les-nereides.ly
```

Both '`les-nereides.texstr`' and '`les-nereides.tex`' need suitable LaTeX wrappers to load appropriate LaTEX packages for interpreting non-ASCII strings.

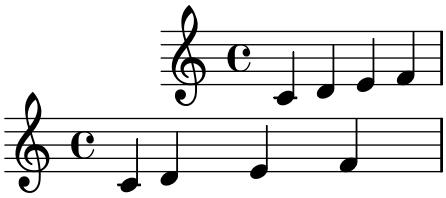
See also

'`input/regression//utf8.ly`'

7.4.2 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
}
```



7.4.3 Overview of text markup commands

The following commands can all be used inside `\markup { }`.

`\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)
: Set `font-shape` to `caps`.

`\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'.

`\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.

`\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

`\encoded-simple` *sym* (symbol) *str* (string)
: A text string, encoded with encoding *sym*. See Section 7.4.1 [Text encoding], page 189 for more information.

`\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.

`\finger` *arg* (markup)
: Set the argument as small numbers.

`\flat`
: Draw a flat symbol.

`\fontsize` *mag* (number) *arg* (markup)
: This sets the relative font size, e.g.

```
A \fontsize #2 { B C } D
```

: This will enlarge the B and the C by two steps.

`\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 #'((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` *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*.

`\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.

`\magnify` *sz* (number) *arg* (markup)
: This sets 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.

`\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 Section C.3 [The Feta font], page 237 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.

`\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.

`\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"
```

`\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
```

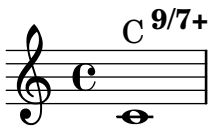
The magical constant 1.75 scales from LilyPond units (staff spaces) to TeX dimensions.

For the postscript backend, use the following

```
gsave /ecrm10 findfont
 10.0 output-scale div
 scalefont setfont 90 rotate (hello) show grestore
```

`\raise` *amount* (number) *arg* (markup)
: This raises *arg*, by the distance *amount*. A negative *amount* indicates lowering:

```
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`.

`\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" }`.

`\small` *arg* (markup)
: Set font size to -1.

`\smaller` *arg* (markup)
: Decrease the font size relative to current setting

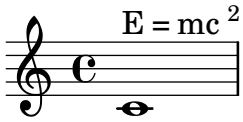
`\stencil` *stil* (unknown)
: Stencil as markup

`\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.

`\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.

`\typewriter` *arg* (markup)
: Use `font-family` typewriter for *arg*.

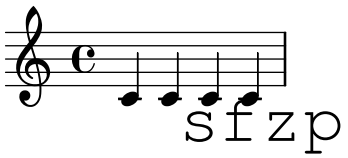
`\upright` *arg* (markup)
: Set font shape to `upright`.

`\vcenter` *arg* (markup)
: Align `arg` to its Y center.

7.4.4 New dynamic marks

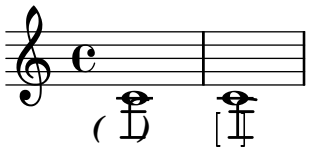
It is possible to print new dynamic marks or text that should be aligned with dynamics. Use `make-dynamic-script` to create these marks.

```
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.

```
\version "2.4.2"
rndf = \markup{ \center-align {\line { \bold{\italic (}
  \dynamic f \bold{\italic )} }} }
boxf = \markup{ \bracket { \dynamic f } }
{ c'1_\rndf c'1_\boxf }
```



7.5 Global layout

The global 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 spoken, this procedure happens in three steps: first, flexible distances ("springs") are chosen, based on durations. All possible line breaking combinations are tried, and the one with the best results – a layout that has uniform density and requires as little stretching or cramping as possible – is chosen.

After spacing and linebreaking, the systems are distributed across pages, taking into account the size of the page, and the size of the titles.

7.5.1 Setting global staff size

To set the global staff size, 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.

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 7.3.1 [Selecting font sizes], page 187.

7.5.2 Paper size

To change the paper size, there are two equal 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 `\score`, then the paper size will only apply to that score.

The following paper sizes are supported: `a6`, `a5`, `a4`, `a3`, `legal`, `letter`, `tabloid`.

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)
```

7.5.3 Page layout

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.

`firstpagenumber`
: The value of the page number of the first page. Default is 1.

`printfirstpagenumber`
: If set to true, will print the page number in the first page. Default is false.

`hsize`
: The width of the page.

`vsize`
: The height of the page.

`topmargin`
: Margin between header and top of the page.

`bottommargin`
: Margin between footer and bottom of the page.

`leftmargin`
: Margin between the left side of the page and the beginning of the music.

`linewidth`
: The length of the systems.

`headsep`
: Distance between the top-most music system and the page header.

`footsep`
: Distance between the bottom-most music system and the page footer.

`raggedbottom`
: If set to true, systems will not be spread across the page.

  This should be set false for pieces that have only two or three systems per page, for example orchestral scores.

`raggedlastbottom`
: If set to false, systems will be spread to fill the last page.

  Pieces that amply fill two pages or more should have this set to true.

`betweensystemspace`
: 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.

  Increasing this will provide a more even appearance of the page at the cost of using more vertical space.

`betweensystempadding`
: 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.

  Increasing this will put systems whose bounding boxes almost touch farther apart.

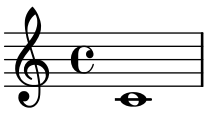
`aftertitlespace`
: Amount of space between the title and the first system.

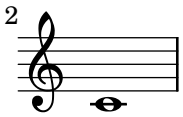
`beforetitlespace`
: Amount of space between the last system of the previous piece and the title of the next.

`betweentitlespace`
: Amount of space between consecutive titles (e.g., the title of the book and the title of a piece).

`systemSeparatorMarkup`
: This contains a markup object, which will be inserted between systems. This is often used for orchestral scores.

  The markup command `\slashSeparator` is provided as a sensible default, for example





Example:

```
\paper{
  hsize = 2\cm
  topmargin = 3\cm
  bottommargin = 3\cm
  raggedlastbottom = ##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's why the value has to be multiplied in the example

```
\paper {
  #(define bottommargin (* 2 cm))
```

```
}
```

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 "Engraved by LilyPond (*version*)".[3]

The header and footer are created by the functions `make-footer` and `make-header`, defined in `\paper`. The default implementations are in '`scm/page-layout.scm`'.

The following settings influence the header and footer layout.

`printpagenumber`
: this boolean controls whether a pagenumber is printed.

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 rightmargin is defined but doesn't set the right margin yet. The value for the right margin has to be defined adjusting the values of the leftmargin and linewidth.

The default page header puts the page number and the `instrument` field from the `\header` block on a line.

7.5.4 Vertical spacing

The height of each system is determined automatically. To prevent systems from bumping into each other, some minimum distances are set. By changing these, you can put staves closer together, and thus put more systems onto one page.

Normally staves are stacked vertically. To make staves maintain a distance, their vertical size is padded. This is done with the property `minimumVerticalExtent`. It takes a pair of numbers, so if you want to make it smaller than its default `#'(-4 . 4)`, then you could set

```
\set Staff.minimumVerticalExtent = #'(-3 . 3)
```

This sets the vertical size of the current staff to 3 staff spaces on either side of the center staff line. The argument of `minimumVerticalExtent` is interpreted as an interval, where the center line is the 0, so the first number is generally negative. The staff can be made larger at the bottom by setting it to `(-6 . 4)`.

To change the amount of space between systems, use `betweensystemspace`. A score with only one staff is still considered to have systems, so setting `betweensystemspace` will be much more useful than changing `minimumVerticalExtent`.

```
\layout {
  betweensystemspace = 10\mm
}
```

See also

Internals: Vertical alignment of staves is handled by the `VerticalAlignment` object.

Bugs

`minimumVerticalExtent` is syntactic sugar for setting `minimum-Y-extent` of the `VerticalAxisGroup` of the current context. It can only be changed score wide.

[3] Nicely printed parts are good PR for us, so please leave the tagline if you can.

7.5.5 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`.

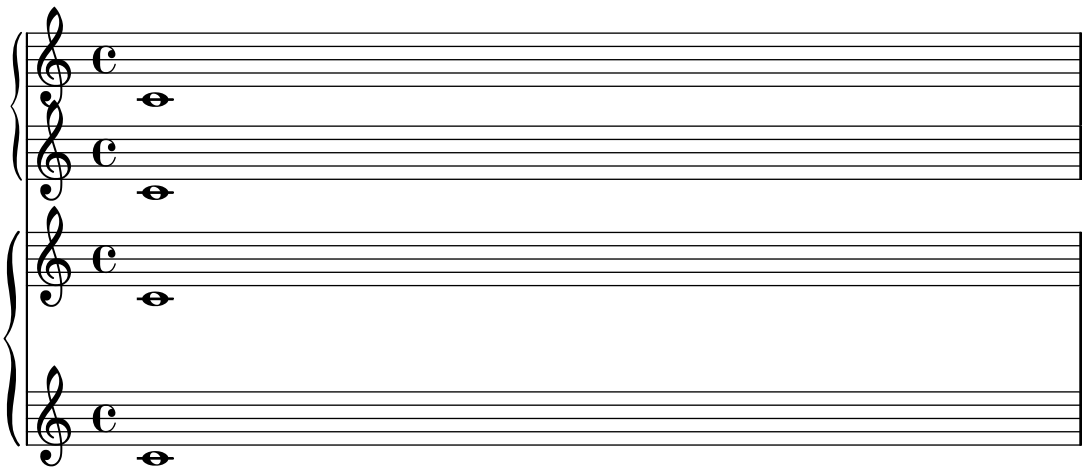
It can be adjusted as follows

```
\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 <<
  \new PianoStaff \with {
    \override VerticalAlignment #'forced-distance = #7
  } <<
    \new Staff { c1 }
    \new Staff { c }
  >>
  \new PianoStaff <<
    \new Staff { c }
    \new Staff { c }
  >>
>>
```



Bugs

`forced-distance` cannot be changed per system.

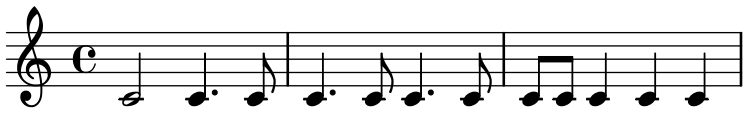
7.5.6 Horizontal Spacing

The spacing engine translates differences in durations into stretchable distances ("springs") of differring 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.

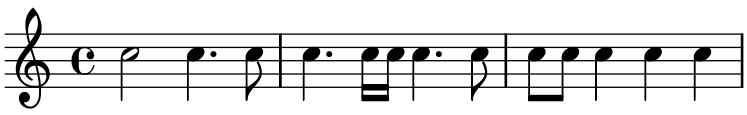
If one would follow the above procedure exactly, then adding a single 32nd note to a score that uses 8th and 16th notes, would widen up the entire score a lot. The shortest note is no longer a 16th, but a 32nd, thus adding 1 NHW to every note. To prevent this, the shortest duration for spacing is not 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.

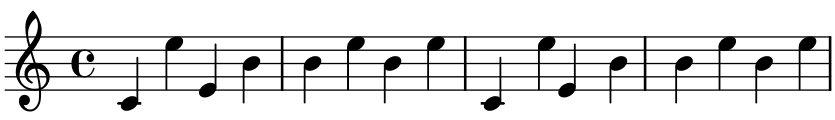
These durations may also be customized. If you set the `common-shortest-duration` in `SpacingSpanner`, then this sets the base duration for spacing. The maximum duration for this base (normally an 8th), is set through `base-shortest-duration`.

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:



See also

Internals: `SpacingSpanner`, `NoteSpacing`, `StaffSpacing`, `SeparationItem`, and `SeparatingGroupSpanner`.

Bugs

Spacing is determined on a score wide basis. If you have a score that changes its character (measured in durations) halfway during the score, the part containing the longer durations will be spaced too widely.

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.

7.5.7 Line length

The most basic settings influencing the spacing are `indent` and `linewidth`. They are set in the `\layout` block. They control the indentation of the first line of music, and the lengths of the lines.

If `raggedright` is set to true in the `\layout` block, then the lines are justified at their natural length. This is useful for short fragments, and for checking how tight the natural spacing is.

The option `raggedlast` is similar to `raggedright`, 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 length.

7.5.8 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: `BreakEvent`.

7.5.9 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 in the `\paper` block.

Predefined commands

`\pageBreak` `\noPageBreak`

7.5.10 Multiple movements

A document may contain multiple pieces of music. Examples of these are an etude book, or an orchestral part with multiple movements. Each movement is entered with a `\score` block,

```
\score {
  ..music..
}
```

The movements are combined together in a `\book` block, like

```
\book {
  \score {
    ..
  }
  \score {
    ..
  }
}
```

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.

```
\book {
  \header {
    title = "Eight miniatures"
    composer = "Igor Stravinsky"
  }
  \score {
    ...
    \header { piece = "Romanze" }
  }
  \score {
    ...
    \header { piece = "Menuetto" }
  }
}
```

7.5.11 Creating titles

Titles are created for each `\score` block, and over a `\book`.

The contents of the titles are taken from the `\header` blocks. The header block for a book supports the following

`title` The title of the music. Centered on top of the first page.

`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.

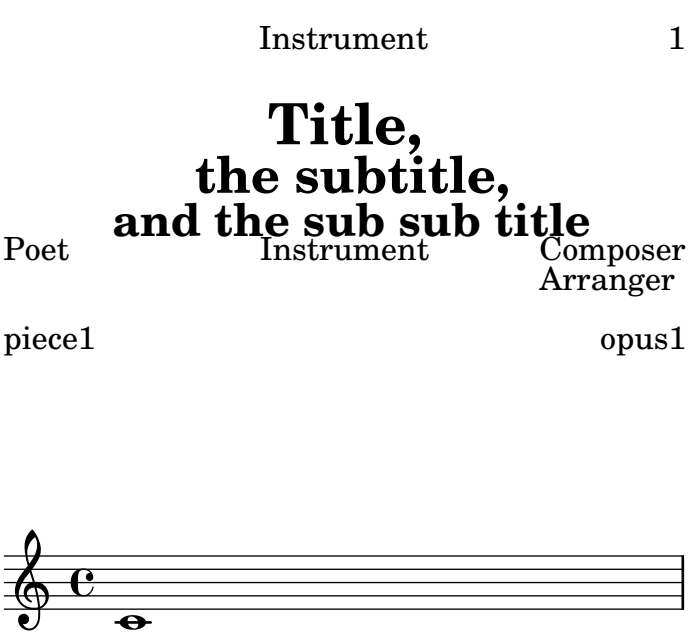
`dedication`
To whom the piece is dedicated.

`piece` Name of the piece, flush-left below the instrument.

`breakbefore`
This forces the title to start on a new page.

Here is a demonstration of the fields available,

```
\paper {
  linewidth = 9.0\cm
  vsize = 10.0\cm
}

\book {
  \header {
    title = "Title,"
    subtitle = "the subtitle,"
    subsubtitle = "and the sub sub title"
    poet = "Poet"
    composer = "Composer"
    texttranslator = "Text Translator"
    meter = "Meter"
    arranger = "Arranger"
    instrument = "Instrument"
    piece = "Piece"
  }

  \score {
    \header {
      piece = "piece1"
      opus = "opus1"
    }
    { c'1 }
  }
  \score {
    \header {
      piece = "piece2"
      opus = "opus2"
    }
    { c'1 }
  }
}
```



Different fonts may be selected for each element by using `\markup`, e.g.,

```
\header {
  title = \markup { \italic { The italic title } }
}
```

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 put over an entire `\book` block. Typically, it has the composer and the title of the piece

`scoreTitleMarkup`
: This is the title put over a `\score` block within a `\book`. 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
   @}
  }
}
```

7.6 File structure

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. This section enumerates them all.

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 `\header` block. This sets the global header block. This is the block containing the definitions for book-wide settings, like composer, title, etc.
- An `\addquote` statement. See Section 5.15.12 [Quoting other voices], page 144 for more information.
- 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`'.
- A `\book` block logically combines multiple movements (i.e., multiple `\score` blocks) in one document. A number of `\scores` creates a single output file, where all movement are concatenated.

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.

  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`'.

- An indentifier, such as

  ```
  foo = { c4 d e d }
  ```

  This can be used later on in the file by entering `\foo`.

The following example shows three things that may be entered at toplevel

```
\layout {
  % movements are non-justified by default
  raggedright = ##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`
- `\renameinput`

8 Interfaces for programmers

8.1 Programmer interfaces for input

8.1.1 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 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 }

#(define newLa (map ly:music-deep-copy
  (list traLaLa traLaLa)))
#(define twice
  (make-sequential-music newLa))

{ \twice }
```

In the above example, music expressions can be 'exported' 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 newLa)) }
```

Bugs

Mixing Scheme and LilyPond identifiers is not possible with the `--safe` option.

8.1.2 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 internals 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 internals manual, under `Music classes`.

- C++ object: Each music object is represented by a C++ object. For technical reasons, different music objects may be represented by different C++ object types. For example, a note is `Event` object, while `\grace` creates a `Grace_music` object.

  We expect that distinctions between different C++ types will disappear in the future.

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`.

8.1.3 Extending music syntax

The syntax of composite music expressions, like `\repeat`, `\transpose`, and `\context` follows the general form of

```
\keyword non-music-arguments music-arguments
```

Such syntax can also be defined as user code. To do this, it is necessary to create a *music function*. This is a specially marked Scheme function. For example, the music function `\applymusic` applies a user-defined function to a music expression. Its syntax is

```
\applymusic #func music
```

A music function is created with `ly:make-music-function`,

```
(ly:make-music-function
```

`\applymusic` takes a Scheme function and a Music expression as arguments. This is encoded in its parameter list,

```
(list procedure? ly:music?)
```

The function itself takes another argument, an Input location object. That object is used to provide error messages with file names and line numbers. The definition is the second argument of `ly:make-music-function`. The body simply calls the function

```
(lambda (where func music)
 (func music))
```

The above Scheme code only defines the functionality. The tag `\applymusic` is selected by defining

```
applymusic = #(ly:make-music-function
                (list procedure? ly:music?)
                (lambda (parser location func music)
                  (func music)))
```

A `def-music-function` macro is introduced on top of `ly:make-music-function` to ease the definition of music functions:

```
applymusic = #(def-music-function (parser location func music)
                (procedure? ly:music?)
                (func music))
```

Examples of the use of `\applymusic` are in the next section.

See also

'ly/music-functions-init.ly'.

8.1.4 Manipulating music expressions

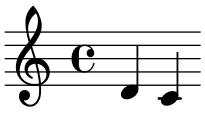
Music objects and their properties can be accessed and manipulated directly, through the \applymusic mechanism. The syntax for \applymusic is

```
\applymusic #func music
```

This means that the Scheme function *func* is called with *music* as its argument. The return value of *func* is the result of the entire expression. *func* may read and write music properties using the functions ly:music-property and ly:music-set-property!.

An example is a function that reverses the order of elements in its argument,

```
#(define (rev-music-1 m)
  (ly:music-set-property! m 'elements
    (reverse (ly:music-property m 'elements)))
  m)

\applymusic #rev-music-1 { c'4 d'4 }
```



The use of such a function is very limited. The effect of this function is void when applied to an argument that does not have multiple children. The following function application has no effect

```
\applymusic #rev-music-1 \grace { c4 d4 }
```

In this case, \grace is stored as GraceMusic, which has no elements, only a single element. Every generally applicable function for \applymusic must – like music expressions themselves – be recursive.

The following example is such a recursive function: It first extracts the elements of an expression, reverses them and puts them back. Then it recurses, both on elements and element children.

```
#(define (reverse-music music)
  (let* ((elements (ly:music-property music 'elements))
         (child (ly:music-property music 'element))
         (reversed (reverse elements)))

    ; set children
    (ly:music-set-property! music 'elements reversed)

    ; recurse
    (if (ly:music? child) (reverse-music child))
    (map reverse-music reversed)

    music))
```

A slightly more elaborate example is in 'input/test/reverse-music.ly'.

Some of the input syntax is also implemented as recursive music functions. For example, the syntax for polyphony

```
<<a \\ b>>
```

is actually implemented as a recursive function that replaces the above by the internal equivalent of

```
<< \context Voice = "1" { \voiceOne a }
   \context Voice = "2" { \voiceTwo b } >>
```

Other applications of `\applymusic` are writing out repeats automatically (‘`input/test/unfold-all-repeats.ly`’), saving keystrokes (‘`input/test/music-box.ly`’) and exporting LilyPond input to other formats (‘`input/no-notation/to-xml.ly`’).

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`.

See also

‘`scm/music-functions.scm`’, ‘`scm/music-types.scm`’, ‘`input/test/add-staccato.ly`’, ‘`input/test/unfold-all-repeats.ly`’, and ‘`input/test/music-box.ly`’.

8.1.5 Using LilyPond syntax inside Scheme

Creating music expressions in Scheme can be tedious, as they are heavily nested and the resulting Scheme code is large. For some simple tasks, this can be avoided, using common LilyPond syntax inside Scheme, with the dedicated `#{ ... #}` syntax.

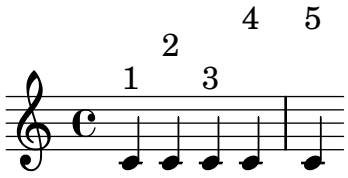
The following two expressions give equivalent music expressions:

```
mynotes = { \override Stem #'thickness = #4
            { c'8 d' } }

#(define mynotes #{ \override Stem #'thickness = #4
                    { c'8 d' } #})
```

The content of `#{ ... #}` is enclosed in an implicit `{ ... }` block, which is parsed. The resulting music expression, a `SequentialMusic` music object, is then returned and usable in Scheme.

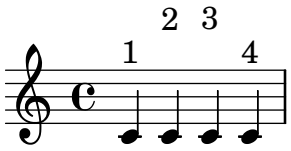
Arbitrary Scheme forms, including variables, can be used in `#{ ... #}` expressions with the `$` character (`$$` can be used to produce a single `$` character). This makes the creation of simple functions straightforward. In the following example, a function setting the TextScript's padding is defined:

```
#(use-modules (ice-9 optargs))
#(define* (textpad padding #:optional once?)
  (ly:export   ; this is necessary for using the expression
               ; directly inside a block
    (if once?
        #{ \once \override TextScript #'padding = #$padding #}
        #{ \override TextScript #'padding = #$padding #})))

{
  c'^"1"
  #(textpad 3.0 #t) % only once
  c'^"2"
  c'^"3"
  #(textpad 5.0)
  c'^"4"
  c'^"5"
}
```



Here, the variable `padding` is a number; music expression variables may also be used in a similar fashion, as in the following example:

```
#(define (with-padding padding)
  (lambda (music)
   #{ \override TextScript #'padding = #$padding
      $music
      \revert TextScript #'padding #}))

{
  c'^"1"
  \applymusic #(with-padding 3) { c'^"2" c'^"3" }
  c'^"4"
}
```

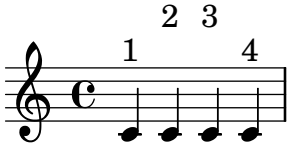


The function created by `(with-padding 3)` adds `\override` and `\revert` statements around the music given as an argument, and returns this new expression. Thus, this example is equivalent to:

```
{
  c'^"1"
  { \override TextScript #'padding = #3
    { c'^"2" c'^"3"}
    \revert TextScript #'padding
  }
  c'^"4"
}
```

This function may also be defined as a music function:

```
withPadding =
  #(def-music-function (parser location padding music) (number? ly:music?)
    #{ \override TextScript #'padding = #$padding
       $music
       \revert TextScript #'padding #})

{
  c'^"1"
  \withPadding #3 { c'^"2" c'^"3"}
  c'^"4"
}
```



8.2 Markup programmer interface

Markups are implemented as special Scheme functions. When applied with as arguments an output definition (`\layout` or `\paper`), and a list of properties and other arguments, produce a Stencil object.

8.2.1 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 < { \bold \italic "hello" \raise #0.4 "world" }
                  \bigger { foo bar baz } >
```

This example exposes the main translation rules between regular LilyPond markup syntax and Scheme markup syntax, which are summed up is this table:

LilyPond	**Scheme**
`\command`	`#:command`
`\variable`	`variable`
`{ ... }`	`#:line ( ... )`
`\center-align < ... >`	`#:center ( ... )`
`string`	`"string"`
`#scheme-arg`	`scheme-arg`

Besides, the whole scheme language is accessible inside the `markup` macro: thus, one may use function calls inside `markup` in order to manipulate character strings for instance. This proves useful when defining new markup commands (see Section 8.2.3 [Markup command definition], page 215).

Bugs

One can not feed the `#:line` (resp `#:center`, `#:column`) command with a variable or the result of a function call. Example:

```
(markup #:line (fun-that-returns-markups))
```

is invalid. One should use the `make-line-markup` (resp., `make-center-markup` or `make-column-markup`) function instead,

```
(markup (make-line-markup (fun-that-returns-markups)))
```

8.2.2 How markups work internally

In a markup like

```
\raise #0.5 "foo"
```

`\raise` is actually represented by the `raise-markup` function. The markup expression is stored as

```
(list raise-markup 0.5 (list simple-markup 'latin1 "foo"))
```

In this case, `latin1` is the input encoding, which is set with the `\encoding` command.

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 "foo" markup)
```

The `raise-markup` first creates the stencil for the `foo` 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`'.

8.2.3 Markup command definition

New markup commands can be defined with the `def-markup-command` scheme macro.

```
(def-markup-command (command-name layout props arg1 arg2 ...)
            (arg1-type? arg2-type? ...)
  ..command body..)
```

The arguments signify

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 TEX's small caps font. Normally, we could select the small caps font as follows:

```
\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 have to define a function using `def-markup-command`. The command should take a single argument, of type markup. Therefore, the start of the definition should read

```
(def-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 translated a bit to the left and top. We will define a `\character` command that takes into account the necessary translation, and uses the newly defined `\smallcaps` command:

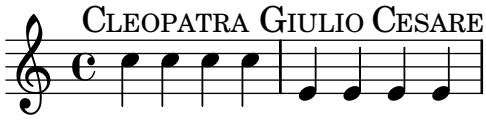
```
#(def-markup-command (character layout props name) (string?)
  "Print the character name in small caps, translated to the left and
  top.  Syntax: \\character #\"name\""
  (interpret-markup layout props
   (markup "" #:translate (cons -3 1) #:smallcaps name)))
```

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 (`""`) before the translated text. Now the `""` 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:

```
#(def-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.

8.3 Contexts for programmers

8.3.1 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)))
```

8.3.2 Running a function on all layout objects

The most versatile way of tuning an object is `\applyoutput`. Its syntax is

```
\applyoutput 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, 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)))
```

9 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 or Texinfo documents.

9.1 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, and Texinfo 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 lilybook.tex
lilypond-book (GNU LilyPond) 2.5.0
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 nice PDF document, run the following commands

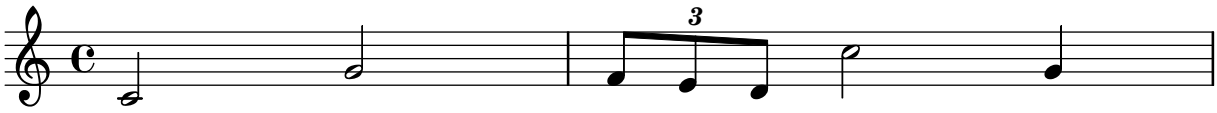
```
dvips -Ppdf -u+lilypond -u+ec-mftrace lilybook
ps2pdf lilybook.ps
```

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`'.

Finally the result of the LaTeX example shown above.[1] This finishes the tutorial section.

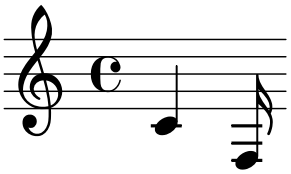
[1] This tutorial is processed with Texinfo, so the example gives slightly different results in layout.

Documents for `lilypond-book` may freely mix music and text. For example,



Options are put in brackets.

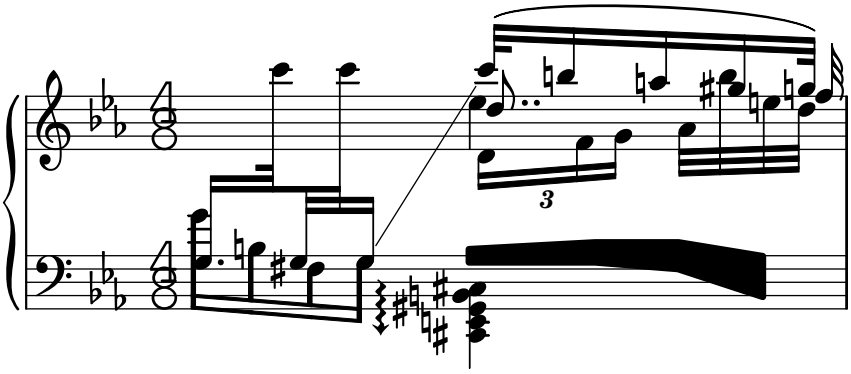
```
c’4 f16
```



Larger examples can be put into a separate file, and introduced with `\lilypondfile`.

Screech and boink
Random complex notation

Han-Wen Nienhuys



9.2 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* (`http://www.ctan.org/tex-archive/info/lshort/english/`) 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

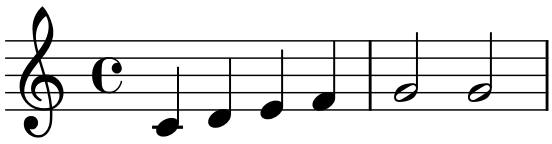
```
\lilypond{ YOUR LILYPOND CODE }
```

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 linewidth 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 `linewidth` music fragment option.

Each snippet calls `\preLilyPondExample` before and `\postLilyPondExample` after the music if those macros have been defined by the user.

For printing the LaTeX document you need a DVI to PostScript translator like `dvips`. For producing PostScript with scalable fonts, add the following options to the `dvips` command line:

```
-Ppdf -u+lilypond.map -u+ec-mftrace.map
```

PDF can then be produced with a PostScript to PDF translator like `ps2pdf` (which is part of GhostScript).

[THE FOLLOWING IS OUT OF DATE.]

LilyPond does not use the LaTeX font handling scheme for lyrics and text markups; it uses the EC font family and has limited support for selecting an input encoding with the `\encoding` keyword if the output is directly processed (these limitations primarily affect LilyPond's native PostScript output). With `lilypond-book`, the encoding issues are completely handled by the document which includes LilyPond snippets; `lilypond` outputs all text strings without modification. The drawback is that LilyPond always applies the EC font metrics to those strings for computing the locations within the music snippets; this often causes unpleasant horizontal (and vertical) shifts. With other words, support for encodings other than latin-1 is possible but usually yields badly positioned text. Future versions of LilyPond will fix this.

Since latin-1 is the default encoding for LilyPond markup and lyrics it is not necessary to explicitly add `\encoding "latin1"` to LilyPond snippets. You might also consider the use of `\encoding "TeX"` instead which basically makes LilyPond skip TeX commands (starting with a backslash) and braces in text strings – it is not recommended, though, since LilyPond gives only a rough approximation to the real string length.

As a corrolary of the last paragraphs the following two lines should be present in the LaTeX document preamble

[END OUT OF DATE]

```
\usepackage[latin1]{inputenc}
\usepackage[T1]{fontenc}
```

and real latin-1 characters should be used in LilyPond snippets; for example, use ß, not `\ss`.

9.3 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

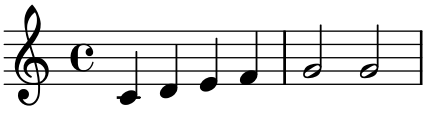
```
@lilypondfile[options,go,here]{filename}
```

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.

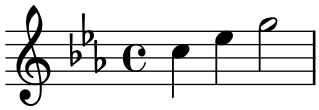
When using the Texinfo output format, `lilypond-book` also generates bitmaps of the music (in PNG format), so you can make an HTML document with embedded music.

9.4 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>
```

9.5 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.

`raggedright`
: Produce ragged-right lines with natural spacing (i.e., `raggedright = ##t` is added to the LilyPond snippet). This is the default for the `\lilypond{}` command if no `linewidth` 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.

`linewidth`
`linewidth=`*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 `linewidth` option is given, `lilypond-book` tries to guess a default for `lilypond` environments which don't use the `raggedright` option.

`notime`
: Do not print the time signature.

`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.

9.6 Invoking `lilypond-book`

`lilypond-book` produces a file with one of the following extensions: '`.tex`', '`.texi`', or '`.html`', depending on the output format. Both '`.tex`' and '`.texi`' files need further processing.

To produce PDF output from the '`.tex`' file, you should do

```
latex yourfile.tex
dvips -Ppdf -u+ec-mftrace.map -u+lilypond.map yourfile.dvi
ps2pdf yourfile.ps
```

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 section "Format with texi2dvi" in *GNU Texinfo*, and section "Creating an Info File" in *GNU Texinfo*.

`lilypond-book` accepts the following command line options:

`-f` *format*
`--format=`*format*
: Specify the document type to process: `html`, `latex`, or `texi` (the default). 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*.

  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
...
```

-P *process*
--process=*command*
: Process LilyPond snippets using *command*. The default command is lilypond.

-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.

The size of a music block is limited to 1.5 KB, due to technical problems with the Python regular expression engine. For longer files, use \lilypondfile.

9.7 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 9.6 [Invoking lilypond-book], page 225, 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
'.tely'	Texinfo
'.tex'	LaTeX
'.texi'	Texinfo
'.texinfo'	Texinfo
'.xml'	HTML

10 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 (`http://lilypond.org`) for more details.

10.1 Invoking `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

```
covert-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.

To upgrade LilyPond fragments in texinfo files, use

```
convert-ly --from=... --to=... --no-version *.itely
```

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.

`-o,--output=`*`file`*
: Set the output file to write.

`-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.

`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.

Bugs

Not all language changes are handled. Only one output option can be specified.

```
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' com
   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. Only very simple cases are fixed.
2.0->2.2:
 Doesn't handle \partcombine
 Doesn't do \addlyrics => \lyricsto, this breaks some scores with multiple stanzas.
2.2->2.4:
 \turnOff (used in \set Staff.VoltaBracket = \turnOff) is not properly converted.
```

10.2 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
```

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.

10.3 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
```

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.

10.4 Invoking `abc2ly`

ABC is a fairly simple ASCII based format. It is described at the ABC site:

`http://www.gre.ac.uk/~c.walshaw/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.

10.5 Invoking `mup2ly`

Mup (Music Publisher) is a shareware music notation program by Arkkra Enterprises. `mup2ly` will convert part of a Mup file to LilyPond format. It is invoked from the command-line as follows:

`mup2ly [`*option*`]...` *mup-file*

The following options are supported by `mup2ly`:

`-d,--debug`
: show what constructs are not converted, but skipped.

`-D, --define=`*name*`[=exp]`
: define macro *name* with opt expansion `exp`

`-E,--pre-process`
: only run the pre-processor

`-h,--help`
: print help

`-o,--output=`*file*
: write output to *file*

`-v,--version`
: version information

`-w,--warranty`
: print warranty and copyright.

Bugs

Only plain notes (pitches, durations), voices, and staves are converted.

10.6 Other formats

LilyPond itself does not come with support for other formats, but there are some external tools that also generate LilyPond files.

These tools include

- Denemo (`http://denemo.sourceforge.net/`).
- xml2ly (`http://www.nongnu.org/xml2ly/`), that imports MusicXML (`http://www.musicxml.com/`)
- NoteEdit (`http://rnvs.informatik.tu-chemnitz.de/~jan/noteedit/noteedit.html`) which imports MusicXML
- Rosegarden (`http://www.all-day-breakfast.com/rosegarden/`), which imports MIDI

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.[1]

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

[1] If you want to know more about Scheme, see `http://www.schemers.org`.

```
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 (* 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. 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[2] 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. This 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,

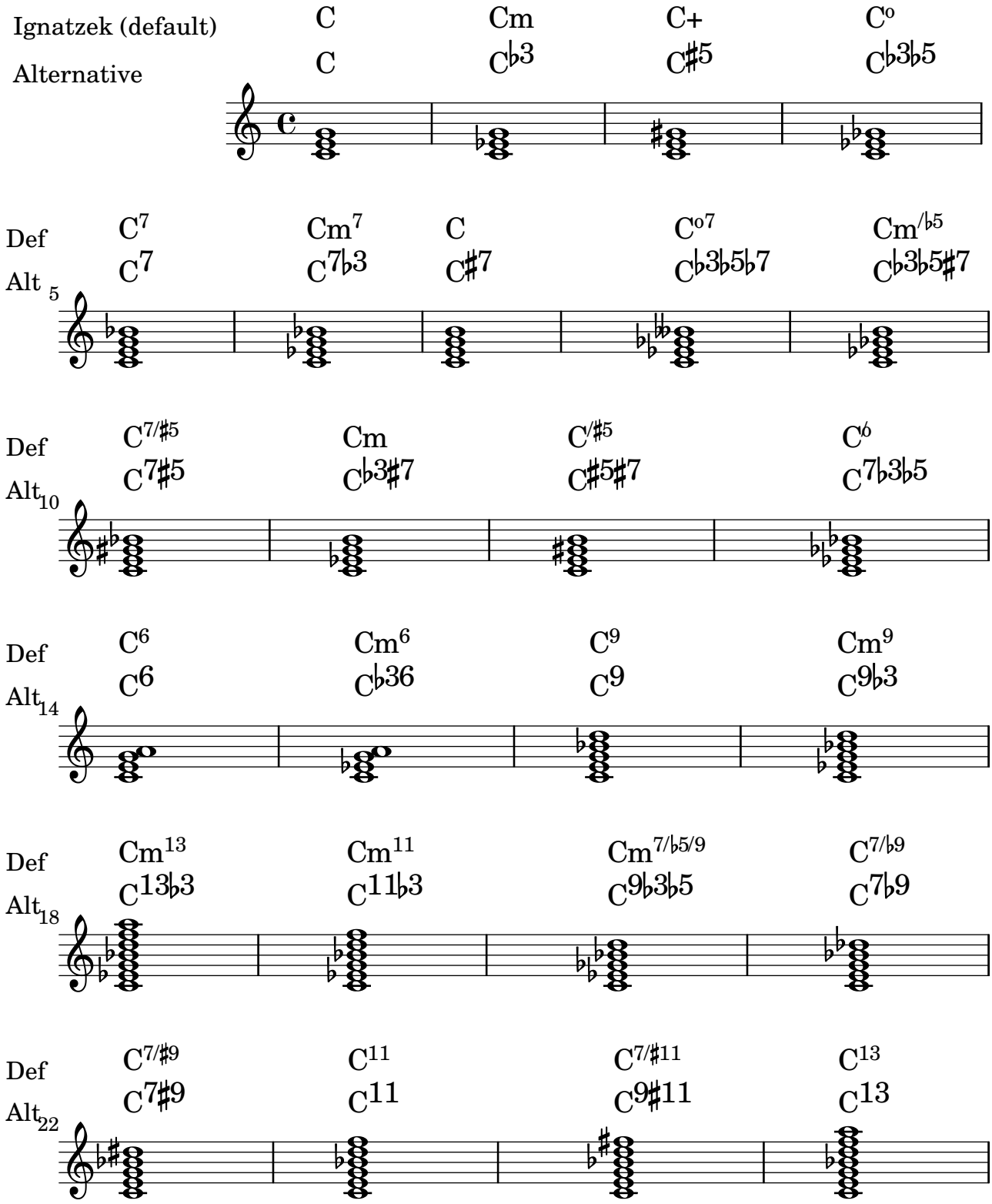
```
#'(stem . head)
#'(staff clef key-signature)
#'((1) (2))
```

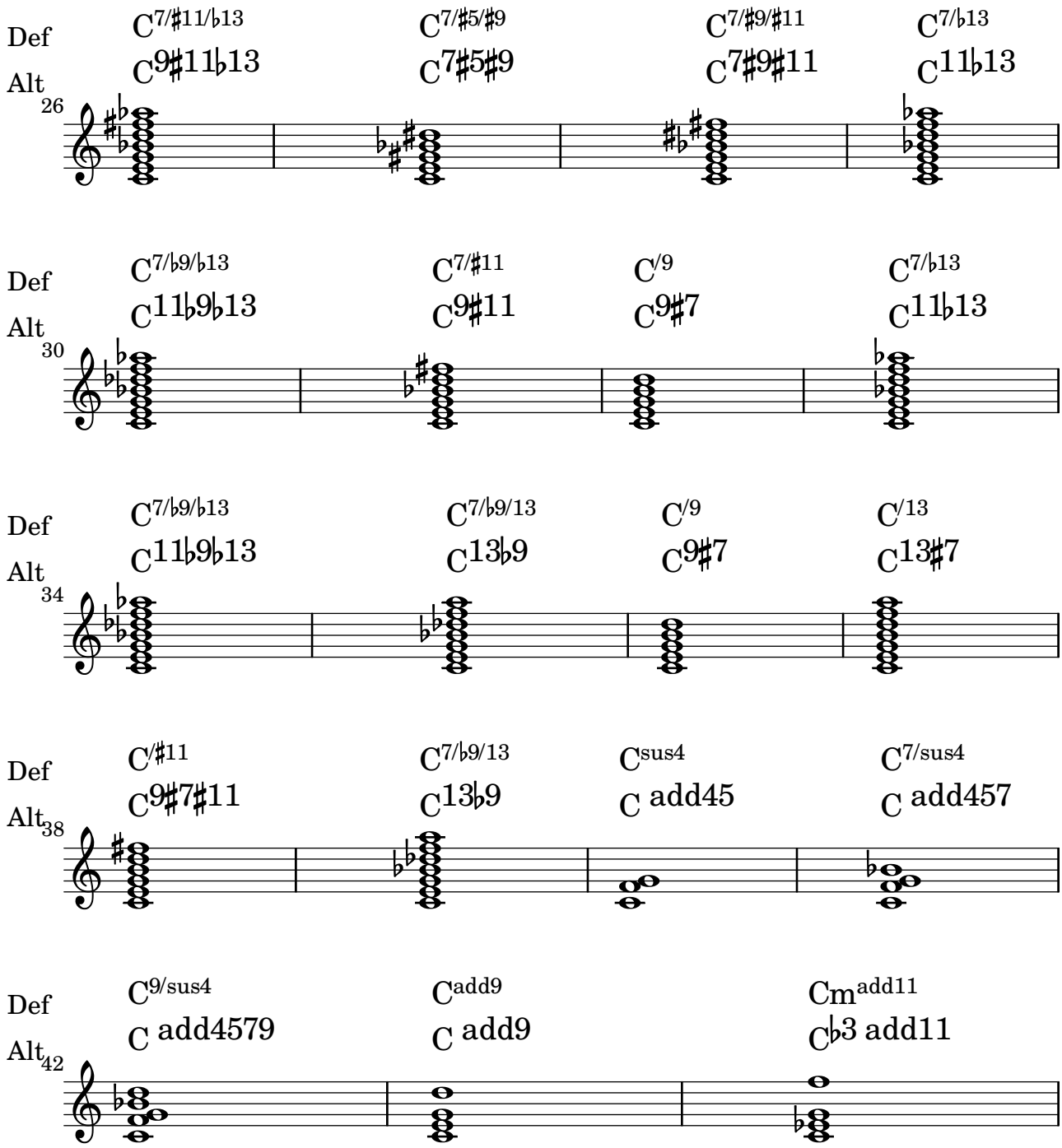
[2] In Scheme terminology, the pair is called `cons`, and its two elements are called `car` and `cdr` respectively.

Appendix C Notation manual details

C.1 Chord name chart

The following charts shows two standard systems for printing chord names, along with the pitches they represent.





C.2 MIDI instruments

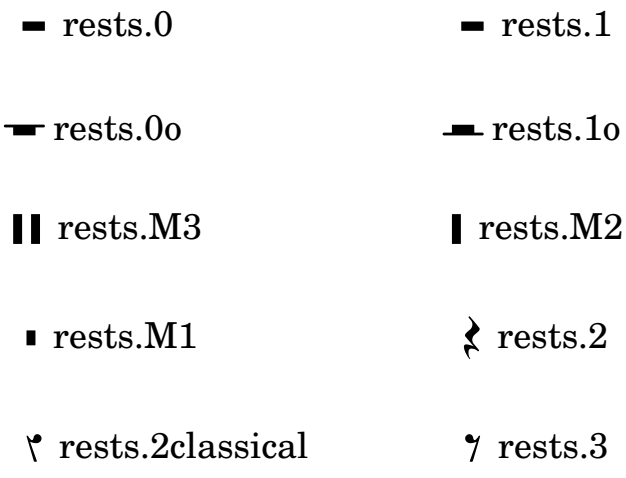
The following is a list of names that can be used for the `midiInstrument` property.

```
acoustic grand          contrabass          lead 7 (fifths)
bright acoustic         tremolo strings     lead 8 (bass+lead)
electric grand          pizzicato strings   pad 1 (new age)
honky-tonk              orchestral strings  pad 2 (warm)
electric piano 1        timpani             pad 3 (polysynth)
electric piano 2        string ensemble 1   pad 4 (choir)
harpsichord             string ensemble 2   pad 5 (bowed)
clav                    synthstrings 1      pad 6 (metallic)
celesta                 synthstrings 2      pad 7 (halo)
glockenspiel            choir aahs          pad 8 (sweep)
music box               voice oohs          fx 1 (rain)
```

```
vibraphone                synth voice          fx 2 (soundtrack)
marimba                   orchestra hit        fx 3 (crystal)
xylophone                 trumpet              fx 4 (atmosphere)
tubular bells             trombone             fx 5 (brightness)
dulcimer                  tuba                 fx 6 (goblins)
drawbar organ             muted trumpet        fx 7 (echoes)
percussive organ          french horn          fx 8 (sci-fi)
rock organ                brass section        sitar
church organ              synthbrass 1         banjo
reed organ                synthbrass 2         shamisen
accordion                 soprano sax          koto
harmonica                 alto sax             kalimba
concertina                tenor sax            bagpipe
acoustic guitar (nylon)   baritone sax         fiddle
acoustic guitar (steel)   oboe                 shanai
electric guitar (jazz)    english horn         tinkle bell
electric guitar (clean)   bassoon              agogo
electric guitar (muted)   clarinet             steel drums
overdriven guitar         piccolo              woodblock
distorted guitar          flute                taiko drum
guitar harmonics          recorder             melodic tom
acoustic bass             pan flute            synth drum
electric bass (finger)    blown bottle         reverse cymbal
electric bass (pick)      shakuhachi           guitar fret noise
fretless bass             whistle              breath noise
slap bass 1               ocarina              seashore
slap bass 2               lead 1 (square)      bird tweet
synth bass 1              lead 2 (sawtooth)    telephone ring
synth bass 2              lead 3 (calliope)    helicopter
violin                    lead 4 (chiff)       applause
viola                     lead 5 (charang)     gunshot
cello                     lead 6 (voice)
```

C.3 The Feta font

The following symbols are available in the Feta font and may be accessed directly using text markup such as `g^\markup { \musicglyph #"scripts-segno" }`, see Section 7.4 [Text markup], page 188.



 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

 dots.dot

 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.s1do

noteheads.d2do noteheads.s2do

noteheads.s0re noteheads.u1re

noteheads.d1re noteheads.u2re

noteheads.d2re noteheads.s0mi

noteheads.s1mi noteheads.s2mi

noteheads.d0fa noteheads.u0fa

noteheads.d1fa noteheads.u1fa

noteheads.u2fa noteheads.d2fa

noteheads.s0la noteheads.s1la

noteheads.s2la noteheads.s0ti

noteheads.u1ti noteheads.d1ti

noteheads.u2ti noteheads.d2ti

scripts.ufermata scripts.dfermata

scripts.ushortfermata scripts.dshortfermata

scripts.ulongfermata scripts.dlongfermata

scripts.uverylongfermata scripts.dverylongfermata

scripts.thumb scripts.sforzato

<> scripts.espr scripts.staccato

scripts.ustaccatissimo scripts.dstaccatissimo

scripts.tenuto scripts.uportato

scripts.dportato scripts.umarcato

scripts.dmarcato scripts.open

+ scripts.stopped scripts.upbow

scripts.downbow scripts.reverseturn

scripts.turn tr scripts.trill

scripts.upedalheel scripts.dpedalheel

scripts.upedaltoe scripts.dpedaltoe

scripts.flageolet scripts.segno

scripts.coda scripts.varcoda

scripts.rcomma scripts.lcomma

scripts.rvarcomma scripts.lvarcomma

scripts.arpeggio scripts.trill_element

scripts.arpeggio.arrow.M1 scripts.arpeggio.arrow.1

scripts.trilelement scripts.prall

 scripts.mordent

 scripts.prallprall

 scripts.prallmordent

 scripts.upprall

 scripts.upmordent

 scripts.pralldown

 scripts.downprall

 scripts.downmordent

 scripts.prallup

 scripts.lineprall

 scripts.caesura

 flags.u3

 flags.u4

 flags.u5

 flags.u6

 flags.d3

 flags.ugrace

 flags.dgrace

 flags.d4

 flags.d5

 flags.d6

 clefs.C

 clefs.C_change

 clefs.F

 clefs.F_change

 clefs.G

 clefs.G_change

 clefs.percussion

 clefs.percussion_change

 clefs.tab

 clefs.tab_change

 timesig.C44

 timesig.C22

 pedal.*

 pedal.M

 pedal..

 pedal.P

 pedal.d

 pedal.e

 pedal.Ped

 accordion.accDiscant

 accordion.accDot

 accordion.accFreebase

 accordion.accStdbase

 accordion.accBayanbase

 accordion.accOldEE

 rests.M3neomensural

 rests.M2neomensural

 rests.M1neomensural

 rests.0neomensural

 rests.1neomensural

 rests.2neomensural

 rests.3neomensural

 rests.4neomensural

 rests.M3mensural

 rests.M2mensural

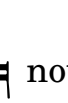
 rests.M1mensural

 rests.0mensural

 rests.1mensural

 rests.2mensural

 rests.3mensural

 rests.4mensural

 noteheads.lneomensural

 noteheads.sM3neomensural

noteheads.sM2neomensural noteheads.sM1neomensural

noteheads.s0neomensural noteheads.0harmonic

noteheads.s1neomensural noteheads.s2neomensural

noteheads.slmensural noteheads.sM3mensural

noteheads.sM2mensural noteheads.sM1mensural

noteheads.s0mensural noteheads.s1mensural

noteheads.s2mensural noteheads.svaticana.punctum

noteheads.svaticana.punctum.cavum

noteheads.svaticana.linea.punctum

noteheads.svaticana.linea.punctum.cavum

noteheads.svaticana.inclinatum

noteheads.svaticana.lpes noteheads.svaticana.vlpes

noteheads.svaticana.upes noteheads.svaticana.vupes

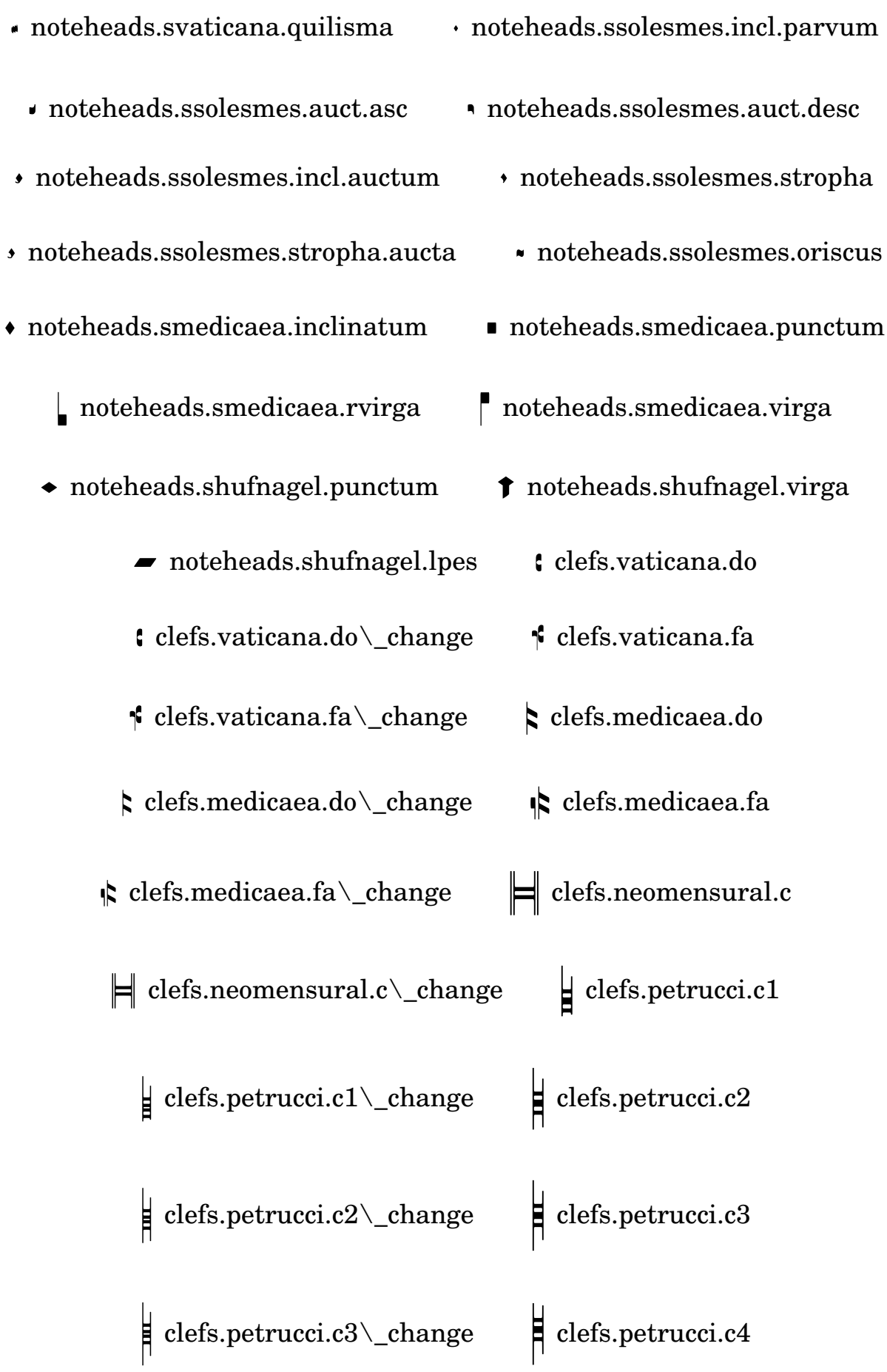
noteheads.svaticana.plica noteheads.svaticana.epiphonus

noteheads.svaticana.vepiphonus

noteheads.svaticana.reverse.plica

noteheads.svaticana.inner.cephalicus

noteheads.svaticana.cephalicus


noteheads.svaticana.quilisma
noteheads.ssolesmes.incl.parvum
noteheads.ssolesmes.auct.asc
noteheads.ssolesmes.auct.desc
noteheads.ssolesmes.incl.auctum
noteheads.ssolesmes.stropha
noteheads.ssolesmes.stropha.aucta
noteheads.ssolesmes.oriscus
noteheads.smedicaea.inclinatum
noteheads.smedicaea.punctum
noteheads.smedicaea.rvirga
noteheads.smedicaea.virga
noteheads.shufnagel.punctum
noteheads.shufnagel.virga
noteheads.shufnagel.lpes
clefs.vaticana.do
clefs.vaticana.do_change
clefs.vaticana.fa
clefs.vaticana.fa_change
clefs.medicaea.do
clefs.medicaea.do_change
clefs.medicaea.fa
clefs.medicaea.fa_change
clefs.neomensural.c
clefs.neomensural.c_change
clefs.petrucci.c1
clefs.petrucci.c1_change
clefs.petrucci.c2
clefs.petrucci.c2_change
clefs.petrucci.c3
clefs.petrucci.c3_change
clefs.petrucci.c4

 clefs.petrucci.c4_change

 clefs.petrucci.c5

 clefs.petrucci.c5_change

 clefs.mensural.c

 clefs.mensural.c_change

 clefs.petrucci.f

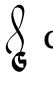
 clefs.petrucci.f_change

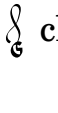
 clefs.mensural.f

 clefs.mensural.f_change

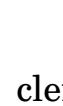
 clefs.petrucci.g

 clefs.petrucci.g_change

 clefs.mensural.g

 clefs.mensural.g_change

 clefs.hufnagel.do

 clefs.hufnagel.do_change

 clefs.hufnagel.fa

 clefs.hufnagel.fa_change

 clefs.hufnagel.do.fa

 clefs.hufnagel.do.fa_change

 custodes.hufnagel.u0

 custodes.hufnagel.u1

 custodes.hufnagel.u2

 custodes.hufnagel.d0

 custodes.hufnagel.d1

 custodes.hufnagel.d2

 custodes.medicaea.u0

 custodes.medicaea.u1

 custodes.medicaea.u2

 custodes.medicaea.d0

 custodes.medicaea.d1

 custodes.medicaea.d2

 custodes.vaticana.u0

 custodes.vaticana.u1

 custodes.vaticana.u2

 custodes.vaticana.d0

 custodes.vaticana.d1

 custodes.vaticana.d2

 custodes.mensural.u0

 custodes.mensural.u1

 custodes.mensural.u2

 custodes.mensural.d0

 custodes.mensural.d1

 custodes.mensural.d2

 accidentals.medicaeaM1

 accidentals.vaticanaM1

 accidentals.vaticana0

 accidentals.mensural1

 accidentals.mensuralM1

 accidentals.hufnagelM1

 flags.mensuralu03

 flags.mensuralu13

 flags.mensuralu23

 flags.mensurald03

 flags.mensurald13

 flags.mensurald23

 flags.mensuralu04

 flags.mensuralu14

 flags.mensuralu24

 flags.mensurald04

 flags.mensurald14

 flags.mensurald24

 flags.mensuralu05

 flags.mensuralu15

 flags.mensuralu25

 flags.mensurald05

 flags.mensurald15

 flags.mensurald25

 flags.mensuralu06

 flags.mensuralu16

 flags.mensuralu26

 flags.mensurald06

 flags.mensurald16

 flags.mensurald26

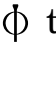
 timesig.mensural44

 timesig.mensural22

 timesig.mensural32

 timesig.mensural64

 timesig.mensural94

 timesig.mensural34

 timesig.mensural68

 timesig.mensural98

 timesig.mensural48

 timesig.mensural68alt

 timesig.mensural24

 timesig.neomensural44

 timesig.neomensural22

 timesig.neomensural32

 timesig.neomensural64

 timesig.neomensural94

 timesig.neomensural34

 timesig.neomensural68

 timesig.neomensural98

 timesig.neomensural48

 timesig.neomensural68alt

 timesig.neomensural24

 scripts.ictus

scripts.uaccentus scripts.daccentus

scripts.usemicirculus scripts.dsemicirculus

scripts.circulus scripts.augmentum

scripts.usignumcongruentiae scripts.dsignumcongruentiae

Appendix D Point and click

Point and click lets you find notes in the input by clicking on them in the Xdvi window. This makes it easier to find input that causes some error in the sheet music.

To use it, you need the following software:

- a dvi viewer that supports src specials.

  The most obvious choice is Xdvi[1], version 22.36 or newer. It is available from ftp.math.berkeley.edu (`ftp://ftp.math.berkeley.edu/pub/Software/TeX/xdvi.tar.gz`).

  Most TEX distributions ship with `xdvik`, which is always a few versions behind the official Xdvi. To find out which Xdvi you are running, try `xdvi -version` or `xdvi.bin -version`.

- an editor with a client/server interface (or a lightweight GUI editor):
  - Emacs. Emacs is an extendable text editor. It is available from `http://www.gnu.org/software/emacs/`. You need version 21 to use column location.
  - XEmacs. XEmacs is very similar to Emacs.
  - NEdit. NEdit runs under Windows and Unix. It is available from `http://www.nedit.org`.
  - GVim. GVim is a GUI variant of VIM, the popular VI clone. It is available from `http://www.vim.org`.
  - jEdit. jEdit is an editor written in Java with extensive plug-in support. The LilyPond plugin for jEdit comes with an DVI viewer, which is preconfigured for point-and-click.

Xdvi must be configured to find the TEX fonts and music fonts. Refer to the Xdvi documentation for more information.

To use point-and-click, add one of these lines to the top of your '`.ly`' file:

```
#(ly:set-point-and-click 'line)
```

When viewing, Control-Mousebutton 1 will take you to the originating spot in the '`.ly`' file. Control-Mousebutton 2 will show all clickable boxes.

If you correct large files with point-and-click, be sure to start correcting at the end of the file. When you start at the top, and insert one line, all following locations will be off by a line.

For using point-and-click with Emacs, add the following In your Emacs startup file (usually '`~/.emacs`'):

```
(server-start)
```

Make sure that the environment variable *XEDITOR* is set to

```
emacsclient --no-wait +%l %f
```

If you use XEmacs instead of Emacs, insert `(gnuserve-start)` in your '`.emacs`' file, and set `XEDITOR` to `gnuclient -q +%l %f`.

For using Vim, set `XEDITOR` to `gvim --remote +%l %f`, or use this argument with Xdvi's `-editor` option.

For using NEdit, set `XEDITOR` to `nc -noask +%l %f`, or use this argument with Xdvi's `-editor` option.

It can also make your editor jump to the exact location of the note you clicked. This is only supported on Emacs and VIM. Users of Emacs version 20 must apply the patch '`emacsclient.patch`'. Users of version 21 must apply '`server.el.patch`' (version 21.2 and earlier). At the top of the `.ly` file, replace the `set-point-and-click` line with the following line:

[1] KDVI also provides src specials, but does not use the kpathsea library, so it cannot find LilyPond font and PostScript library files.

```
#(ly:set-point-and-click 'line-column)
```

and set `XEDITOR` to `emacsclient --no-wait +%l:%c %f`. Vim users can set `XEDITOR` to `gvim --remote +:%l:norm%c| %f`.

Appendix E 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.

E.0.1 ADDENDUM: How to use this License for your documents

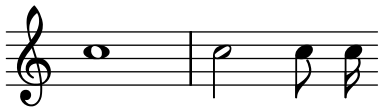
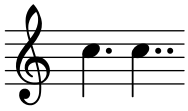
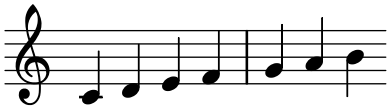
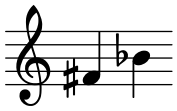
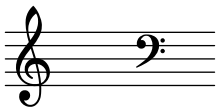
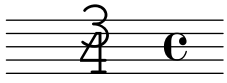
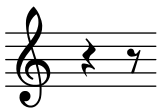
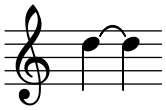
To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
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''.
```

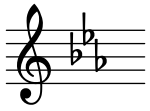
If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

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.

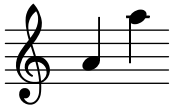
Appendix F Cheat sheet

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	

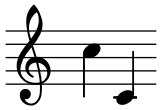
`\key es \major` — key signature



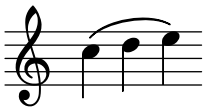
note`'` — raise octave



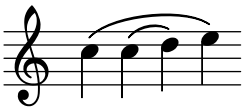
note`,` — lower octave



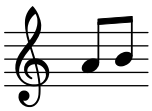
`c( d e)` — slur



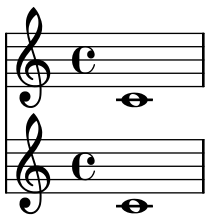
`c\( c( d) e\)` — phrasing slur



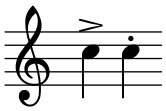
`a8[ b]` — beam



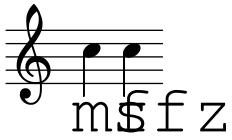
`<< \new Staff ... >>` — more staves



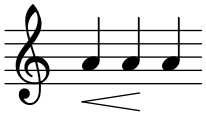
`c-> c-.` — articulations



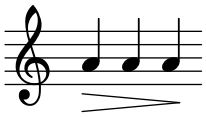
`c\mf c\sfz` dynamics



`a\< a \!a` crescendo

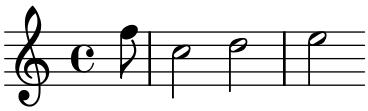


`a\> a a\!` decrescendo

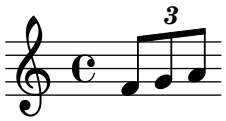


`< >` chord

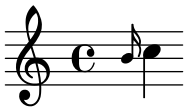
`\partial 8` upstep



`\times 2/3 {f g a}` triplets



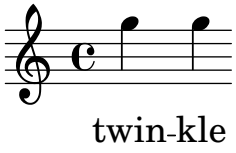
`\grace` grace notes



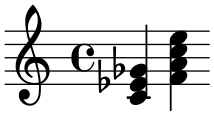
`\lyricmode { twinkle }` entering lyrics
`\new Lyrics` printing lyrics twinkle

twinkle

`twin -- kle` lyric hyphen

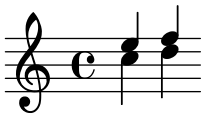


`\chordmode { c:dim f:maj7 }` chords



`\context ChordNames` printing chord names

`<<{e f} \\{c d}>>` polyphony



`s4 s8 s16` spacer rests

Appendix G Unified index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X