GNU LilyPond

The music typesetter

Han-Wen Nienhuys
Jan Nieuwenhuizen
Jürgen Reuter
Rune Zedeler

(For LilyPond version 2.2.6)

Table of Contents

Preface **1**
Notes to version 2.2 1

1 Introduction **2**
1.1 Engraving 2
1.2 Automated engraving 3
1.3 What symbols to engrave? 4
1.4 Music representation 6
1.5 Example applications 7
1.6 About this manual 8

2 Tutorial **10**
2.1 First steps 10
2.2 Running LilyPond 12
2.3 More about pitches 12
2.4 Entering ties 14
2.5 Automatic and manual beams 14
2.6 Octave entry 15
2.7 Music expressions explained 16
2.8 More staves 18
2.9 Adding articulation marks to notes 19
2.10 Combining notes into chords 20
2.11 Basic rhythmical commands 21
2.12 Commenting input files 21
2.13 Printing lyrics 22
2.14 A lead sheet 23
2.15 Listening to output 24
2.16 Titling 24
2.17 Single staff polyphony 25
2.18 Piano staves 26
2.19 Organizing larger pieces 26
2.20 An orchestral part 27
2.21 Integrating text and music 28

3 Notation manual **31**
3.1 Note entry 31
3.1.1 Notes 31
3.1.2 Pitches 31
3.1.3 Chromatic alterations 32
3.1.4 Chords 33
3.1.5 Rests 33
3.1.6 Skips 33
3.1.7 Durations 34
3.1.8 Stems 35
3.1.9 Ties 35
3.1.10 Tuplets 36
3.2 Easier music entry 37

3.2.1 Relative octaves ... 37
3.2.2 Octave check ... 38
3.2.3 Bar check ... 38
3.2.4 Skipping corrected music ... 39
3.2.5 Automatic note splitting ... 39
3.3 Staff notation ... 40
3.3.1 Staff symbol ... 40
3.3.2 Key signature ... 40
3.3.3 Clef ... 40
3.3.4 Ottava brackets ... 41
3.3.5 Time signature ... 42
3.3.6 Partial measures ... 43
3.3.7 Unmetered music ... 43
3.3.8 Bar lines ... 43
3.4 Polyphony ... 45
3.5 Beaming ... 46
3.5.1 Manual beams ... 47
3.5.2 Setting automatic beam behavior ... 48
3.6 Accidentals ... 49
3.7 Expressive marks ... 51
3.7.1 Slurs ... 51
3.7.2 Phrasing slurs ... 52
3.7.3 Breath marks ... 52
3.7.4 Metronome marks ... 53
3.7.5 Text spanners ... 53
3.7.6 Analysis brackets ... 54
3.7.7 Articulations ... 54
3.7.8 Fingering instructions ... 56
3.7.9 Text scripts ... 57
3.7.10 Grace notes ... 57
3.7.11 Glissando ... 59
3.7.12 Dynamics ... 60
3.8 Repeats ... 61
3.8.1 Repeat syntax ... 61
3.8.2 Repeats and MIDI ... 62
3.8.3 Manual repeat commands ... 62
3.8.4 Tremolo repeats ... 63
3.8.5 Tremolo subdivisions ... 63
3.8.6 Measure repeats ... 64
3.9 Rhythmic music ... 64
3.9.1 Showing melody rhythms ... 64
3.9.2 Entering percussion ... 64
3.9.3 Percussion staves ... 65
3.10 Piano music ... 67
3.10.1 Automatic staff changes ... 67
3.10.2 Manual staff switches ... 68
3.10.3 Pedals ... 69
3.10.4 Arpeggio ... 70
3.10.5 Staff switch lines ... 71
3.11 Vocal music ... 71
3.11.1 Entering lyrics ... 71
3.11.2 The Lyrics context ... 72
3.11.3 More stanzas ... 74
3.11.4 Ambitus ... 75

3.12 Tablatures . . . 76
3.12.1 Tablatures basic . . . 76
3.12.2 Non-guitar tablatures . . . 77
3.13 Chord names . . . 77
3.13.1 Chords mode . . . 78
3.13.2 Printing chord names . . . 80
3.14 Orchestral music . . . 81
3.14.1 Multiple staff contexts . . . 81
3.14.2 Rehearsal marks . . . 82
3.14.3 Bar numbers . . . 83
3.14.4 Instrument names . . . 83
3.14.5 Transpose . . . 84
3.14.6 Instrument transpositions . . . 85
3.14.7 Multi measure rests . . . 85
3.14.8 Automatic part combining . . . 86
3.14.9 Hiding staves . . . 87
3.14.10 Different editions from one source . . . 88
3.14.11 Quoting other voices . . . 89
3.15 Ancient notation . . . 90
3.15.1 Ancient note heads . . . 90
3.15.2 Ancient accidentals . . . 91
3.15.3 Ancient rests . . . 91
3.15.4 Ancient clefs . . . 92
3.15.5 Ancient flags . . . 94
3.15.6 Ancient time signatures . . . 95
3.15.7 Custodes . . . 95
3.15.8 Divisiones . . . 96
3.15.9 Ligatures . . . 97
3.15.9.1 White mensural ligatures . . . 98
3.15.9.2 Gregorian square neumes ligatures . . . 98
3.15.10 Vaticana style contexts . . . 102
3.15.11 Figured bass . . . 103
3.16 Contemporary notation . . . 104
3.16.1 Clusters . . . 104
3.16.2 Fermatas . . . 104
3.17 Special notation . . . 105
3.17.1 Balloon help . . . 105
3.17.2 Easy Notation note heads . . . 105
3.18 Sound . . . 105
3.18.1 MIDI block . . . 106
3.18.2 MIDI instrument names . . . 106

4 Changing defaults . . . 107

4.1 Scheme tutorial . . . 107
4.2 Interpretation contexts . . . 109
4.2.1 Creating contexts . . . 110
4.2.2 Changing context properties on the fly . . . 111
4.2.3 Modifying context plug-ins . . . 112
4.2.4 Layout tunings within contexts . . . 113
4.2.5 Changing context default settings . . . 115
4.2.6 Defining new contexts . . . 115
4.2.7 Which properties to change . . . 117
4.3 Tuning output . . . 117
4.3.1 Common tweaks . . . 117

4.3.2 Constructing a tweak 118
4.3.3 Navigating the program reference 119
4.3.4 Layout interfaces 120
4.3.5 Determining the grob property 121
4.4 Fonts 121
4.4.1 Selecting font sizes 121
4.4.2 Font selection 122
4.5 Text markup 123
4.5.1 Overview of text markup commands 124
4.6 Global layout 128
4.6.1 Setting global staff size 128
4.6.2 Vertical spacing 128
4.6.3 Horizontal Spacing 129
4.6.4 Line breaking 130
4.6.5 Page layout 131
4.7 Output details 132

5 Invoking LilyPond 133

5.1 Invoking lilypond 133
5.1.1 Titling layout 134
5.1.2 Additional parameters 134
5.2 Invoking the lilypond binary 135
5.3 Command line options 135
5.4 Environment variables 136
5.5 Error messages 137
5.6 Reporting bugs 138
5.7 Editor support 138
5.8 Point and click 138

6 lilypond-book manual 140

6.1 Integrating Texinfo and music 140
6.2 Integrating LaTeX and music 140
6.3 Integrating HTML and music 141
6.4 Music fragment options 142
6.5 Invoking lilypond-book 143
6.6 Bugs 144

7 Converting from other formats 145

7.1 Invoking convert-ly 145
7.2 Invoking midi2ly 145
7.3 Invoking etf2ly 146
7.4 Invoking abc2ly 147
7.5 Invoking musedata2ly 147
7.6 Invoking mup2ly 148

Unified index 149

Appendix A Notation manual details 158

A.1 Chord name chart 158
A.2 MIDI instruments 159
A.3 The Feta font 160
A.4 All context properties 164
A.5 Layout properties 171

Appendix B Literature list 182

Appendix C Interfaces for programmers 183
C.1 Programmer interfaces for input 183
C.1.1 Input variables and Scheme 183
C.1.2 Internal music representation 183
C.1.3 Manipulating music expressions 184
C.2 Markup programmer interface 185
C.2.1 Markup construction in scheme 185
C.2.2 Markup command definition 186
C.3 Contexts for programmers 187
C.3.1 Context evaluation 187
C.3.2 Running a function on all layout objects 188

Appendix D Cheat sheet 189

Appendix E GNU Free Documentation License 192
E.0.1 ADDENDUM: How to use this License for your documents 197

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 is 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 that 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 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 to version 2.2

During the 2.1 development cycle, the cleanup of the existing features has continued unabated. Major areas of improvement are orchestral notation, lyrics formatting and font size handling.

Han-Wen and Jan

Utrecht/Eindhoven, The Netherlands, March 2004.

1 Introduction

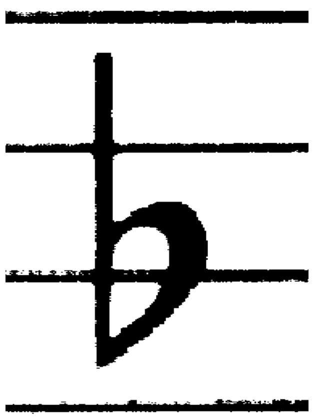
LilyPond is a system for formatting music prettily. This chapter discusses the backgrounds of LilyPond. It explains the problem of printing music with computers, and our approach to solving those problems.

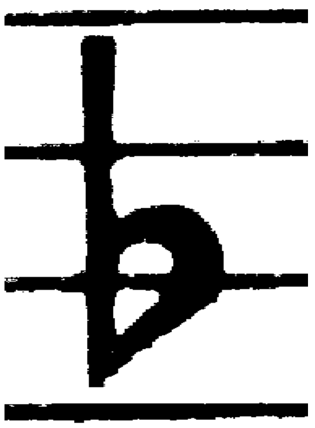
1.1 Engraving

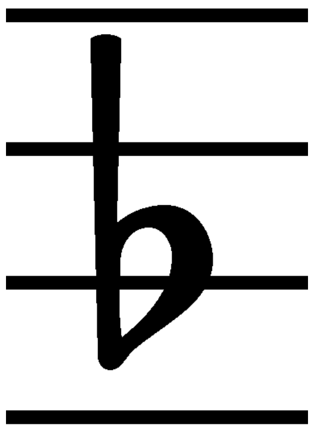
The art of music typography is called *(plate) engraving*. The term derives from the traditional process of music printing. Just a few decades ago, sheet music was made by cutting and stamping the music into a zinc or pewter plate in mirror image. The plate would be inked, the depressions caused by the 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 ten years of practical training before he could be a master engraver.

Nowadays, all newly printed music is produced with computers. This has obvious advantages; prints are cheaper to make, 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.

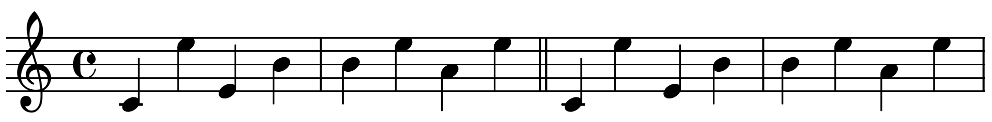
The images below illustrate the difference between traditional engraving and typical computer output, and the third picture shows how LilyPond mimics the traditional look. The left picture shows a scan of a flat symbol from a Henle edition published in 2000. In the center show symbol from a hand engraved 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 a poor result. 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-up 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 piece of music; 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, they become like a labyrinth. If the musician looks away once or has a lapse in his concentration, he will be lost on the page.

Similarly, the strong visual look of bold symbols on heavy staff lines stands out better when music is far away from 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 his task. For performance material like sheet music, this is doubly important: musicians have a limited amount of attention. The less attention they need for reading, the more they can focus on playing itself. In other words, better typography translates to better performances.

Hopefully, these examples also demonstrate that music typography is an art that is subtle and complex, and to produce 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 ultimately. 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 automatically, so manual overrides are less and less necessary.

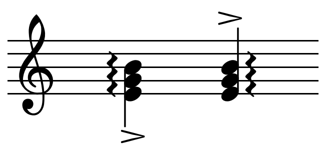
When we started we wrote the LilyPond program entirely in the C++ programming language, the program's functionality was set in stone by the developers. That proved to be unsatisfactory for a number of reasons:

- When LilyPond makes mistakes, users need to override formatting decisions. Therefore, the user must access to the formatting engine. Hence, rules and settings cannot be fixed by us at compile time, but they 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.

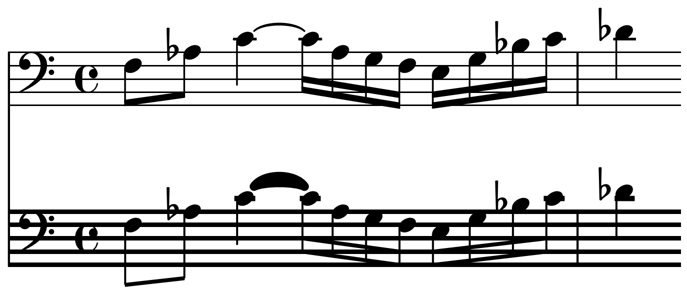
These problems have been addressed by integrating the GUILE interpreter for the Scheme programming language and rewriting parts of LilyPond in Scheme. The new, flexible formatting is built around the notion of graphical objects, described by Scheme variables and functions. This architecture encompasses 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 arpeggio. In the first chord, the graphical objects have all directions down (or left). The second chord has all directions up (right).

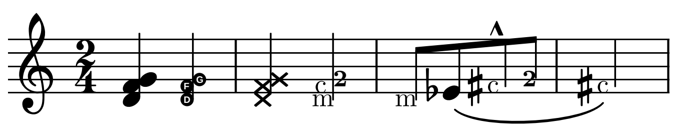


The process of formatting a score consists of reading and writing the variables of graphical objects.

Some variables have a preset value. For example, the thickness of many lines—a characteristic of typographical style—are preset variables. Changing them gives a different typographical impression



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



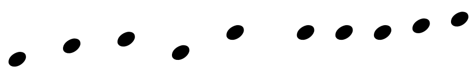
1.3 What symbols to engrave?

The formatting process in LilyPond 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.

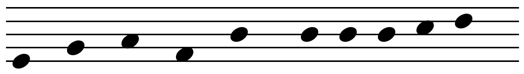
Common music notation is a system of recording music that has evolved over the past 1000 years. The form that is now in common use, dates from the early renaissance. Although, the basic form (i.e. note heads on a 5-line staff) has not changed, the details still change to express the innovations of contemporary notation. Hence, it encompasses some 500 years of music. Its applications range from monophonic melodies to monstrous counterpoint 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? We have broken 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. People that translate musical ideas to graphic symbols are called copyists or engravers, so by analogy, each plug-in is called `engraver`.

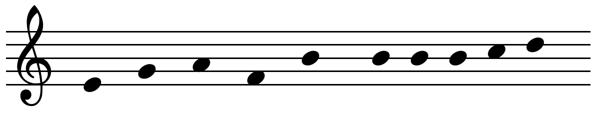
In the following example, we see how we start out with a plug-in for note heads, the `Note_heads_engraver`.



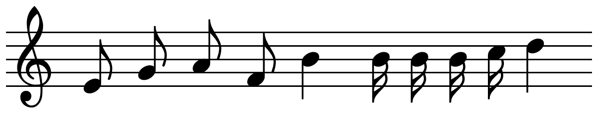
Then a `Staff_symbol_engraver` adds the staff



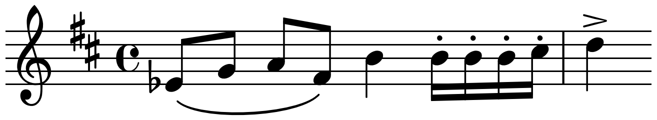
The `Clef_engraver` defines a reference point for the staff



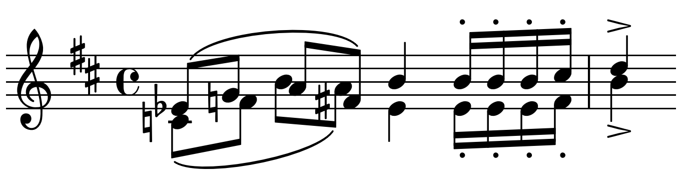
And the `Stem_engraver` adds stems



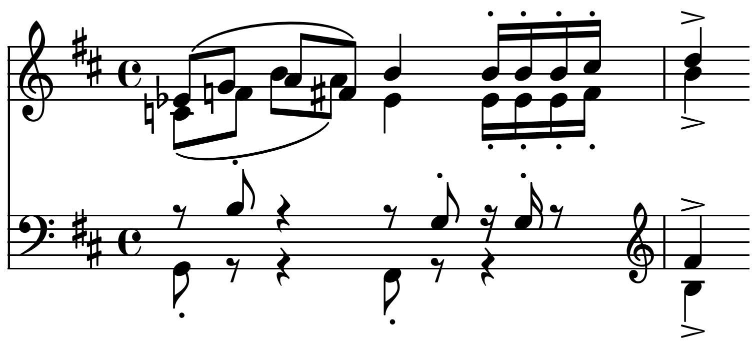
The `Stem_engraver` is notified of any note head coming along. Every time one (or more, for a chord) note head is seen, a stem object is created and connected to the note head. By adding engravers for beams, slurs, accents, accidentals, bar lines, time signature, and key signature, we get a complete 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. In polyphonic notation, many voices can share a staff. Similarly, more Staff contexts can be put into a single Score context



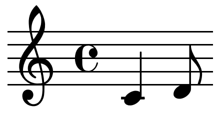
1.4 Music representation

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

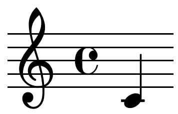
```
c'4 d'8
```

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



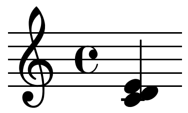
On a microscopic scale, such syntax is easy to use. On a larger scale, syntax also needs structure. How else can you enter complex pieces like symphonies and operas? The structure is formed by the concept of music expressions: by combining small fragments of music into larger ones, more complex music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

```
c4
```



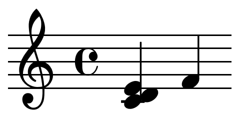
Combine this simultaneously with two other notes by enclosing in << and >>

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



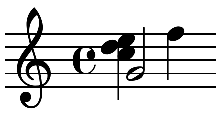
This expression is put in sequence by enclosing it in curly braces { ... }

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



The above is another expression, and therefore, it many combined again with a simultaneous expression; in this case, a half note

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

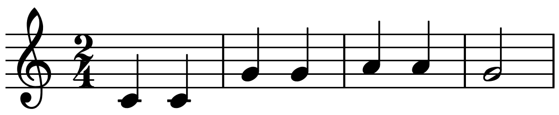


Such recursive structures can be specified neatly and formally in a context-free grammar. The parsing code is also generated from this grammar. In other words, the syntax of LilyPond is clearly and unambiguously defined.

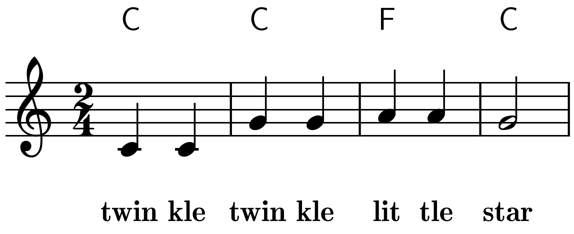
User-interfaces and syntax are what people see and deal with most. They are partly a matter of taste, and also subject of much discussion. Although discussions on taste do have their merit, they are not very productive. In the larger picture of LilyPond, the importance of input syntax is small: inventing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

1.5 Example applications

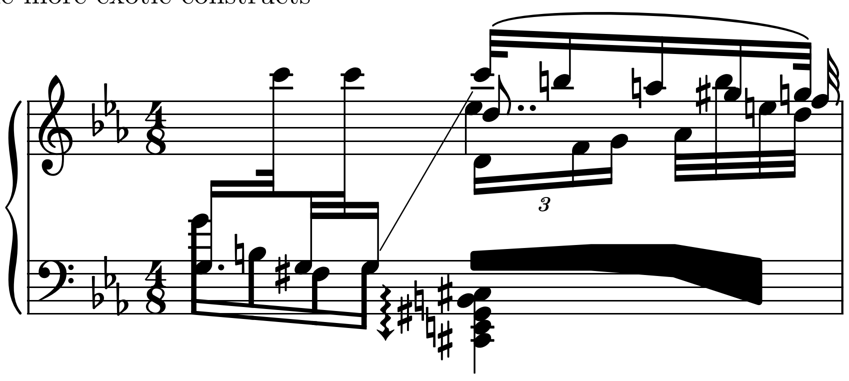
We have written LilyPond as an experiment of how to condense the art of music engraving into a computer program. Thanks to all that hard work, the program can now be used to perform useful tasks. The simplest application is printing notes



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



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 PostScript 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 [Notation manual], page 31* discusses topics grouped by notation construct. Once you master the basics, this is the place to look up details.
- *Chapter 4 [Changing defaults], page 107* explains how to fine tune layout.
- *Chapter 5 [Invoking LilyPond], page 133* shows how to run LilyPond and its helper programs.
- *Chapter 6 [lilypond-book manual], page 140* explains the details behind creating documents with in-line music examples (like this manual).
- *Chapter 7 [Converting from other formats], page 145* 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 B [Literature list], page 182* 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 documents the nit-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 thicknesses, distances, etc, can be changed in input files. There are a huge number of formatting options, and all of them are described in the generated documentation. Each section of the notation manual has a **See also** subsection, which refers to the the generated documentation. In the HTML document, these subsections have clickable links.
- Templates

  After you have gone through the tutorial, you should be able to write input files. In practice, writing files from scratch turns out to be intimidating. To give you a head start, we have

[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.

collected a number of often-used formats in example files. These files can be used as a start; simply copy the template and add notes in the appropriate places.

- 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 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-1.7.19/input/test/bla.ly`'. On binary packages for the Unix platform, the documentation and examples can typically be found somewhere below '`/usr/share/doc/lilypond/`'. Initialization files, for example '`scm/lily.scm`', or '`ly/engraver-init.ly`', are usually found in the directory '`/usr/share/lilypond/`'.

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

2 Tutorial

This tutorial starts with 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.

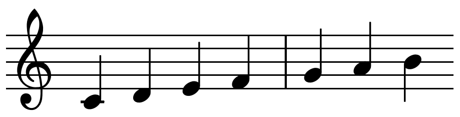
By cutting and pasting the full input into a test file, you have a starting template for experiments. If you like learning in this way, you will probably want to print out or bookmark Appendix D [Cheat sheet], page 189, which is a table listing all commands for quick reference.

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

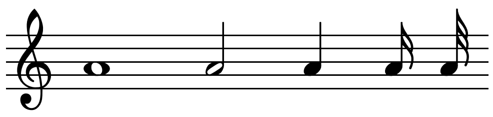
```
c d e f g a b
```

the result looks like this



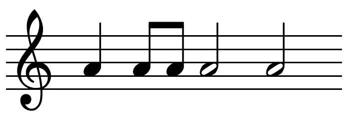
The duration of a note is specified by a number after the note name. '1' for a whole note, '2' for a half note, '4' for a quarter note and so on

```
a1 a2 a4 a16 a32
```



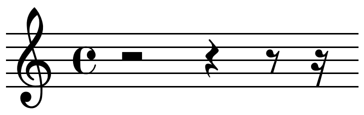
If you do not specify a duration, the duration last entered is used; the duration of the first note defaults to a quarter

```
a a8 a a2 a
```



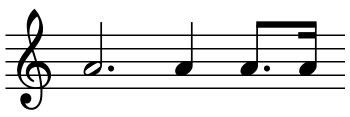
Rests are entered just like notes, but with the name "r"

```
r2 r4 r8 r16
```



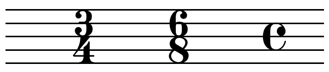
Add a dot '.' after the duration to get a dotted note

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



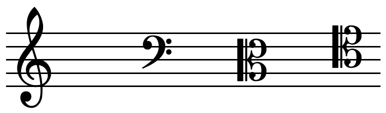
The meter (or time signature) can be set with the `\time` command

```
\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
```



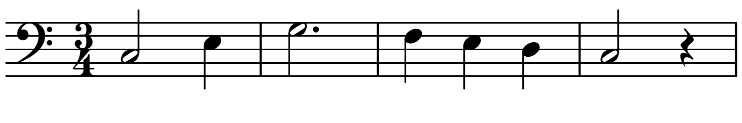
To recognize names like `c` and `d` as pitches, they have to be entered inside a so-called `\notes` block. This block is formed by enclosing notes and commands are enclosed in curly braces `{ ... }` and adding the keyword `\notes` before the opening brace

```
\notes {
  \time 3/4
  \clef bass
  c2 e4 g2.
  f4 e d c2 r4
}
```

Now the piece of music is almost ready to be printed. Enclose the `\notes` block in a `\score` block

```
\score {
  \notes {
    \time 3/4
    \clef bass
    c2 e4 g2.
    f4 e d c2 r4
  }
}
```

and the music will be converted to printable output.



In many examples in this manual, both `\score` and `\notes` and accompanying braces are left out for brevity. However, they must be present when feeding the file to LilyPond.

For more elaborate information on

Entering pitches and durations
: see Section 3.1.2 [Pitches], page 31 and Section 3.1.7 [Durations], page 34.

Clefs
: see Section 3.3.3 [Clef], page 40

Rests
: see Section 3.1.5 [Rests], page 33.

Time signatures and other timing commands
    see Section 3.3.5 [Time signature], page 42.

2.2 Running LilyPond

In the last section we explained what kind of things you could 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`'

```
\score {
  \notes { c'4 e' g' }
}
```

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

```
lilypond test.ly
```

You will see something resembling

```
GNU LilyPond 1.8.0
Now processing: '/home/fred/ly/test.ly'
Parsing...
Interpreting music...[1]
  ...  more interesting stuff ...
PDF output to 'test.pdf'...
DVI output to 'test.dvi'...
```

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

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

```
\score {
  \notes { 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 (♯) pitch is made by adding '`is`' to the name, a flat (♭) pitch by adding '`es`'. As you might expect, a double sharp or double flat is made by adding '`isis`' or '`eses`':[5]

```
cis1 ees fisis aeses
```

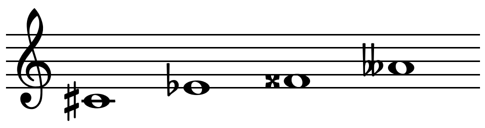
[1] There are macro files for VIM addicts, and there is a `LilyPond-mode` for Emacs addicts. If it has not been installed already, refer to the file '`INSTALL.txt`'

[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 http://www.cs.wisc.edu/~ghost/ (`Ghostscript`), a freely available package for viewing and printing PDF and PostScript files.

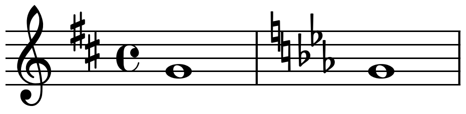
[4] Any simple or programmer-oriented editor will do, for example Notepad. Do not use a word processor, its formatting codes will confuse LilyPond

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



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

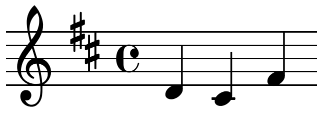
```
\key d \major
g1
\key c \minor
g
```



Key signatures together with the pitches (including alterations) are used together 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 a 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.

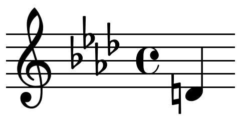
In this example



no note gets an explicit accidental, but still you 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, 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 music can be printed according to different conventions. See Section 3.6 [Accidentals], page 49 for some examples how accidentals can be printed according to different rules.

For more information on

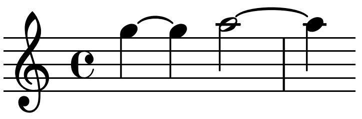
Accidentals
: see Section 3.6 [Accidentals], page 49

Key signature
: see Section 3.3.2 [Key signature], page 40

2.4 Entering ties

A tie is created by adding a tilde "~" to the first note being tied

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

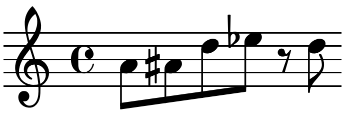


For more information on Ties, see Section 3.1.9 [Ties], page 35.

2.5 Automatic and manual beams

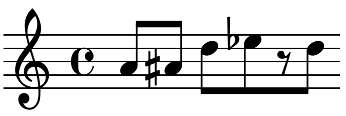
Beams are drawn automatically

```
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].

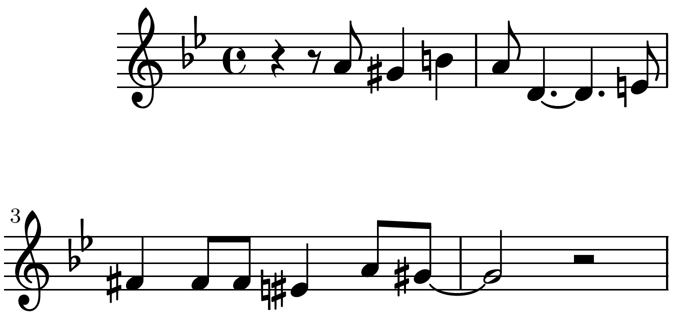
```
a8[ ais] d[ es r d]
```



For more information on beams, see Section 3.5 [Beaming], page 46.

Here are key signatures, accidentals and ties in action

```
\score {
  \notes {
    \time 4/4
    \key g \minor
    \clef violin
    r4 r8 a8 gis4 b
    g8 d4.~ d e8
    fis4 fis8 fis8 eis4  a8 gis~
    gis2 r2
  }
}
```

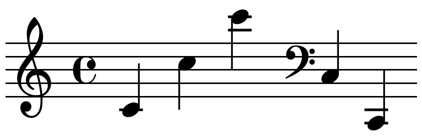


There are some interesting points to note in this example. Bar lines and beams are drawn automatically. Line breaks are calculated automatically; it does not matter where the line breaks are in the source file. Finally, the order in which time, key and clef changes are entered is not relevant: in the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

2.6 Octave entry

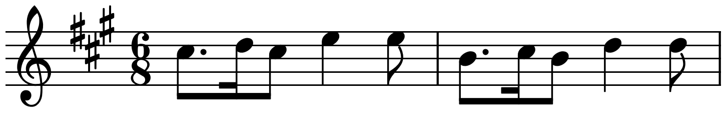
To raise a note by an octave, add a high quote `'` (apostrophe) to the note name, to lower a note one octave, add a "low quote" `,` (a comma). Middle C is `c'`

```
c'4 c'' c''' \clef bass c c,
```



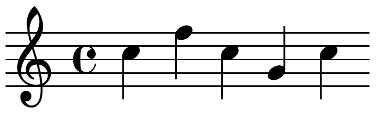
An example of the use of quotes is in the following Mozart fragment

```
\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
```



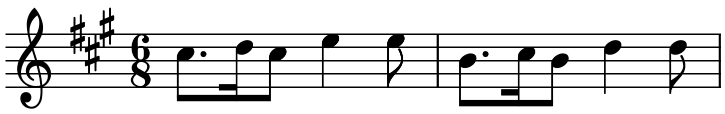
This example shows that music in a high register needs lots of quotes. This makes the input less readable, and it is a source of errors. The solution is to use "relative octave" mode. In practice, this is the most convenient way to copy existing music. To use relative mode, add `\relative` before the piece of music. You must also give a note from which relative starts, in this case `c''`. If you do not use octavation quotes (i.e. do not add `'` or `,` after a note), relative mode chooses the note that is closest to the previous one. For example, `c f` goes up while `c g` goes down

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



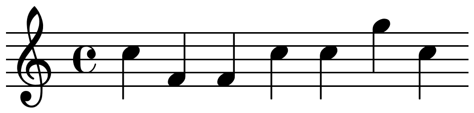
Since most music has small intervals, pieces can be written almost without octavation quotes in relative mode. The previous example is entered as

```
\relative c'' {
  \key a \major
  \time 6/8
  cis8. 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,
}
```



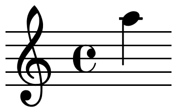
In `\relative` mode, quotes or commas no longer determine the absolute height of a note. Rather, the height of a note is relative to the previous one, and changing the octave of a single note shifts all following notes an octave up or down.

For more information on Relative octaves see Section 3.2.1 [Relative octaves], page 37 and Section 3.2.2 [Octave check], page 38.

2.7 Music expressions explained

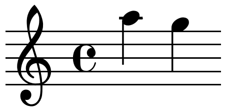
In input files, music is represent by so-called *music expression*. We have already seen in the previous examples; a single note is a music expression

```
a4
```



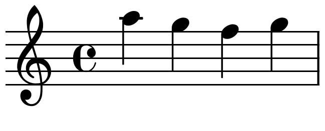
Enclosing group of notes in braces creates a new music expression

```
{ a4 g4 }
```



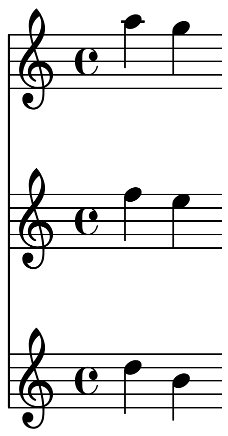
Putting a bunch of music expressions (notes) in braces, means that they should be played in sequence. The result again is a music expression, which can be grouped with other expressions sequentially. Here, the expression from the previous example is combined with two notes

```
{ { 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 notes) are combined simultaneously

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



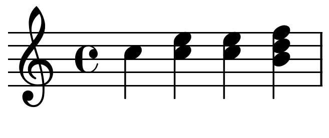
This mechanism is similar to mathematical formulas: a big formula is created by composing small formulas. Such formulas are called expressions, and their definition is recursive, so you can make arbitrarily complex and large expressions. For example,

```
1

1 + 2

(1 + 2) * 3

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

This example shows a sequence of expressions, where each expression is contained in the next one. The simplest expressions are numbers and operators (like +, * and /). Parentheses are used to group expressions.

Like mathematical expressions, music expressions can be nested arbitrarily deep, e.g.

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



When spreading expressions over multiple lines, 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,

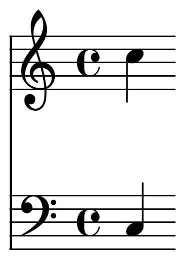
```
\score {
  \notes <<
    {
      ...
    }
    {
      ...
    }
  >>
}
```

Some editors have special support for entering LilyPond, and can help indenting source files. See Section 5.7 [Editor support], page 138 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`'s are then combined parallel with `<<` and `>>`, as demonstrated here

```
<<
  \new Staff { \clef violin c'' }
  \new Staff { \clef bass c }
>>
```



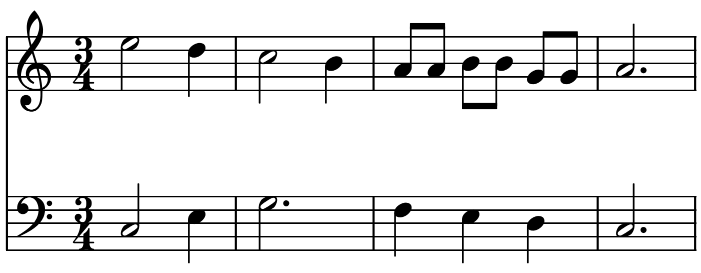
The command `\new` introduces a "notation context." A notation context is an environment in which musical events (like notes or `\clef` commands) are interpreted. For simple pieces, such notation contexts are created implicitly. For more complex pieces, it is best to mark contexts explicitly. This ensures that each fragment gets its own stave.

There are several types of contexts: `Staff`, `Voice` and `Score` handle normal music notation. Other staves are also `Lyrics` (for setting lyric texts) and `ChordNames` (for printing 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

```
\score {
  \notes <<
    \new Staff {
      \time 3/4
      \clef violin
      \relative c'' {
        e2 d4 c2 b4 a8[ a]
        b[ b] g[ g] a2. }
    }
    \new Staff {
       \clef bass
       c2 e4  g2.
       f4 e d c2.
    }
  >>
}
```

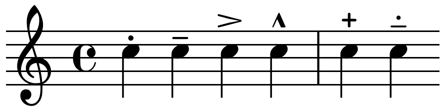


For more information on context see the description in Section 4.2 [Interpretation contexts], page 109.

2.9 Adding articulation marks to notes

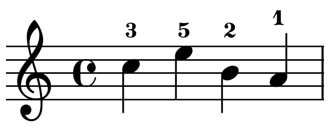
Common accents can be added to a note using a dash (‘-’) and a single character

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



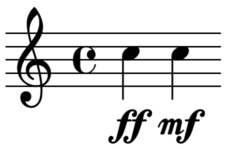
Similarly, fingering indications can be added to a note using a dash (‘-’) and the digit to be printed

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



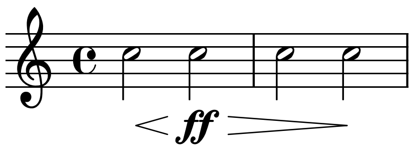
Dynamic signs are made by adding the markings (with a backslash) to the note

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



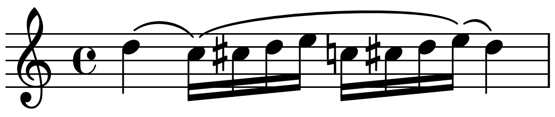
Crescendi and decrescendi are started with the commands `\<` and `\>`. An ending dynamic, for example `\f`, will finish the crescendo, or the command `\!` can be used

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

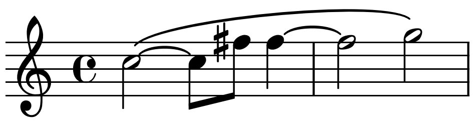


A slur is a curve drawn across many notes, and indicates legato articulation. The starting note and ending note are marked with a “(” and a “)” respectively

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

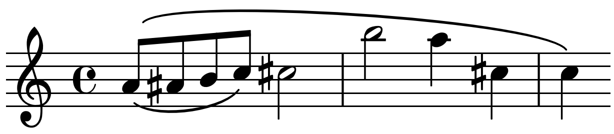


A slur looks like a tie, but it has a different meaning. A tie simply makes the first note sound longer, and can only be used on pairs of notes with the same pitch. Slurs indicate the articulations of notes, and can be used on larger groups of notes. Slurs and ties are also nested in practice



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

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



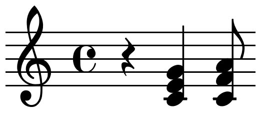
For more information on

Fingering	see Section 3.7.8 [Fingering instructions], page 56
Articulations	see Section 3.7.7 [Articulations], page 54
Slurs	see Section 3.7.1 [Slurs], page 51
Phrasing slurs	see Section 3.7.2 [Phrasing slurs], page 52
Dynamics	see Section 3.7.12 [Dynamics], page 60
Fingering	

2.10 Combining notes into chords

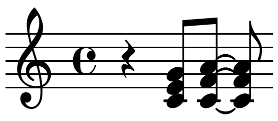
Chords can be made by surrounding pitches with angled brackets. Angled brackets are the symbols `<` and `>`.

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

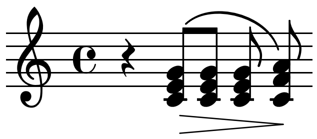


You can combine markings like beams and ties with chords. They must be placed outside the angled brackets

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



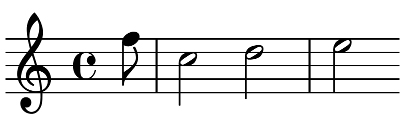
```
r4 <c e g>8\>( <c e g> <c e g>  <c f a>8\!)
```



2.11 Basic rhythmical commands

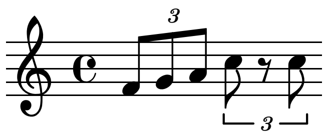
A pickup is entered with the keyword `\partial`. It is followed by a duration: `\partial 4` is a quarter note upstep and `\partial 8` an eighth note

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



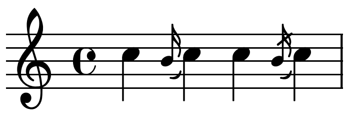
Tuplets are made with the `\times` keyword. It takes two arguments: a fraction and a piece of music. The duration of the piece of music is multiplied by the fraction. Triplets make notes occupy 2/3 of their notated duration, so a triplet has 2/3 as its fraction

```
\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 3.7.10 [Grace notes], page 57,
Tuplets	see Section 3.1.10 [Tuplets], page 36,
Pickups	see Section 3.3.6 [Partial measures], page 43.

2.12 Commenting input files

A comment is a remark for the human reader of the music input, it is ignored and has no effect on the printed output. There are two types of comments. The percent symbol `%` introduces a line comment; the rest of the line is ignored. A block comments 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

%{
    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 really is a kind of comment. The version statement marks for which version of LilyPond the file was written. To mark a file for version 2.1.17, use

```
\version "2.1.17"
```

These annotations make future upgrades of LilyPond go more smoothly. Changes in the syntax are handled with a special program, '`convert-ly`' (see Section 7.1 [Invoking convert-ly], page 145), 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
```

To prevent certain words (for example "as") as being read as a pitch, the input-mode must be switched. This is done with `\lyrics`. In `\lyrics` mode, all words are read as lyric syllables.

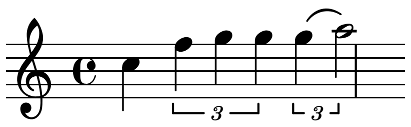
```
  \lyrics { I want to break free }
```

Again, the braces `{}` signify that the syllables are sung in sequence.

By default, music expressions are interpreted in `Staff` context. For lyrics, this is obviously not desirable, so it is necessary to explicitly specify a `Lyrics` context,

```
\new Lyrics  \lyrics { I want to break free }
```

The melody for this song is as follows



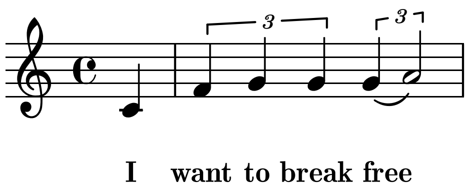
The lyrics can be set to these notes, combining both with the `\lyricsto` keyword

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

where *name* identifies to which melody the lyrics should be aligned. In this case, there is only one melody, so we can leave it empty.

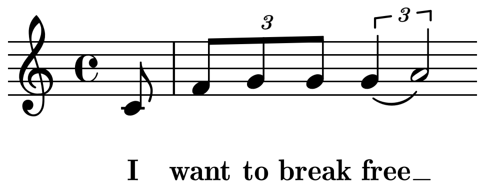
The final result is

```
\score  {
 \notes <<
   \partial 4
   \relative c' {
      c4
      \times 2/3 { f g g } \times 2/3 { g4( a2) }
   }
   \lyricsto "" \new Lyrics \lyrics { I want to break free }
 >>
}
```



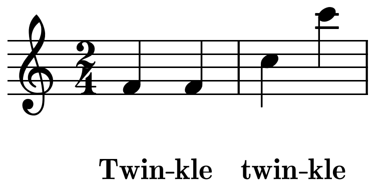
This melody ends on a melisma, a single syllable ("free") sung to more than one note. This is indicated with an *extender line*. It is entered as two underscores, i.e.

```
\lyrics { 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 3.11 [Vocal music], page 71.

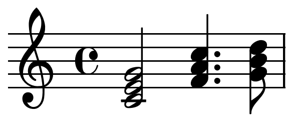
2.14 A lead sheet

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

```
c2 f4. g8
```

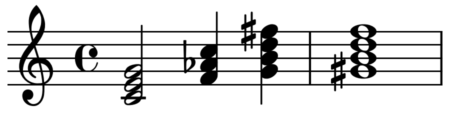
but now, each pitch is read as the root of a chord instead of a note. This mode is switched on with `\chords`

```
\score {
  \chords { c2 f4. g8 }
}
```



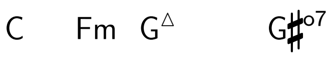
Other chords can be created by adding modifiers after a colon. The following example shows a few common modifiers

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



For lead sheets, chords are not printed on staves, but as names on a line of themselves. Hence, we have to override the context with `\new`, rendering the music expression in a `ChordNames` context

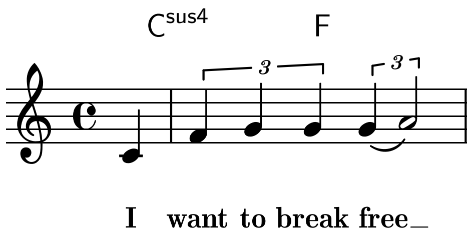
```
\new ChordNames \chords { c2 f4.:m g4.:maj7 gis8:dim7 }
```



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

```
\score {
  <<
```

```
    \new ChordNames \chords { chords }
    \notes the melody
    \lyricsto "" \new Lyrics \lyrics { the text }
  >>
}
```



A complete list of modifiers and other options for layout can be found in Section 3.1.4 [Chords], page 33.

2.15 Listening to output

MIDI (Musical Instrument Digital Interface) is a standard for connecting and controlling digital instruments. A MIDI file is like a tape recording of a MIDI instrument.

To create a MIDI from a music piece of music, add a `\midi` block causes LilyPond to create a MIDI file, so you can listen to the music you entered. It is great for checking the music: octaves that are off or accidentals that were mistyped stand out very much when listening to the musical transcription.

The `\midi` block is added to `\score`, for example,

```
\score {
    ..music..
    \midi  { \tempo 4=72 }
}
```

Here, the tempo is specified using the `\tempo` command. In this case the tempo of quarter notes is set to 72 beats per minute. More information on auditory output in the notation manual, Section 3.18 [Sound], page 105.

If there is a `\midi` command in a `\score`, only MIDI will be produced. If notation is needed too, a `\paper` block must be added too

```
\score {
    ..music..
    \midi  { \tempo 4=72 }
    \paper { }
}
```

2.16 Titling

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 { ... }`. For example,

```
\header {
  title = "Eight miniatures"
  composer = "Igor Stravinsky"
  tagline = "small is beautiful"
}

\score { ... }
```

When the file is processed the title and composer are printed above the music. The 'tagline' is a short line printed at bottom of the last page which normally says "Engraved by LilyPond, version . . .". In the example above it is replaced by the line "small is beautiful."[6]

Normally, the `\header` is put at the top of the file. However, for a document that contains multiple pieces (e.g. an etude book, or an orchestral part with multiple movements), the header can be put in the `\score` block as follows; in this case, the name of each piece will be printed before each movement

```
\header {
  title = "Eight miniatures"
  composer = "Igor Stravinsky"
  tagline = "small is beautiful"
}

\score { ...
  \header { piece = "Adagio" }
}
\score { ...
  \header { piece = "Menuetto" }
}
```

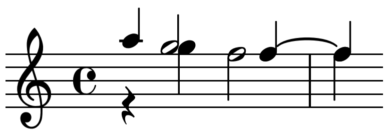
More information on titling can be found in Section 5.1 [Invoking lilypond], page 133.

2.17 Single staff polyphony

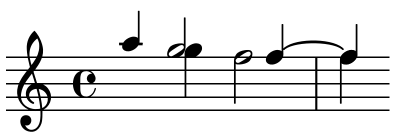
When different melodic lines are combined on a single staff they are printed as polyphonic voices: each voice has its own stems, slurs and beams, and the top voice has the stems up, while the bottom voice has them down.

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

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

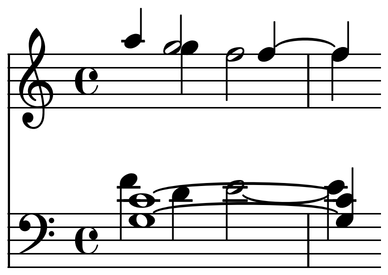


For polyphonic music typesetting, spacer rests can also be convenient: these are rests that do not print. It is useful for filling up voices that temporarily do not play. Here is the same example with a spacer rest instead of a normal rest



Again, these expressions can be nested arbitrarily

[6] Nicely printed parts are good PR for us, so please leave the tagline if you can.



More features of polyphonic typesetting in the notation manual in Section 3.4 [Polyphony], page 45.

2.18 Piano staves

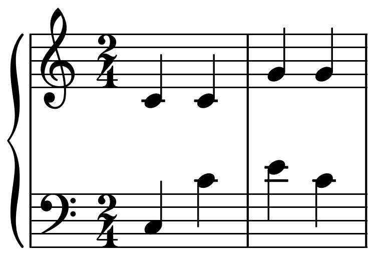
Piano music is typeset in two staves connected by a brace. Printing such a staff is similar to the polyphonic example in Section 2.8 [More staves], page 18

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

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

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

Here is a full-fledged example



More information on formatting piano music is in Section 3.10 [Piano music], page 67.

2.19 Organizing larger pieces

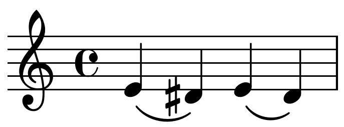
When all of the elements discussed earlier are combined to produce larger files, the `\score` blocks get a lot bigger, because the music expressions are longer, and, in the case of polyphonic pieces, more deeply nested. Such large expressions can become unwieldy.

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

```
namedMusic = \notes { ...
```

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 = \notes {
  e'4( dis'4)
}
\score {
  \new Staff { \seufzer \seufzer }
}
```



The name of an identifier should have alphabetic characters only; no numbers, underscores or dashes. The assignment should be outside of the `\score` block.

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

```
\score {
  \notes { c4^\name }
  \paper {
    \aFivePaper
    linewidth = \width
  }
}
```

More information on the possible uses of identifiers is in the technical manual, in TODO.

2.20 An orchestral part

In orchestral music, all notes are printed twice: both in a part for the musicians, and 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 score.

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

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

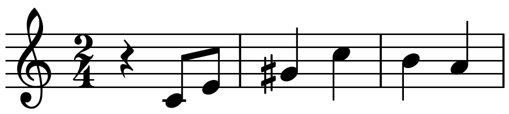
Then, an individual part is made by putting the following in a file

```
\include "horn-music.ly"
\header {
  instrument = "Horn in F"
}
\score {
  \notes \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 tuning of a normal French Horn in F. The transposition can be seen in the following output



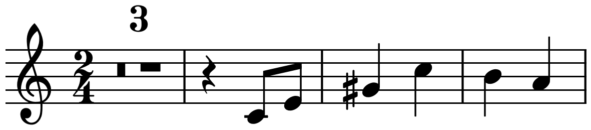
In ensemble pieces, one of the voices often does not play for many measures. This is denoted by a special rest, the multi-measure rest. It is entered with a capital `R` followed by a duration (1 for a whole note, 2 for a half note, etc.) By multiplying the duration, longer rests can be constructed. For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

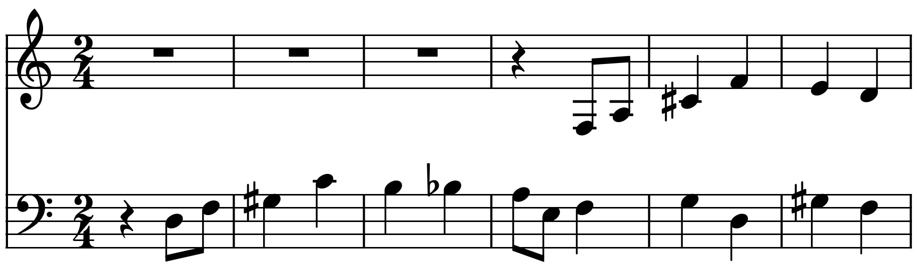
This commands sets the property `skipBars` property in the `Score` context to true (`##t`). Prepending the rest and this option to the music above, leads to the following result



The score is made by combining all of the music in a `\score` block. 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"

\score {
  \simultaneous {
    \new Staff \hornNotes
    \new Staff \bassoonNotes
  }
}
```

leading to



More in-depth information on preparing parts and scores in the notation manual, in Section 3.14 [Orchestral music], page 81.

Setting run-time variables ("properties") is discussed in ref-TODO.

2.21 Integrating text and music

Some texts include music examples. Examples 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 HTML, LaTEX, and Texinfo documents.

A script called `lilypond-book` will extract the music fragments, run format them, and put back the resulting notation. This program is fully described in Chapter 6 [lilypond-book manual], page 140. Here we show a small example. The example also contains explanatory text, so we will not comment on it further

```
\documentclass[a4paper]{article}
\begin{document}

Documents for lilypond-book may freely mix music and text.  For
example,

\begin{lilypond}
  \score { \notes \relative c' {
     c2 g'2 \times 2/3 { f8 e d } c'2 g4
  } }
\end{lilypond}

If there is no \verb+\score+ block in the fragment,
\texttt{lilypond-book} will supply one

\begin{lilypond}
  c'4
\end{lilypond}

In this example two things happened: a
\verb+\score+ block was added, and the line width was set to natural
length.

Options are put in brackets.

\begin[staffsize=26,verbatim]{lilypond}
  c'4 f16
\end{lilypond}


Larger examples can be put in a separate file, and introduced with
\verb+\lilypondfile+.

\lilypondfile{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) 2.1.19
Reading ‘input/tutorial/lilybook.tex’
Reading ‘input/screech-boink.ly’
lots of stuff deleted
Writing ‘out/lilybook.tex’
$ cd out
$ latex lilybook
lots of stuff deleted
$ xdvi lilybook
```

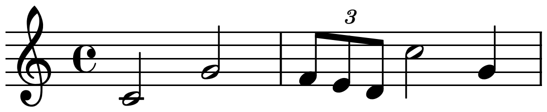
To convert the file into a nice PDF document, run the following commands

```
$ dvips -Ppdf -u +lilypond lilybook
$ ps2pdf lilybook.ps
```

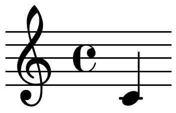
Running lilypond-book and running latex creates a lot of temporary files, which would clutter up the working directory. To remedy this, use the `output` option. It will create the files in a separate subdirectory '`out`'.

The result looks more or less like

Documents for lilypond-book may freely mix music and text. For example



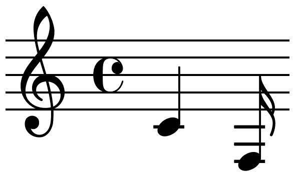
If you have no `\score` block in the fragment, `lilypond-book` will supply one



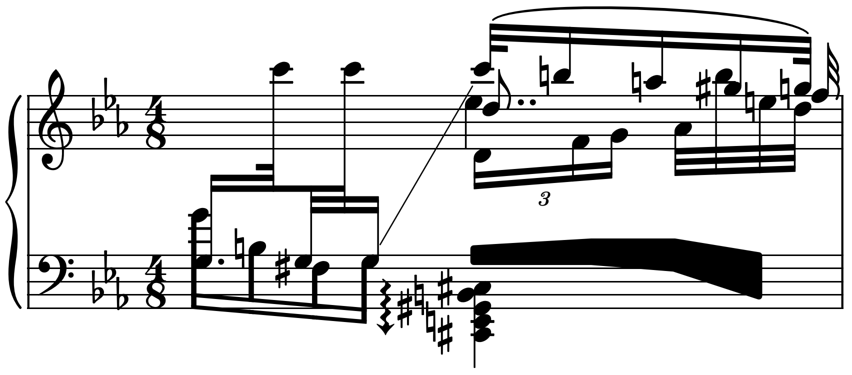
In this example two things happened: a `score` block was added, and the line width was set to natural length.

Options are put in brackets.

```
c'4 f16
```



Larger examples can be put in a separate file, and introduced with `\lilypondfile`.



3 Notation manual

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

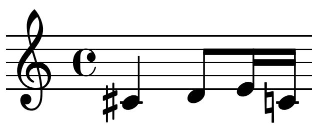
3.1 Note entry

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

3.1.1 Notes

A note is printed by specifying its pitch and then its duration[1]

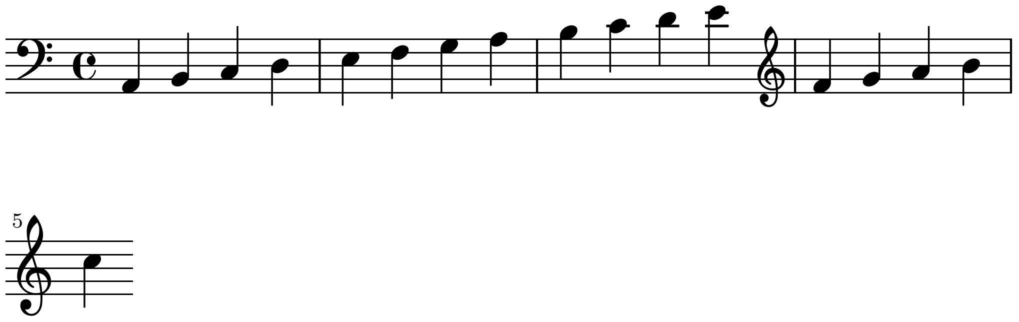
```
cis'4 d'8 e'16 c'16
```



3.1.2 Pitches

The most common syntax for pitch entry is used in `\chords` and `\notes` mode. In these modes, pitches may be designated by names. The notes are specified by the letters `a` through `g`, while 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''
```



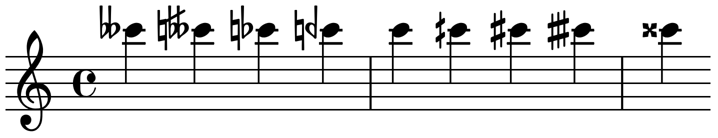
A sharp is formed by adding `-is` to the end of a pitch name and a flat is formed by adding `-es`. Double sharps and double flats are obtained by adding `-isis` or `-eses`. These names are the Dutch note names. In Dutch, `aes` is contracted to `as`, but both forms are accepted. Similarly, both `es` and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

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

```
ceses4
ceseh
ces
ceh
c
```

[1] Notes constitute the most basic elements of LilyPond input, but they do not form valid input on their own without a `\score` block. However, for the sake of brevity and simplicity we will generally omit `\score` blocks and `\paper` declarations in this manual.

```
cih
cis
cisih
cisis
```

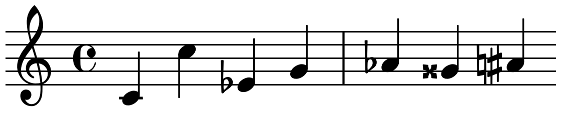


There are predefined sets of note names for various other languages. To use them, include the language specific init file. For example: `\include "english.ly"`. The available language files and the note names they define are

	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 -x (double)	-f/-flat
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`.

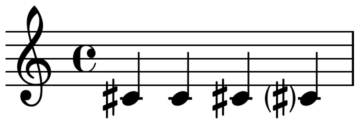
See also

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

3.1.3 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

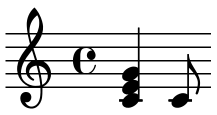
The automatic production of accidentals can be tuned in many ways. For more information, refer to Section 3.6 [Accidentals], page 49.

Bugs

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

3.1.4 Chords

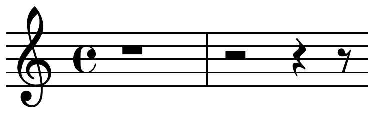
A chord is formed by a enclosing a set of pitches in `<` and `>`. A chord may be followed by a duration, and a set of articulations, just like simple notes.



3.1.5 Rests

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

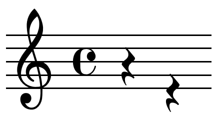
```
r1 r2 r4 r8
```



Whole bar rests, centered in middle of the bar, must be done with multi-measure rests. They are discussed in Section 3.14.7 [Multi measure rests], page 85.

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

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



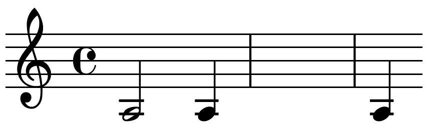
See also

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

3.1.6 Skips

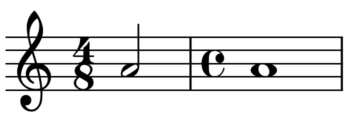
An invisible rest (also called a 'skip') can be entered like a note with note name '`s`' or with `\skip` *`duration`*

```
a2 s4 a4 \skip 1 a4
```



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

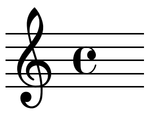
```
\score {
  \new Staff <<
    { \time 4/8 \skip 2 \time 4/4 }
    \notes\relative c'' { a2 a1 }
  >>
}
```



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.

```
\score { \notes { s4 } }
```



The same fragment using `\skip` results in an empty page.

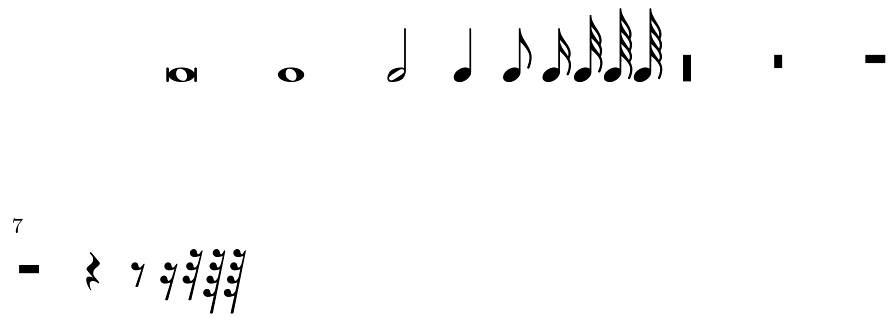
See also

Program reference: `SkipEvent`, `SkipMusic`.

3.1.7 Durations

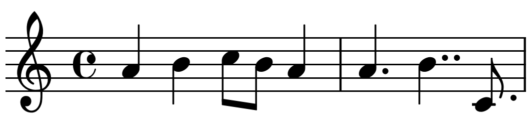
In Note, Chord, and Lyrics mode, durations are designated by numbers and dots: durations are entered as their reciprocal values. For example, a quarter note is entered using a `4` (since it is a 1/4 note), while a half note is entered using a `2` (since it is a 1/2 note). For notes longer than a whole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

```
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. The duration can be followed by dots (‘`.`’) in order to obtain dotted note lengths

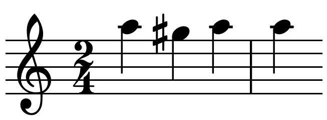
```
a' b' c''8 b' a'4 a'4. b'4.. c'8.
```



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
```



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`, `\dotsBoth`.

See also

This manual: Section 3.1.10 [Tuplets], page 36

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

3.1.8 Stems

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

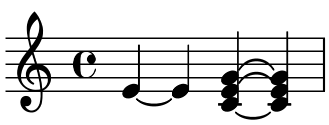
Predefined commands

`\stemUp`, `\stemDown`, `\stemBoth`.

3.1.9 Ties

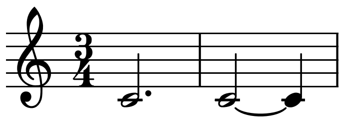
A tie connects two adjacent note heads of the same pitch. The tie in effect extends the length of a note. Ties should not be confused with slurs, which indicate articulation, or phrasing slurs, which indicate musical phrasing. A tie is entered using the tilde symbol '~'

```
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.

In its meaning a tie is just a way of extending a note duration, similar to the augmentation dot; in the following example there are two ways of notating exactly the same concept



If you need to tie a lot of notes over bars, it may be easier to use automatic note splitting (see Section 3.2.5 [Automatic note splitting], page 39).

Predefined commands

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

See also

In this manual: Section 3.2.5 [Automatic note splitting], page 39.

Program reference: `TieEvent`, `Tie`.

For tying only a subset of the note heads of a pair of chords, see '`input/regression/tie-chord-partial.ly`'.

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.

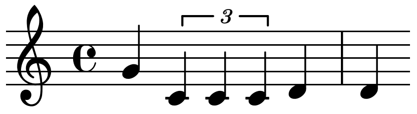
3.1.10 Tuplets

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

```
\times fraction musicexpr
```

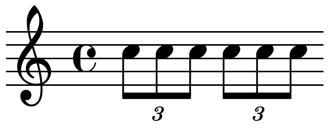
The duration of *musicexpr* will be multiplied by the fraction. The fraction's denominator will be printed over the notes, optionally with a bracket. The most common tuplet is the triplet in which 3 notes have the length of 2, so the notes are 2/3 of their written length

```
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, 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`, `\tupletBoth`.

See also

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`'.

3.2 Easier music entry

This section deals with tricks and features of the input language that were added solely to help entering music, finding and correcting mistakes. There are also external tools that make debugging easier. See Section 5.8 [Point and click], page 138 for more information.

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

3.2.1 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: a single error puts the rest of the piece off by one octave

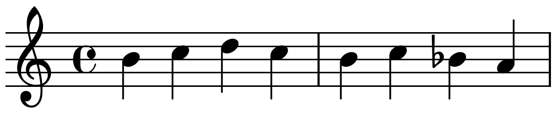
```
\relative startpitch musicexpr
```

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`.

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 must be specified that will act as the predecessor of the first note of *musicexpr*.

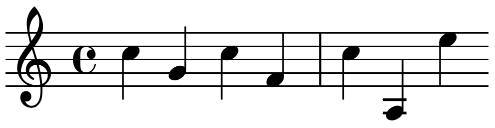
Here is the relative mode shown in action

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



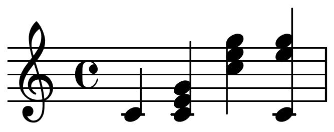
Octave changing marks are used for intervals greater than a fourth

```
\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. To parse the note name as a pitch, it must surrounded by `\notes`

The relative conversion will not affect `\transpose`, `\chords` or `\relative` sections in its argument. If you want to use relative within transposed music, you must place an additional `\relative` inside the `\transpose`.

3.2.2 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, but a d’ is found. In the output, the octave is corrected for this and the following notes.

There is also a syntax that is separate from the notes.

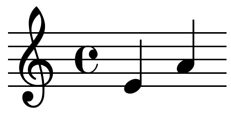
```
\octave pitch
```

This checks that *pitch* (without octave) yields *pitch* (with octave) in \relative mode. If not, a warning is printed, and the octave is corrected, for example, the first check is passed successfully. The second check fails with an error message. The octave is adjusted so 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. Hence, the `\octave` check may be deleted without changing the meaning of the piece.

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



3.2.3 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

```
\lyrics {
  \time 2/4
```

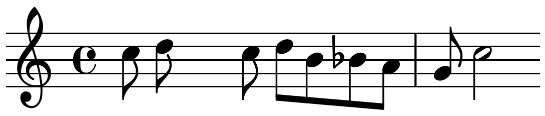
```
    Twin -- kle | Twin -- kle
  }
```

Failed bar checks are caused by entering incorrect durations. Incorrect durations often completely garble up the score, especially if it is polyphonic, so you should start correcting the score by scanning for failed bar checks and incorrect durations. To speed up this process, you can use `skipTypesetting`, described in the next section.

3.2.4 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

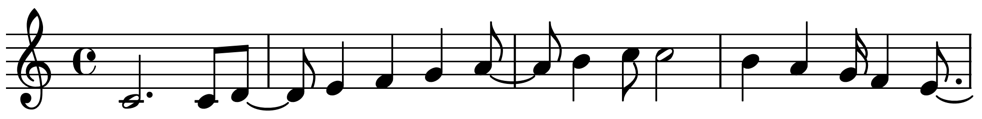
```
\relative c'' {
  c8 d
  \set Score.skipTypesetting = ##t
  e f g a g c, f e d
  \set Score.skipTypesetting = ##f
  c d b bes a g c2 }
```

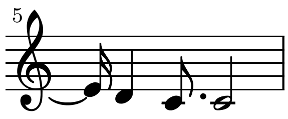


3.2.5 Automatic note splitting

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; the engraver will not insert tuplets.

See also

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

Program reference: `Completion_heads_engraver`.

3.3 Staff notation

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

3.3.1 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.

3.3.2 Key signature

The key signature indicates the scale 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.

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.

Bugs

The ordering of a key cancellation is wrong when it is combined with repeat bar lines. The cancellation is also printed after a line break.

See also

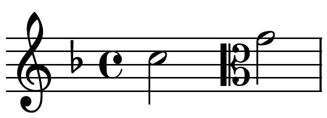
Program reference: `KeyChangeEvent`, and `KeySignature`.

3.3.3 Clef

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

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

```
\key f\major  c''2 \clef alto g'2
```

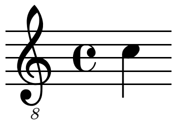


Supported clef-names include

`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), `centralCPosition` and `clefOctavation`. A clef is printed when any of these properties are changed.

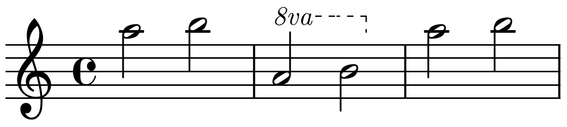
See also

Program reference: the object for this symbol is `Clef`.

3.3.4 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"
```

See also

Program reference: `OttavaBracket`.

Examples: '`input/regression/ottava.ly`', '`input/regression/ottava-broken.ly`'.

Bugs

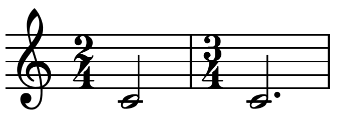
`set-octavation` will get confused when clef changes happen during an octavation bracket.

3.3.5 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.
```

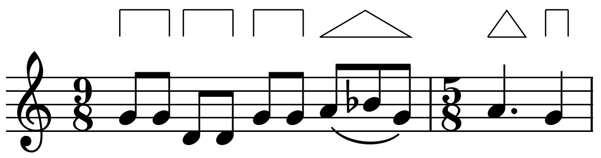


The symbol that is printed can be customized with the `style` property. Setting it to `#'()` uses fraction style for 4/4 and 2/2 time. There are many more options for its layout. See '`input/test/time.ly`' for more examples.

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 {
  \notes \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
  }
  \paper {
    \context {
      \StaffContext
      \consists "Measure_grouping_engraver"
    }
  }
}
```



See also

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

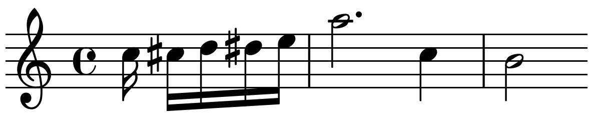
Bugs

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

3.3.6 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.

3.3.7 Unmetered music

Bar lines and bar numbers are calculated automatically. For unmetered music (e.g. cadenzas), 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

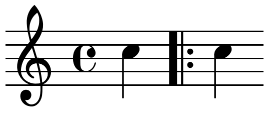
`\cadenzaOn`, `\cadenzaOff`.

3.3.8 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.

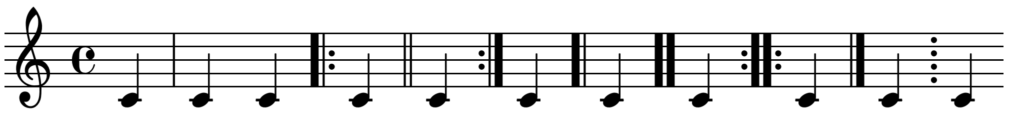
Special types of bar lines can be forced with the `\bar` command

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



The following bar types are available

```
c4
\bar "|" c
\bar "" c
\bar "|:" c
\bar "||" c
\bar ":|" c
\bar ".|" c
\bar ".|." c
\bar ":|:" c
\bar "|." c
\bar ":" c
```



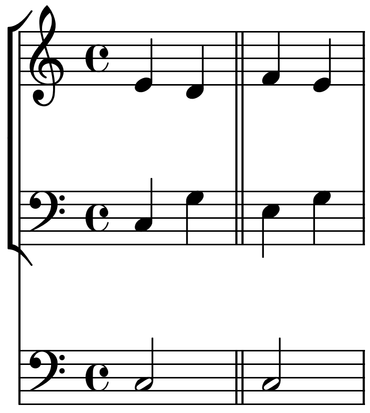
For allowing line breaks, there is a special command,

```
\bar "empty"
```

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 }
>>
```



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

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.

You are encouraged to use `\repeat` for repetitions. See Section 3.8 [Repeats], page 61.

See also

In this manual: Section 3.8 [Repeats], page 61.

Program reference: the bar line objects that are created at `Staff` level are called `BarLine`, the bar lines that span staves are `SpanBar` objects.

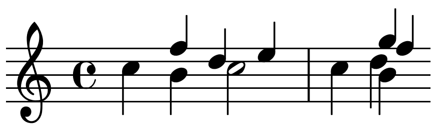
The bar lines at the start of each system are `SystemStartBar`, `SystemStartBrace`, and `SystemStartBracket`. Only one of these types is created in every context, and that type is determined by the property `systemStartDelimiter`.

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

3.4 Polyphony

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

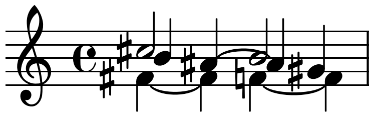
```
\context Staff \relative c'' {
  c4 << { f d e  } \\ { b c2 } >>
  c4 << g' \\ b, \\  f' \\ d >>
}
```



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

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  } >>
```

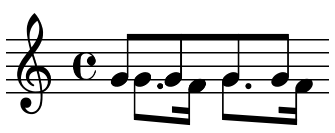


The command `\oneVoice` will revert back to the normal setting.

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

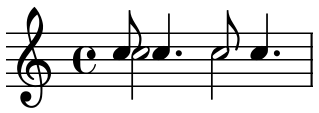
```
\context Voice << {
  g8 g8
  \override Staff.NoteCollision
    #'merge-differently-dotted = ##t
  g8 g8
} \\ { g8.[ f16] g8.[ f16] } >>
```

[2] Polyphonic voices are sometimes called "layers" other notation packages



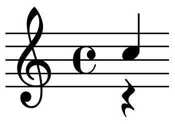
Similarly, you can merge half note heads with eighth notes, by setting `merge-differently-headed`

```
\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

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



Predefined commands

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

The following commands specify in what chords of the current voice should be shifted: the outer voice has `\shiftOff`, and the inner voices have `\shiftOn`, `\shiftOnn`, etc.

`\shiftOn`, `\shiftOnn`, `\shiftOnnn`, `\shiftOff`.

See also

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

Examples: See also example files '`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

Resolving collisions is a intricate subject, and only a few situations are handled. When LilyPond cannot cope, the `force-hshift` property of the `NoteColumn` object and pitched rests can be used to override typesetting decisions.

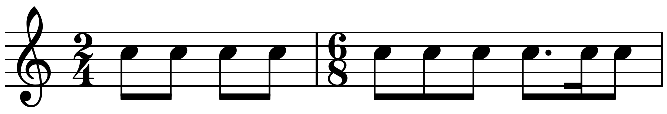
When using `merge-differently-headed` with an upstem eighth or a shorter note, and a downstem half note, the eighth note gets the wrong offset.

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 3.16.1 [Clusters], page 104).

3.5 Beaming

Beams are used to group short notes into chunks that are aligned with the metrum. They are inserted automatically

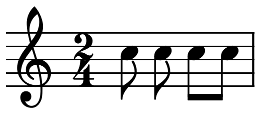
```
\time 2/4 c8 c c c \time 6/8 c c c c8. c16  c8
```



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
```



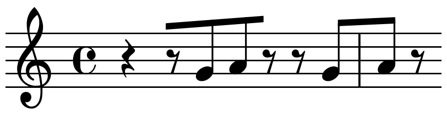
See also

Program reference: `Beam`.

3.5.1 Manual beams

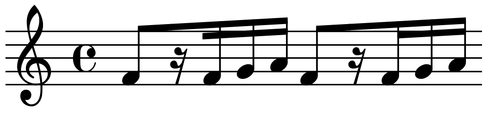
In some cases it may be necessary to override the automatic beaming algorithm. For example, the autobeamer will not put beams over rests or bar lines. Such beams are specified by manually: the begin and end point are marked with `[` and `]`

```
\context Staff {
  r4 r8[ g' a r8] r8 g[ | a] r8
}
```



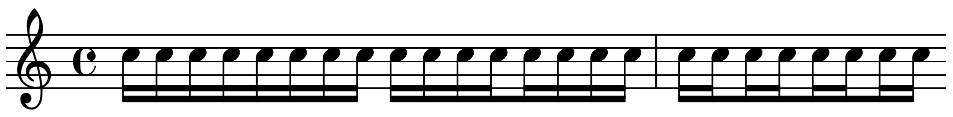
Normally, beaming patterns within a beam are determined automatically. If necessary, the properties `stemLeftBeamCount` and `stemRightBeamCount` can be used to override the defaults. If either property is set, its value will be used only once, and then it is erased

```
\context Staff {
  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]
```



Kneed beams are inserted automatically, when a large gap is detected between the note heads. This behavior can be tuned through the object property `auto-knee-gap`.

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

Bugs

Automatically kneed cross-staff beams cannot be used together with hidden staves.

3.5.2 Setting automatic beam behavior

In normal time signatures, automatic beams can start on any note but can only end in a few positions within the measure: beams can end on a beat, or at durations specified by the properties in `autoBeamSettings`. The defaults for `autoBeamSettings` are defined in '`scm/auto-beam.scm`'.

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

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

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.

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

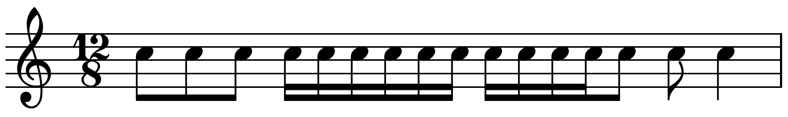
`\autoBeamOff`, `\autoBeamOn`.

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 eight beams and sixteenth end at 3 eights; 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`.

3.6 Accidentals

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

Common rules for typesetting accidentals have been canned in a function. This function is called as follows

```
#(set-accidental-style 'modern 'Voice)
```

The function takes two arguments: a symbol that denotes the style (in the example, `modern`), and another symbol that denotes the context name (in this example, `Voice`). If no context name is supplied, `Staff` is the default.

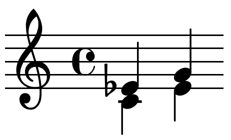
The following styles are supported

`default` This is the default typesetting behavior. It should correspond 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 `code`.

This leads to some weird and often unwanted results because accidentals from one voice do not get canceled in other voices

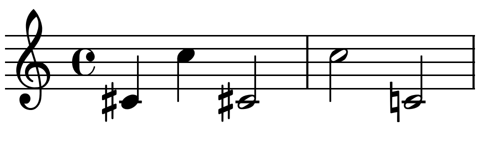
```
\context Staff <<
  #(set-accidental-style 'voice)
  <<
    { es g } \\
    { c, e }
>> >>
```



Hence you should only use `voice` 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 you use `modern` or `modern-cautionary` instead.

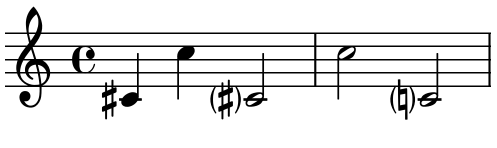
`modern` This rule corresponds to the common practice in the 20th century. This rule prints the same accidentals as `default`, but temporary accidentals also are canceled in other octaves. Furthermore, in the same octave, they also get canceled in the following measure

```
#(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`.

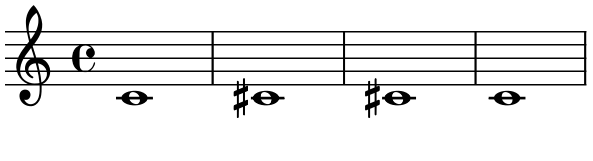
`modern-voice-cautionary`
This rule is the same as `modern-voice`, but with the extra accidentals (the ones not typeset by `voice`) typeset as cautionaries. Even though all accidentals typeset by `default` *are* typeset by this variable then some of them are typeset as cautionaries.

`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`
As `#(set-accidental-style 'piano)' , str)` but with the extra accidentals typeset as cautionaries.

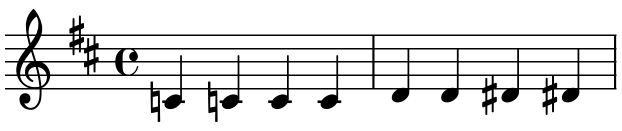
`no-reset` This is the same as `default` but with accidentals lasting "forever" and not only until the next measure

```
#(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 only a problem when accidentals in a chord depend on each other. This problem can be solved by manually inserting `!` and `?` for the problematic notes.

In the default scheme, accidentals only depend on other accidentals with the same pitch on the same staff, so no conflicts are possible.

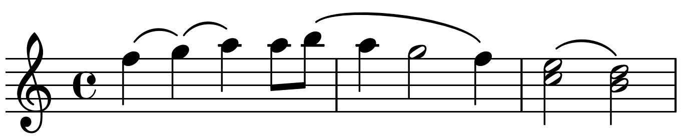
3.7 Expressive marks

3.7.1 Slurs

A slur indicates that notes are to be played bound or *legato*.

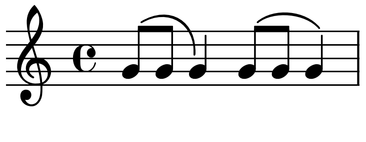
They are entered using parentheses

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



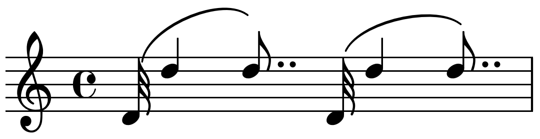
Slurs avoid crossing stems, and are generally attached to note heads. However, in some situations with beams, slurs may be attached to stem ends. If you want to override this layout you can do this through the object property `attachment` of `Slur`. Its value is a pair of symbols, specifying the attachment type of the left and right end points

```
\slurUp
\override Stem #'length = #5.5
g'8(g g4)
\override Slur #'attachment = #'(stem . stem)
g8( g g4)
```



If a slur would strike through a stem or beam, the slur will be moved away upward or downward. If this happens, attaching the slur to the stems might look better

```
\stemUp \slurUp
d32( d'4 d8..)
\override Slur #'attachment = #'(stem . stem)
d,32( d'4 d8..)
```



Predefined commands

`\slurUp`, `\slurDown`, `\slurBoth`, `\slurDotted`, `\slurSolid`.

See also

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

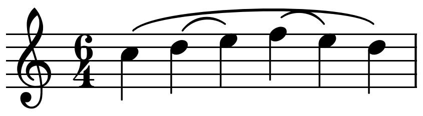
Bugs

Producing nice slurs is a difficult problem, and LilyPond uses a simple, empiric method to produce slurs. In some cases, its results are ugly.

3.7.2 Phrasing slurs

A phrasing slur (or phrasing mark) connects chords and is used to indicate a musical sentence. It is started 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, you should use `\phrasingSlurUp`, `\phrasingSlurDown`, and `\phrasingSlurBoth`.

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

Predefined commands

`\phrasingSlurUp`, `\phrasingSlurDown`, `\phrasingSlurBoth`.

See also

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

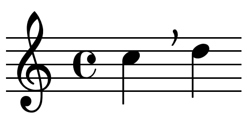
Bugs

Phrasing slurs have the same limitations in their formatting as normal slurs. Putting phrasing slurs over rests leads to spurious warnings.

3.7.3 Breath marks

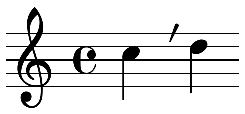
Breath marks are entered using `\breathe`

```
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`'.

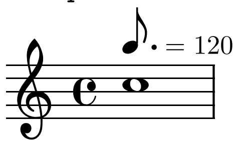
3.7.4 Metronome marks

Metronome settings can be entered as follows

```
    \tempo duration = per-minute
```

In the MIDI output, they are interpreted as a tempo change, and in the paper output, a metronome marking is printed

```
\tempo 8.=120 c''1
```



See also

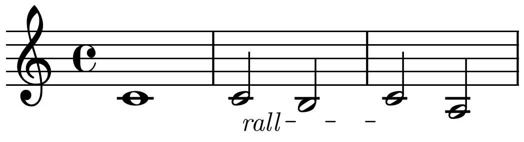
Program reference: `MetronomeChangeEvent`.

3.7.5 Text spanners

Some performance indications, e.g. *rallentando* or *accelerando*, are written as texts, and extended over many measures with dotted lines. You can create such texts using text spanners: attach `\startTextSpan` and `\stopTextSpan` to the start and ending note of the spanner.

The string to be printed, as well as the style, is set through object properties

```
\relative c' {
  c1
  \override TextSpanner #'direction = #-1
  \override TextSpanner #'edge-text = #'("rall " . "")
  c2\startTextSpan b c\stopTextSpan a
}
```



See also

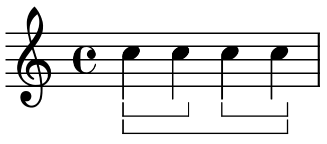
Internals `TextSpanEvent`, `TextSpanner`.

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

3.7.6 Analysis brackets

Brackets are used in musical analysis to indicate structure in musical pieces. LilyPond supports a simple form of nested horizontal brackets. To use this, add the `Horizontal_bracket_engraver` to `Staff` context. A bracket is started with `\startGroup` and closed with `\stopGroup`

```
\score {
  \notes \relative c'' {
    c4\startGroup\startGroup
    c4\stopGroup
    c4\startGroup
    c4\stopGroup\stopGroup
  }
  \paper {
    \context {
      \StaffContext \consists "Horizontal_bracket_engraver"
}}}
```



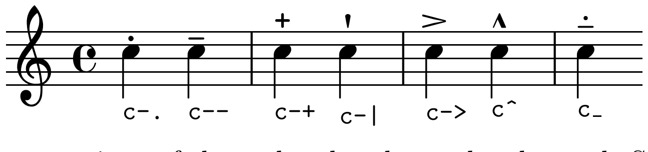
See also

Program reference: `HorizontalBracket`, `NoteGroupingEvent`.

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

3.7.7 Articulations

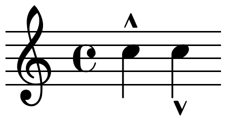
A variety of symbols can appear above and below notes to indicate different characteristics of the performance. They are added to a note by adding a dash and the character signifying the articulation. They are demonstrated here



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

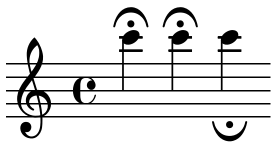
The script is automatically placed, but if you need to force directions, you can use `_` to force them down, or `^` to put them up

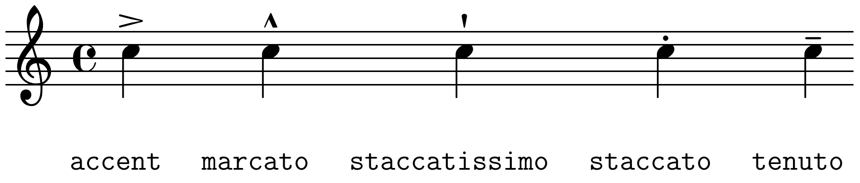
```
c''4^^ c''4_^
```

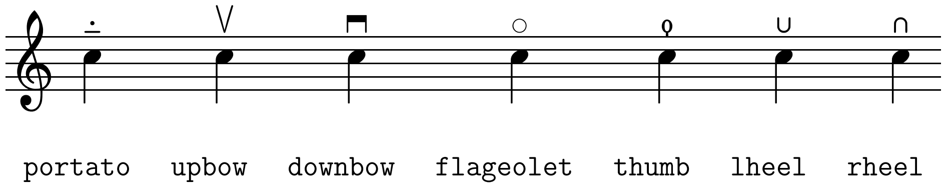


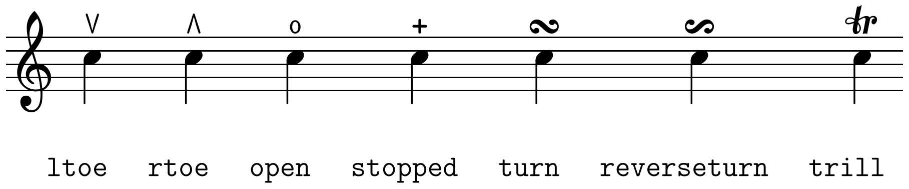
Other symbols can be added using the syntax *note**name*, e.g. `c4\fermata`. Again, they can be forced up or down using `^` and `_`, e.g.

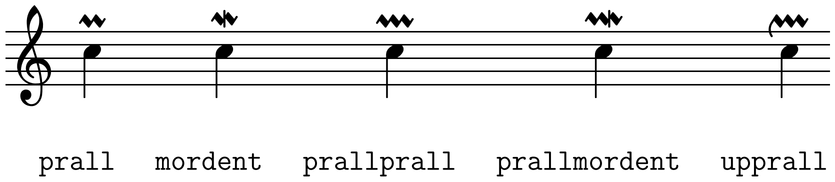
```
c\fermata c^\fermata c_\fermata
```

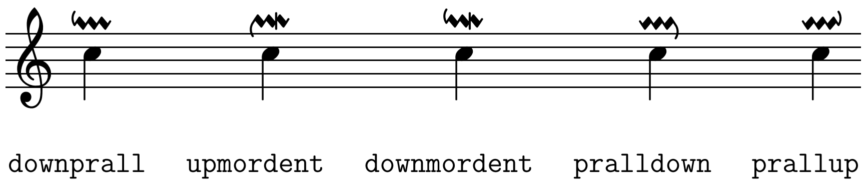


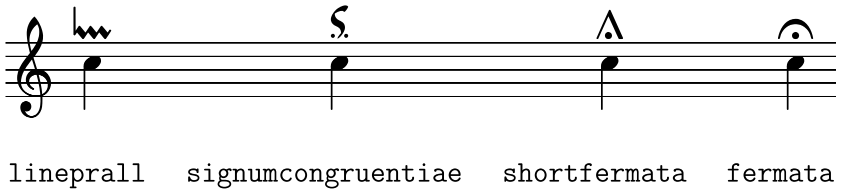

accent
marcato
staccatissimo
staccato
tenuto

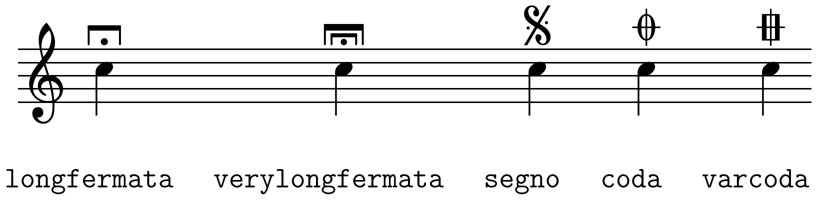

portato
upbow
downbow
flageolet
thumb
lheel
rheel


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

Predefined commands

`\scriptUp`, `\scriptDown`, `\scriptBoth`.

See also

Program reference: `ScriptEvent`, and `Script`.

Bugs

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

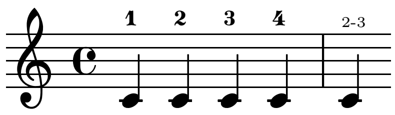
3.7.8 Fingering instructions

Fingering instructions can be entered using

```
    note-digit
```

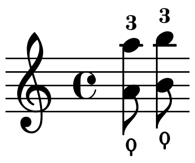
For finger changes, use markup texts

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



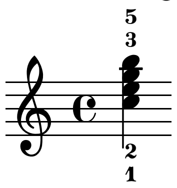
You can use the thumb-script to indicate that a note should be played with the thumb (e.g. in cello music)

```
<a' a''-3>8_\thumb <b' b''-3>_\thumb
```



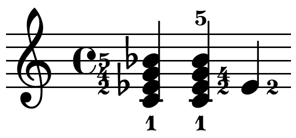
Fingerings for chords can also be added to individual notes of the chord by adding them after the pitches

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



iIn 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
\set fingeringOrientations = #'(right)
<es-2>4
```



The last note demonstrates how fingering instructions can be put close to note heads in monophonic music, using this feature.

See also

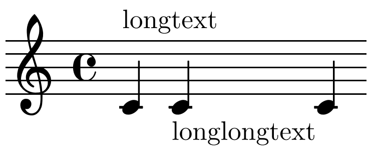
Program reference: `FingerEvent`, and `Fingering`.

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

3.7.9 Text scripts

It is possible to place arbitrary strings of text or markup text (see Section 4.5 [Text markup], page 123) 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

```
\relative c' {
  c4^"longtext" \fatText c4_"longlongtext" c4
}
```



It is possible to use TEX commands in the strings, but this should be avoided because the exact dimensions of the string can then no longer be computed.

Predefined commands

`\fatText`, `\emptyText`.

See also

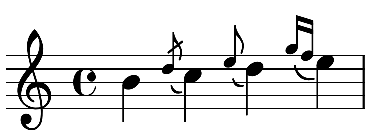
In this manual: Section 4.5 [Text markup], page 123.

Program reference: `TextScriptEvent`, `TextScript`.

3.7.10 Grace notes

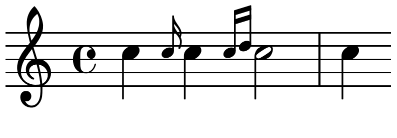
Grace notes are ornaments that are written out. The most common ones are acciaccatura, which should be played as very short. It is denoted by a slurred small note with a slashed stem. The appoggiatura is a grace note that takes a fixed fraction of the main note, is and denoted as a slurred note in small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



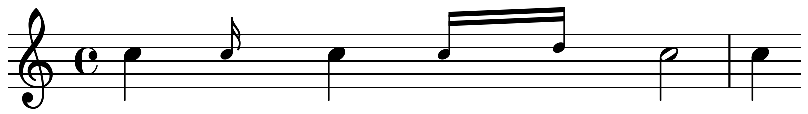
Both are special forms of the `\grace` command. By prefixing this keyword to a music expression, a new one is formed, which will be printed in a smaller font and takes up no logical time in a measure.

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



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

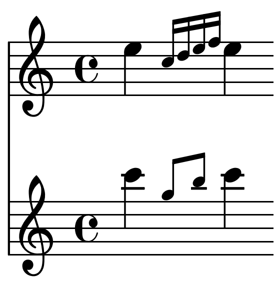
Internally, timing for grace notes is done using a second, ‘grace’ time. Every point in time consists of two rational numbers: one denotes the logical time, one denotes the grace timing. The above example is shown here with timing tuples



$$\mathbf{(0,0)\ (\tfrac{1}{4},\tfrac{-1}{16})\ (\tfrac{1}{4},0)\ (\tfrac{2}{4},\tfrac{-1}{8})\ (\tfrac{2}{4},\tfrac{-1}{16})\ (\tfrac{2}{4},0)}$$

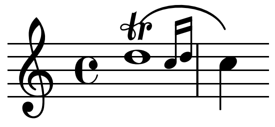
The placement of grace notes is synchronized between different staves. In the following example, there are two sixteenth graces notes for every eighth grace note

```
<< \new Staff { e4 \grace { c16[ d e f] } e4 }
   \new Staff { c'4 \grace { g8[ b] } c4 } >>
```



If you want to end a note with a grace, the standard trick is to put the grace notes after a “space note”

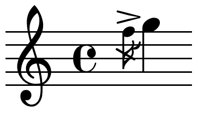
```
\context Voice {
  << { d1^\trill ( }
     { s2 \grace { c16[ d] } } >>
  c4)
}
```



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 {
    \override Stem #'direction = #-1
    f16->
    \revert Stem #'direction
  }
  g4
}
```



The overrides should also be reverted inside the grace section.

If the layout of grace sections must be changed throughout the music, then this can be accomplished through 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`, `startAccacciaturaMusic`, `stopAccacciaturaMusic`, `startAppoggiaturaMusic`, `stopAppoggiaturaMusic`. More information is in the file '`ly/grace-init.ly`'.

See also

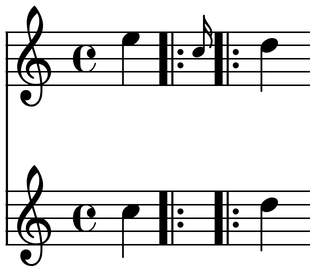
Program reference: `GraceMusic`.

Bugs

A score that starts with an `\grace` section needs an explicit `\context Voice` declaration, otherwise the main note and 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 } >>
```

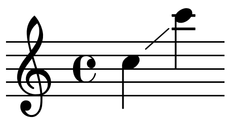
Grace sections should only be used within sequential music expressions. Nesting or juxtaposing grace sections is not supported, and might produce crashes or other errors.

3.7.11 Glissando

A glissando is a smooth change in pitch. It is denoted by a line or a wavy line between two notes.

A glissando line can be requested by attaching a `\glissando` to a note

```
c'\glissando c'
```



See also

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

Example files: ‘`input/regression,glissando.ly`’

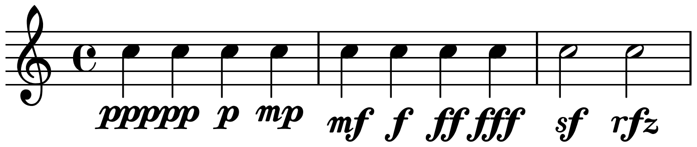
Bugs

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

3.7.12 Dynamics

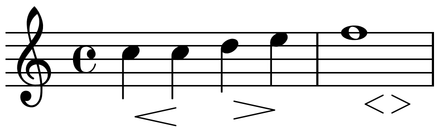
Absolute dynamic marks are specified using a command after a note `c4\ff`. The available dynamic marks are `\ppp`, `\pp`, `\p`, `\mp`, `\mf`, `\f`, `\ff`, `\fff`, `\fff`, `\fp`, `\sf`, `\sff`, `\sp`, `\spp`, `\sfz`, and `\rfz`

```
c'\ppp c\pp c \p c\mp c\mf c\f c\ff c\fff
c2\sf 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, if you must use spacer notes if multiple marks during one note are needed

```
c''\< c''\! d''\> e''\!
<< f''1 { s4 s4\< s4\! \> s4\! } >>
```

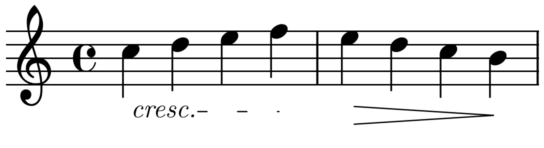


This may give rise to very short hairpins. Use `minimum-length` in `Voice.Hairpin` to lengthen them, for example

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

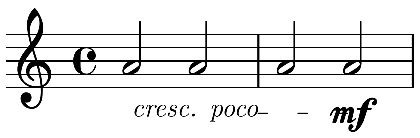
You can also use a text saying *cresc.* instead of hairpins. Here is an example how to do it

```
\setTextCresc
c \< d e f\!
\setHairpinCresc
e\> d c b\!
```



You can also supply your own texts

```
\context Voice {
  \set crescendoText = \markup { \italic "cresc. poco" }
  \set crescendoSpanner = #'dashed-line
  a'2\< a a a\!\mf
}
```



Predefined commands

`\dynamicUp`, `\dynamicDown`, `\dynamicBoth`.

See also

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

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

3.8 Repeats

Repetition is a central concept in music, and multiple notations exist for repetitions. In LilyPond, most of these notations can be captured in a uniform syntax. One of the advantages is that they can be rendered in MIDI accurately.

The following types of repetition are supported

`unfold`	Repeated music is fully written (played) out. Useful for MIDI output, and entering repetitive music.
`volta`	This is the normal notation: Repeats are not written out, but alternative endings (volte) are printed, left to right.
`tremolo`	Make tremolo beams.
`percent`	Make beat or measure repeats. These look like percent signs.

3.8.1 Repeat syntax

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

```
\repeat variant repeatcount repeatbody
```

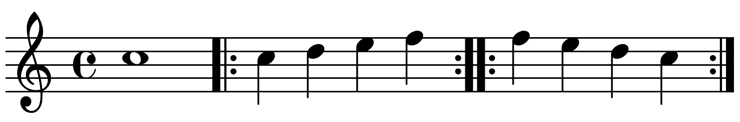
If you have alternative endings, you may add

```
\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.

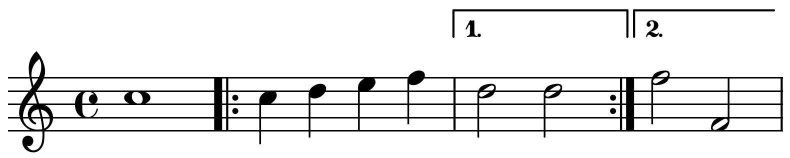
Normal notation repeats are used like this

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

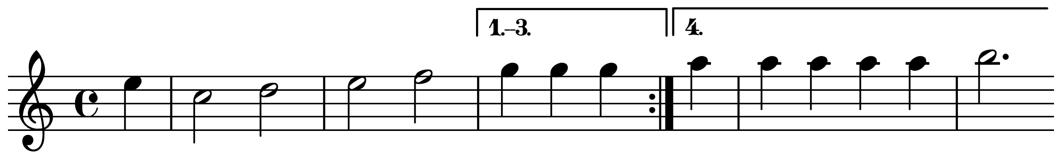


With alternative endings

```
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. } }
}
```



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.

3.8.2 Repeats and MIDI

For instructions on how to expand repeats for MIDI output, see the example file '`input/test/unfold-all-repeats.ly`'.

Bugs

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.

3.8.3 Manual repeat commands

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

the symbol `start-repeat`,
: which prints a `|:` bar line,

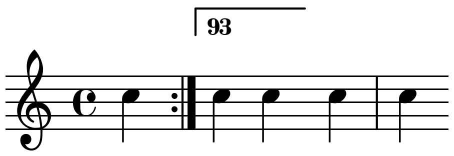
the symbol `end-repeat`,
: which prints a `:|` bar line,

the list `(volta` *`text`*`)`,
: which prints a volta bracket saying *text*: The text can be specified as a text string or as a markup text, see Section 4.5 [Text markup], page 123. Do not forget to change the font, as the default number font does not contain alphabetic characters. Or,

the list `(volta #f)`, which
: stops a running volta bracket

```
c4
  \set Score.repeatCommands = #'((volta "93") end-repeat)
```

```
c4 c4
  \set Score.repeatCommands = #'((volta #f))
c4 c4
```



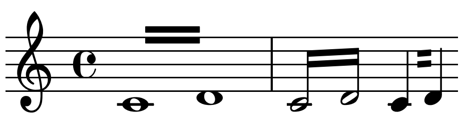
See also

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

3.8.4 Tremolo repeats

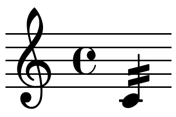
To place tremolo marks between notes, use `\repeat` with tremolo style

```
\score {
  \context Voice \notes\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
```



A similar mechanism is the tremolo subdivision, described in Section 3.8.5 [Tremolo subdivisions], page 63.

See also

In this manual: Section 3.8.5 [Tremolo subdivisions], page 63, Section 3.8 [Repeats], page 61.

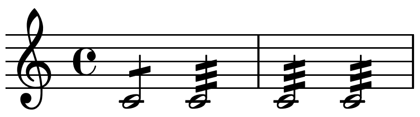
Program reference: tremolo beams are `Beam` objects. Single stem tremolos are `StemTremolo` objects. The music expression is `TremoloEvent`.

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

3.8.5 Tremolo subdivisions

Tremolo marks can be printed on a single note by adding '`:`[*length*]' after the note. The length 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

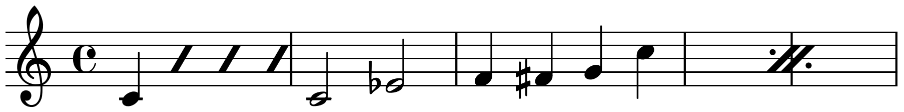
In this manual: Section 3.8.4 [Tremolo repeats], page 63.

Elsewhere: `StemTremolo`, `TremoloEvent`.

3.8.6 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 a one and two measures are replaced by percent-like signs, patterns that divide the measure length are replaced by slashes

```
\context Voice { \repeat  "percent" 4  { c'4 }
  \repeat "percent" 2 { c'2 es'2 f'4 fis'4 g'4 c''4 }
}
```



See also

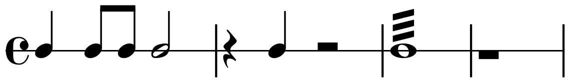
Program reference: `RepeatSlash`, `PercentRepeat`, `PercentRepeatedMusic`, and `DoublePercentRepeat`.

3.9 Rhythmic music

3.9.1 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

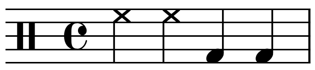
Program reference: `RhythmicStaff`.

Examples: '`input/regression/rhythmic-staff.ly`'.

3.9.2 Entering percussion

Percussion notes may be entered in `\drums` mode, which is similar to `notes`. Each piece of percussion has a full name and an abbreviated name, and both be used in input files

```
hihat hh bassdrum bd
```



The complete list of drum names is in the init file 'ly/drumpitch-init.ly'.

See also

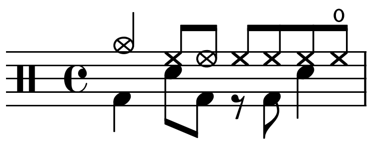
Program reference: `DrumNoteEvent`.

3.9.3 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.

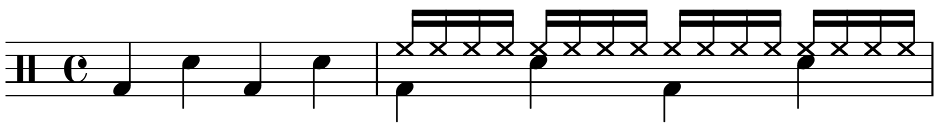
To typeset the music, the notes must be interpreted in a `DrumStaff` and `DrumVoice` contexts

```
up = \drums { crashcymbal4 hihat8 halfopenhihat hh hh hh openhihat }
down = \drums { bassdrum4 snare8 bd r bd sn4 }
\score {
  \new DrumStaff <<
    \new DrumVoice { \voiceOne \up }
    \new DrumVoice { \voiceTwo \down }
  >>
}
```



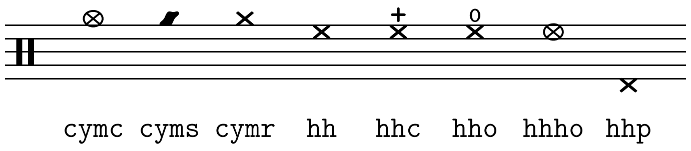
The above example shows verbose polyphonic notation. The short polyphonic notation, described in Section 3.4 [Polyphony], page 45, can also be used if the `DrumVoices` are instantiated by hand first. For example,

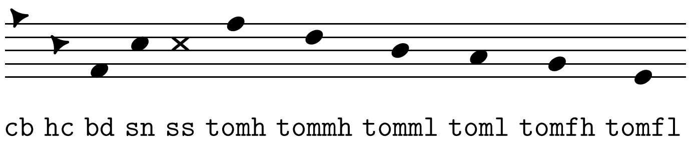
```
\drums \new DrumStaff <<
  \context DrumVoice = "1" {  s1 *2 }
  \context DrumVoice = "2" {  s1 *2 }
  {
    bd4 sn4 bd4 sn4
    <<
      { \repeat unfold 16 hh16 }
      \\
      { bd4 sn4 bd4 sn4 }
    >>
  }
>>
```



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

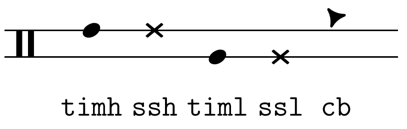
`drums-style`
is the default. It typesets a typical drum kit on a five-line staff



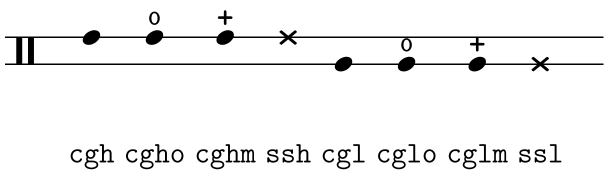


The drum scheme supports six different toms. When there fewer toms, simply select the toms that produce the desired result, i.e. to get toms on the three middle lines you use `tommh`, `tomml` and `tomfh`.

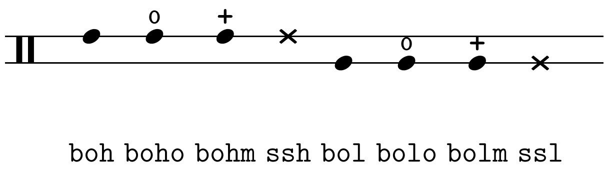
`timbales-style`
: to typeset timbales on a two line staff



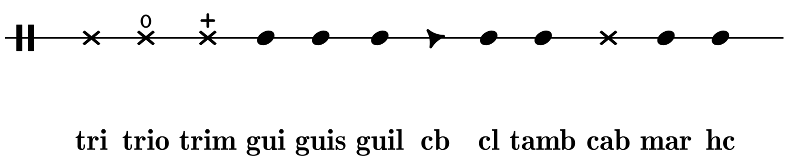
`congas-style`
: to typeset congas on a two line staff



`bongos-style`
: to typeset bongos on a two line staff

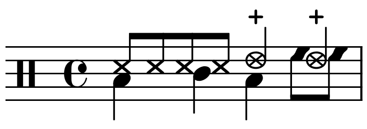


`percussion-style`
: to typeset all kinds of simple percussion on one line staves



If you do not like any of the predefined lists you can define your own list at the top of your file

```
#(define mydrums '(
         (bassdrum     default   #f         -1)
         (snare        default   #f         0)
         (hihat        cross     #f         1)
         (pedalhihat   xcircle   "stopped" 2)
         (lowtom       diamond   #f       3)))
 up = \drums { hh8 hh hh hh hhp4 hhp }
 down = \drums { bd4 sn bd toml8 toml }
 \score {
   \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.

3.10 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

There is no support for putting chords across staves. You can get this result by increasing the length of the stem in the lower stave so it reaches the stem in the upper stave, or vice versa. An example is included with the distribution as 'input/test/stem-cross-staff.ly'.

Dynamics are not centered, but kludges do exist. See 'input/template/piano-dynamics.ly'.

The distance between the two staves is normally fixed across the entire score. It is possible to tune this per system, but it does require arcane command incantations. See 'input/test/piano-staff-distance.ly'.

3.10.1 Automatic staff changes

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

```
\autochange \context Voice { ...music... }
```

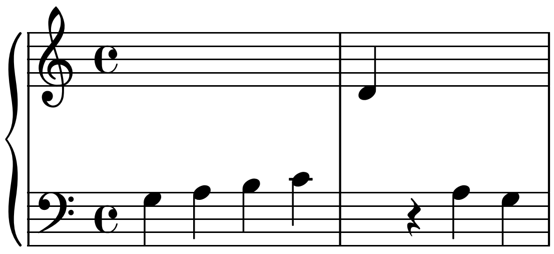
The two staves of the piano staff must be named up and down.

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 ... \new Voice ...
```

The autochanger switches on basis of pitch (middle C is the turning point), and it looks ahead skipping over rests to switch in advance. Here is a practical example

```
\score { \notes \context PianoStaff <<
  \context Staff = "up" {
    \autochange \new Voice \relative c' {
       g4 a  b c d r4 a g } }
  \context Staff = "down" {
       \clef bass
       s1*2
} >> }
```



In this example, spacer rests are used to prevent the bottom staff from terminating too soon.

See also

In this manual: Section 3.10.2 [Manual staff switches], page 68.

Program reference: `AutoChangeMusic`.

Bugs

The staff switches often do 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.

3.10.2 Manual staff switches

Voices can be switched between staves manually, using the following 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

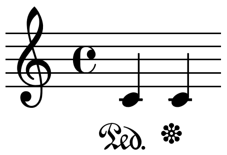
```
\context Staff = down
  \new Voice { ... \change Staff = up ... }
```

3.10.3 Pedals

Pianos have pedals that alter the way sound are 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

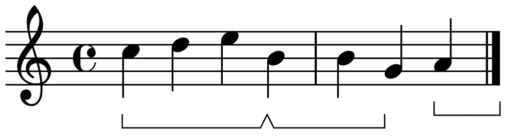
```
c'4\sustainDown c'4\sustainUp
```



What is printed can be modified by setting `pedalXStrings`, where *X* is one of the pedal types: `Sustain`, `Sostenuto` or `UnaCorda`. Refer to `SustainPedal` in the program reference for more information.

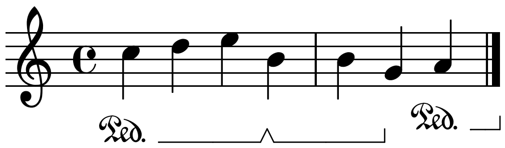
Pedals can also be indicated by a sequence of brackets, by setting the `pedalSustainStyle` property to `bracket` objects

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



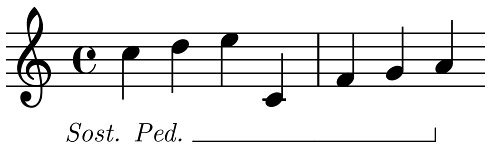
A third style of pedal notation is a mixture of text and brackets, obtained by setting the `pedalSustainStyle` style property to `mixed`

```
\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.

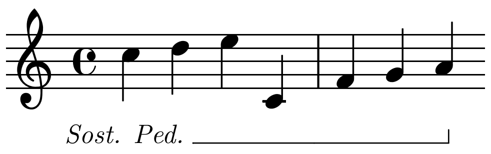
```
c\sostenutoDown d e c, f g a\sostenutoUp
```



For fine-tuning of 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 end of the note head

```
\override Staff.PianoPedalBracket
   #'shorten-pair = #'(0 . -1.0)
```

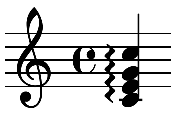
```
c\sostenutoDown d e c, f g a\sostenutoUp
```



3.10.4 Arpeggio

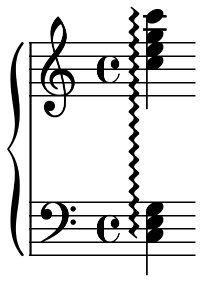
You can specify an arpeggio sign on a chord by attaching an `\arpeggio` to a chord

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



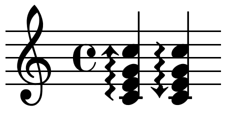
When an arpeggio crosses staves, you attach an arpeggio to the chords in both staves, and set `PianoStaff.connectArpeggios`

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



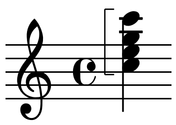
The direction of the arpeggio is sometimes denoted by adding an arrowhead to the wiggly line

```
\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`, `\arpeggioUp`, `\arpeggioBoth`, `\arpeggioBracket`.

See also

Program reference: `ArpeggioEvent` music expressions lead to `Arpeggio` objects. Cross staff arpeggios are `PianoStaff.Arpeggio`.

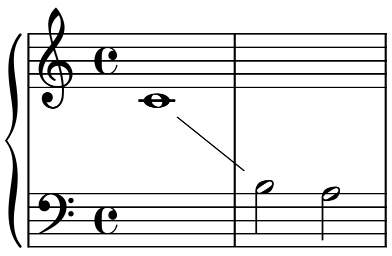
Bugs

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

3.10.5 Staff switch lines

Whenever a voice switches to another staff a line connecting the notes can be printed automatically. This is enabled if the property `PianoStaff.followVoice` is set 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

The associated object is `VoiceFollower`.

Predefined commands

`\showStaffSwitch`, `\hideStaffSwitch`.

3.11 Vocal music

This section discusses how to enter and print lyrics.

3.11.1 Entering lyrics

Lyrics are entered in a special input mode. This mode is is introduced by the keyword `\lyrics`. In this mode you can enter lyrics, with punctuation and accents without any hassle. Syllables are entered like notes, but with pitches replaced by text. For example,

```
\lyrics { Twin-4 kle4 twin- kle litt- le star2 }
```

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 `^`.

Subsequent characters of a word can be any character that is not a digit and not white space. One important consequence of this is that a word can end with `}`. The following example is usually a bug. The syllable includes a `}`, and hence the opening brace is not balanced

```
\lyrics { twinkle}
```

Similarly, a period following a 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 which 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

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

However, at least for english texts, you should use

```
\lyrics { He said: ``Let my peo ple go'' }
```

to get the correct shape of the starting and ending quote.

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: events `LyricEvent`, `HyphenEvent`, and `ExtenderEvent`. Objects `LyricHyphen`, `LyricExtender` and `LyricText`.

Examples: '`input/test/lyric-hyphen-retain.ly`'.

Bugs

The definition of lyrics mode is too complex.

3.11.2 The Lyrics context

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

```
\context Lyrics \lyrics ...
```

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`.

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

```
\context ChoirStaff \notes <<
  \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

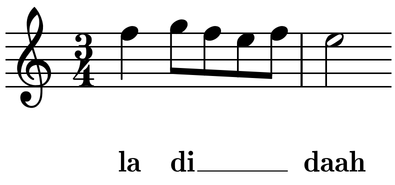
```
\lyricsto "soprano" \context Lyrics = sopranoLyrics
   the lyrics
```

The final input would resemble

```
<< \context ChoirStaff \notes << 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 \lyrics {
    la di __ daah
  }
>>
```



In addition, notes are considered a melisma if they are manually beamed, and automatic beaming (see Section 3.5.2 [Setting automatic beam behavior], page 48) is switched off. The criteria for deciding melismata can be tuned with the property `melismaBusyProperties`. See `Melisma_translator` in the program reference for more information.

When multiple stanzas are put on the same melody, it can happen that two stanzas have melismata in different locations. This can be remedied by switching off melismata for one `Lyrics`. This is achieved by setting the `ignoreMelismata` property to `#t`. An example is shown in ‘`input/regression/lyric-combine-new.ly`’.

A complete example of a SATB score setup is in the file ‘`input/template/satb.ly`’.

Predefined commands

`\melisma`, `\melismaEnd`

See also

Program reference: Music expressions: `LyricCombineMusic`, Contexts: `Lyrics`, `Melisma_translator`.

Examples: ‘`input/template/satb.ly`’, ‘`input/regression/lyric-combine-new.ly`’.

Bugs

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

For proper processing of extender lines, the `Lyrics` and `Voice` should be linked. This can be achieved either by using `\lyricsto` or by setting corresponding names for both contexts. The latter is explained in Section 3.11.3 [More stanzas], page 74.

3.11.3 More stanzas

The lyrics should be aligned with the note heads of the melody. To achieve this, each line of lyrics should be marked to correspond with the melodic line. This is done automatically when `\lyricsto`, but it can also be done manually.

To this end, give the `Voice` context an identity

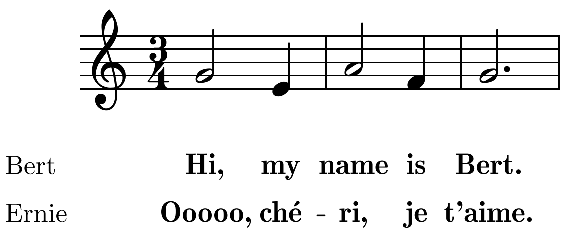
```
\context Voice = duet {
     \time 3/4
     g2 e4 a2 f4 g2.  }
```

Then set the `Lyrics` contexts to names starting with that identity followed by a dash. In the preceding example, the `Voice` identity is `duet`, so the identities of the `Lyrics`s are marked `duet-1` and `duet-2`

```
  \context Lyrics = "duet-1" {
    Hi, my name is Bert. }
  \context Lyrics = "duet-2" {
    Ooooo, ch\'e -- ri, je t'aime. }
```

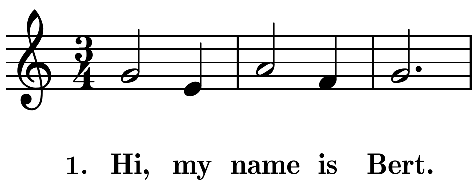
The complete example is shown here

```
\score {
  <<
    \notes \relative c'' \context Voice = duet {
      \time 3/4
       g2 e4 a2 f4 g2. }
    \lyrics <<
      \lyricsto "duet" \new Lyrics {
        \set vocalName = "Bert"
        Hi, my name is Bert. }
      \lyricsto "duet" \new Lyrics {
        \set vocalName = "Ernie"
        Ooooo, ch\'e -- ri, je t'aime. }
    >>
  >>
}
```



Stanza numbers can be added by setting `stanza`, e.g.

```
<<
  \context Voice = duet {
    \time 3/4 g2 e4 a2 f4 g2. }
   \lyrics \lyricsto "duet" \new Lyrics {
        \set stanza = "1. "
        Hi, my name is Bert. }
>>
```



This example also demonstrates how names of the singers can be added using `vocalName` analogous to instrument annotations for staves. A short version may be entered as `vocNam`.

To make empty spaces in lyrics, use `\skip`.

See also

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

Bugs

Input for lyrics introduces a syntactical ambiguity

```
foo = bar
```

is interpreted as assigning a string identifier `\foo` such that it contains `"bar"`. However, it could also be interpreted as making or a music identifier `\foo` containing the syllable 'bar'. The force the latter interpretation, use

```
foo = \lyrics bar4
```

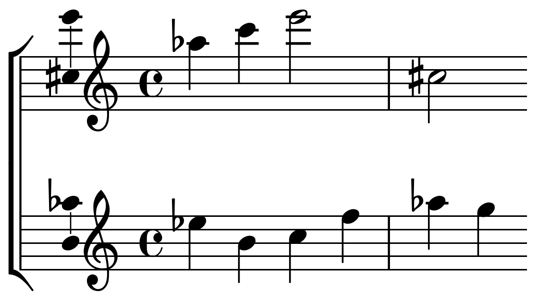
3.11.4 Ambitus

The term *ambitus* denotes a range of pitches for a given voice in a part of music. It also may denote the pitch range that a musical instrument is capable of playing. Ambituses are printed on vocal parts, so singers can easily determine if it meets his or her capabilities.

It 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 ambituses, add the `Ambitus_engraver` to the `Voice` context, for example,

```
\paper {
  \context {
    \VoiceContext
    \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.

It is possible to tune individual ambituses for multiple voices on a single staff, for example by erasing or shifting them horizontally. An example is in '`input/test/ambitus-mixed.ly`'.

See also

Program reference: `Ambitus`.

Examples: '`input/regression/ambitus.ly`', '`input/test/ambitus-mixed.ly`'.

Bugs

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

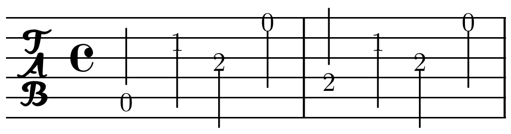
3.12 Tablatures

Tablature notation is used for notating music for plucked string instruments. It notates pitches not by using note heads, but by indicating on which string and fret a note must be played. LilyPond offers limited support for tablature.

3.12.1 Tablatures basic

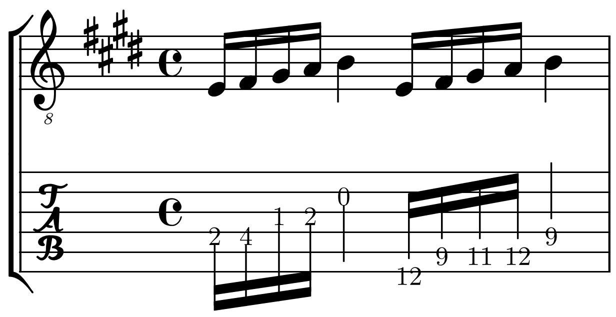
The string number associated to a note is given as a backslash followed by a number, e.g. `c4\3` for a C quarter on the third string. By default, string 1 is the highest one, and the tuning defaults to the standard guitar tuning (with 6 strings). The notes are printed as tablature, by using `TabStaff`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

```
\notes \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

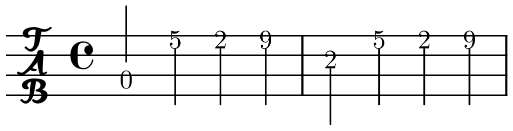
Chords are not handled in a special way, and hence the automatic string selector may easily select the same string to two notes in a chord.

3.12.2 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. Thus, the notes are e, a, d, and g

```
\context TabStaff <<
  \set TabStaff.stringTunings = #'(-5 -10 -15 -20)

  \notes {
    a,4 c' a e' e c' a e'
  }
>>
```



Bugs

No guitar special effects have been implemented.

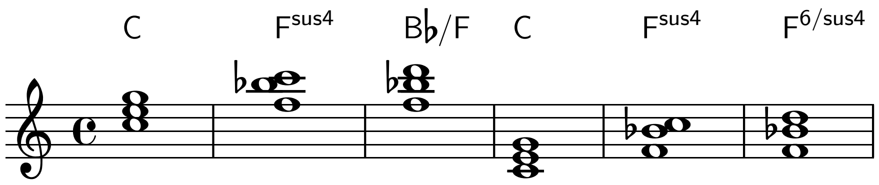
See also

Program reference: `Tab_note_heads_engraver`.

3.13 Chord names

LilyPond has support for both 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 = \notes \transpose c c' {
  \chords {
    c1 f:sus4 bes/f
  }
  <c e g>
  <f bes c'>
  <f bes d'>
}

\score {
  << \context ChordNames \twoWays
     \context Voice \twoWays >> }
```

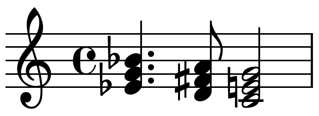


This example also shows that the chord printing routines do not try to be intelligent. The last chord (`f bes d`) is not interpreted as an inversion.

3.13.1 Chords mode

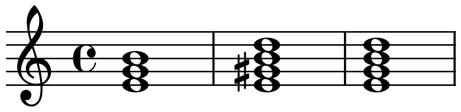
Chord mode is a mode where you can input sets of pitches using common names. It is introduced by the keyword `\chords`. In chords mode, a chord is entered by the root, which is entered like a common pitch

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



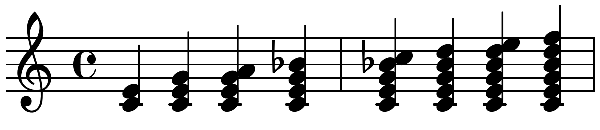
Other chords may be entered by suffixing a colon, and introducing a modifier, and optionally, a number

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



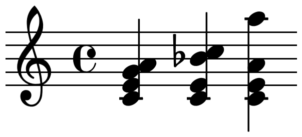
The first number following the root is taken to be the 'type' of the chord, thirds are added to the root until it reaches the specified number

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



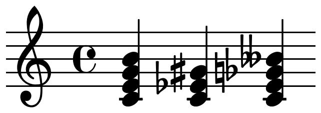
More complex chords may also be constructed adding separate steps to a chord. Additions are added after the number following the colon, and are separated by dots

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



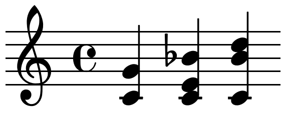
Chord steps can be altered by suffixing a `-` or `+` sign to the number

```
\chords { 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

```
\chords { c^3 c:7^5 c:9^3.5 }
```



Modifiers can be used to change pitches. The following modifiers are supported

`m` is the minor chord. This modifier lowers the 3rd and (if present) the 7th step.

`dim` is the diminished chord. This modifier lowers the 3rd, 5th and (if present) the 7th step.

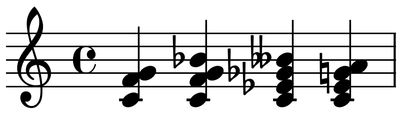
`aug` is the augmented chord. This modifier raises the 5th step.

`maj` is the major 7th chord. This modifier raises the 7th step if present.

`sus` is 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.

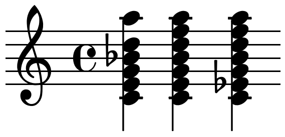
Modifiers can be mixed with additions

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



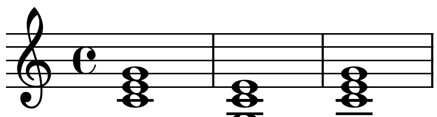
Since an unaltered 11 does not sound good when combined with an unaltered 13, the 11 is removed in this case (unless it is added explicitly)

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



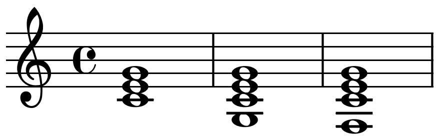
An inversion (putting one pitch of the chord on the bottom), as well as bass notes, can be specified by appending `/`*pitch* to the chord

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



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

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

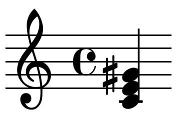


Chords is a mode similar to `\lyrics`, `\notes` etc. Most of the commands continue to work, for example, `r` and `\skip` can be used to insert rests and spaces, and property commands may be used to change various settings.

Bugs

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

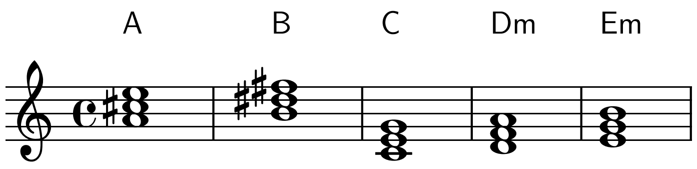
```
\chords { c:5.5-.5+ }
```



3.13.2 Printing chord names

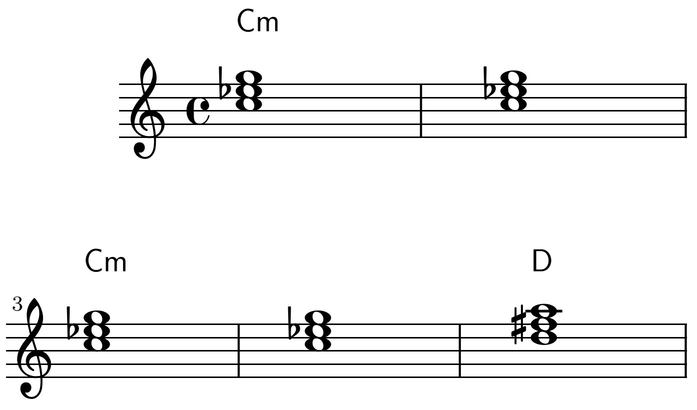
For displaying printed chord names, use the `ChordNames` context. The chords may be entered either using the notation described above, or directly using `<` and `>`

```
scheme = \notes {
  \chords {a1 b c} <d' f' a'>  <e' g' b'>
}
\score {
  \notes <<
    \context ChordNames \scheme
    \context Staff \scheme
  >>
}
```



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

```
scheme = \chords {
  c1:m c:m \break c:m c:m d
}
\score {
  \notes <<
    \context ChordNames {
      \set chordChanges = ##t
      \scheme }
    \context Staff \transpose c c' \scheme
  >>
}
```



The default chord name layout is a system for Jazz music, proposed by Klaus Ignatzek (see Appendix B [Literature list], page 182). It can be tuned through the following properties

`chordNameExceptions`
: This is a list that contains the chords that have special formatting. For an example, see '`input/regression/chord-name-exceptions.ly`'.

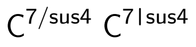
`majorSevenSymbol`

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 \chords {
  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.

The predefined variables `\germanChords`, `\semiGermanChords` set these variables.

`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.

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 A.1 [Chord name chart], page 158. 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`', '`input/test/chord-names-german.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.

3.14 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.

3.14.1 Multiple staff contexts

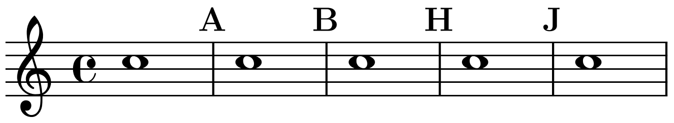
Polyphonic scores consist of many staves. These staves can be constructed in three different ways

- The group is started with a brace at the left, and bar lines are connected. This is done with the `GrandStaff` context.
- The group is started with a bracket, and bar lines are connected. This is done with the `StaffGroup` context
- The group is started with a vertical line. Bar lines are not connected. This is the default for the score.

3.14.2 Rehearsal marks

To print a rehearsal mark, use the `\mark` command

```
\relative c'' {
  c1 \mark \default
  c1 \mark \default
  c1 \mark #8
  c1 \mark \default
  c1 \mark \default
}
```

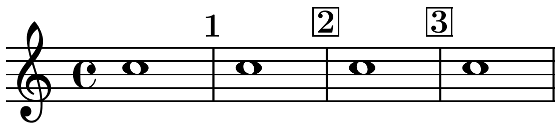


(The letter I is skipped in accordance with engraving traditions.)

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.

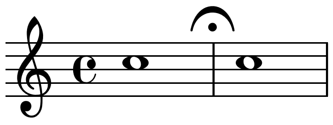
```
\set Score.markFormatter = #format-mark-numbers
c1 \mark \default
c1 \mark \default
\set Score.markFormatter
   = #(lambda (mark  context)
       (make-bold-markup (make-box-markup (number->string mark))))
c1 \mark \default
c1 \mark \default
```



The file '`scm/translation-functions.scm`' contains the definitions of `format-mark-numbers` (the default format) and `format-mark-letters`. They can be used as inspiration for other formatting functions.

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

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



In the case of a line break, marks must also be printed at the end of the line, and not at the beginning. Use the following to force that behavior

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

See also

Program reference: `MarkEvent`, `RehearsalMark`.

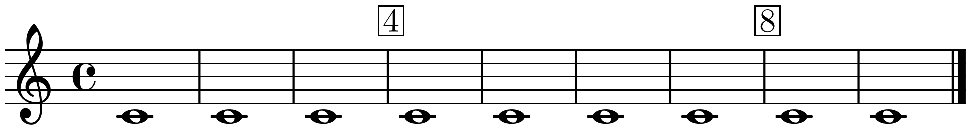
Init files: '`scm/translation-functions.scm`' contains the definition of `format-mark-numbers` and `format-mark-letters`. They can be used as inspiration for other formatting functions.

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

3.14.3 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`'



See also

Program reference: `BarNumber`.

Examples: '`input/test/bar-number-every-five-reset.ly`', and '`input/test/bar-number-regular-interval.ly`'.

Bugs

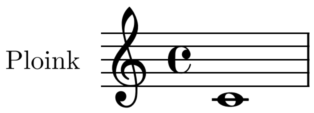
Bar numbers can collide with the `StaffGroup` bracket, if there is one at the top. To solve this, the `padding` property of `BarNumber` can be used to position the number correctly.

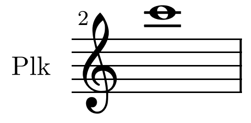
3.14.4 Instrument names

In an orchestral score, instrument names are printed 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 start, `instrument` is used, for the next ones `instr` is used.

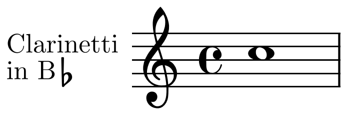
```
\set Staff.instrument = "Ploink "
\set Staff.instr = "Plk "
c1
\break
c''
```





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

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



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.

3.14.5 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

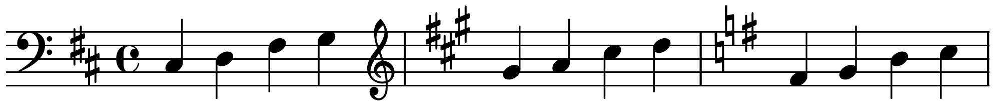
```
\transpose d e ...
```

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 ...
```

Since *from* and *to* are pitches, so `\transpose` must be inside a `\notes` section. `\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 =\notes { \key d \major cis d fis g }
\score { \notes \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`.

3.14.6 Instrument transpositions

The key of a transposing instrument can also be specified. This applies to many wind instruments, for example, clarinets (B-flat, A and E-flat), horn (F) and trumpet (B-flat, C, D and E-flat).

The transposition is entered after the keyword `\transposition`

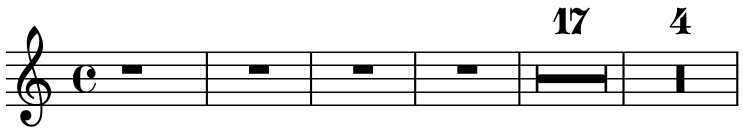
```
\transposition bes   %%  B-flat clarinet
```

This command sets the property `instrumentTransposition`. The value of this property is used for MIDI output and quotations. It does not affect how notes are printed in the current staff.

3.14.7 Multi measure rests

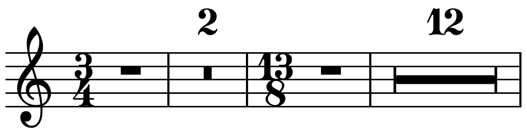
Multi measure rests are entered using '`R`'. It is specifically meant for full bar rests and for entering parts: the rest can expand to fill a score with rests, or it can be printed as a single multimeasure rest. This expansion is controlled by the property `Score.skipBars`. If this is set to true, empty 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

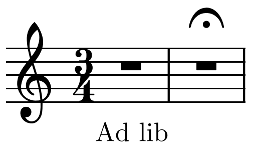
```
\set Score.skipBars = ##t
\time 3/4
R2. | R2.*2
\time 13/8
R1*13/8
R1*13/8*12
```



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.

Texts can be added to multi-measure rests by using the *note*-`markup` syntax (see Section 4.5 [Text markup], page 123). In this case, the number is replaced. If you need both texts and the number, you must add the number by hand. A variable (`\fermataMarkup`) is provided for adding fermatas

```
\time 3/4
R2._\markup { "Ad lib" }
R2.^\fermataMarkup
```



If you want to have a 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.

There is no way to automatically condense multiple rests into a single multimeasure rest. Multi measure rests do not take part in rest collisions.

Be careful when entering multimeasure 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.

3.14.8 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
```

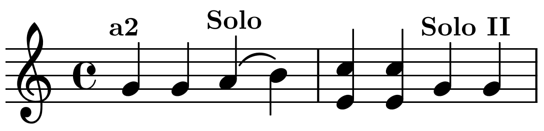
The music expressions will be interpreted as `Voice` contexts. If using relative octaves, `\relative` should be specified for both music expressions, i.e.

```
\partcombine \relative ...  musicexpr1
  \relative ... musicexpr2
```

A `\relative` section that is outside of `\partcombine` has no effect on the pitches of *musicexpr1* and *musicexpr2*.

The following example demonstrates the basic functionality of the part combiner: putting parts on one staff, and setting stem directions and polyphony

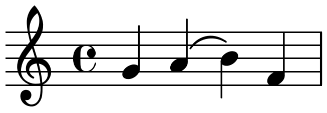
```
\new Staff \partcombine
  \relative g' { g g a( b) c c r r }
  \relative g' { g g r4 r e e g g }
```



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 `soloADue` to false

```
\new Staff <<
  \set Staff.soloADue = ##f
  \partcombine
    \relative g' { g a( b) r }
    \relative g' { g r4 r f }
>>
```



See also

Program reference: `PartCombineMusic`, `SoloOneEvent`, and `SoloTwoEvent`, and `UnisonoEvent`.

Bugs

In `soloADue` mode, 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`.

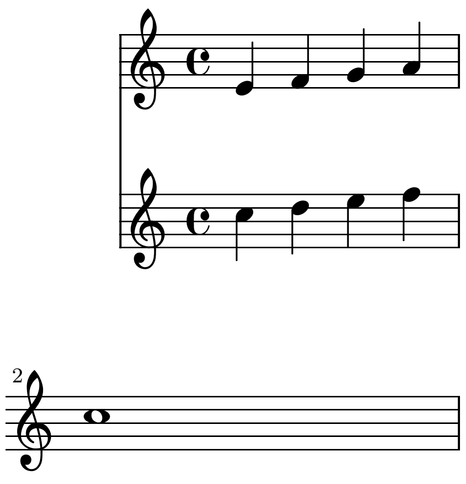
Internally, the `\partcombine` interprets both arguments as `Voice`s named `one` and `two`, and then decides when the parts can be combined. Consequently, if the arguments switch to differently named `Voice` contexts, the events in those will be ignored.

3.14.9 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 these line 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

```
\score  {
  \notes \relative c' <<
    \new Staff { e4 f g a \break c1 }
    \new Staff { c4 d e f \break R1 }
  >>
  \paper {
    linewidth = 6.\cm
    \context { \RemoveEmptyStaffContext }
    }
  }
```



The first page shows all staves in full. If empty staves should be removed from the first page 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.

3.14.10 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
\relative c' <<
  \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
```

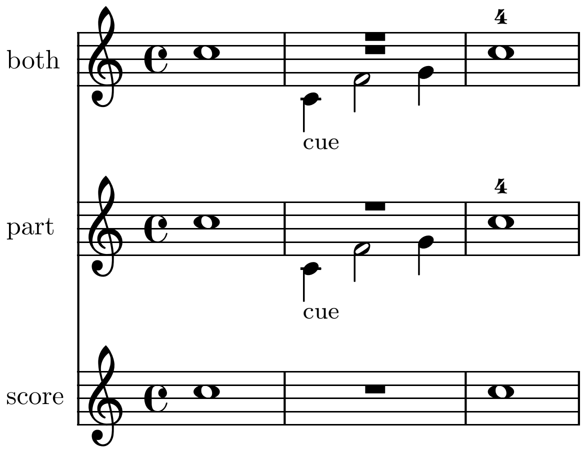
This defines a note with a conditional fingering indication.

By applying the `remove-tag` function, tagged expressions can be filtered. For example,

```
\simultaneous {
  the music
  \apply #(remove-tag 'score) the music
  \apply #(remove-tag '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`'.

3.14.11 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 code `\addquote` command. The quotation may then be done with `\quote`

```
  \addquote name music
  \quote name duration
```

Here, *name* is an identifying string. The *music* is any kind of music. This is an example of `\addquote`

```
\addquote clarinet \notes\relative c' {
  f4 fis g gis
}
```

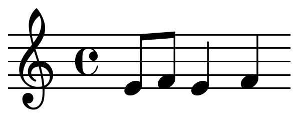
During a part, a piece of music can be quoted with the `\quote` command.

```
  \quote clarinet 2.
```

This would cite 3 quarter notes (a dotted half note) of the previously added clarinet voice.

Quotations take into account the transposition both source and target instruments, if they are specified using the `\transposition` command.

```
  \addquote clarinet \notes\relative c' {
    \transposition bes
    f4 fis g gis
  }
  \score {
    \notes {
    e'8 f'8 \quote clarinet 2
  } }
```



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.

See also

In this manual: Section 3.14.6 [Instrument transpositions], page 85.

Examples: '`input/regression/quote.ly`' '`input/regression/quote-transposition.ly`'

Program reference: `QuoteMusic`.

3.15 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 3.15.1 [Ancient note heads], page 90, Section 3.15.2 [Ancient accidentals], page 91, Section 3.15.3 [Ancient rests], page 91, Section 3.15.4 [Ancient clefs], page 92, Section 3.15.5 [Ancient flags], page 94 and Section 3.15.6 [Ancient time signatures], page 95. By manipulating such a grob property, the typographical appearance of the affected graphical objects can be accommodated for a specific notation flavor without need for introducing any new notational concept.

Other aspects of ancient notation can not that easily be expressed as in terms of just changing a style property of a graphical object. Therefore, some notational concepts are introduced specifically for ancient notation, see Section 3.15.7 [Custodes], page 95, Section 3.15.8 [Divisiones], page 96, Section 3.15.9 [Ligatures], page 97, and Section 3.15.11 [Figured bass], page 103.

If this all is way 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 (see Section 3.15.10 [Vaticana style contexts], page 102). Use them to set up predefined style-specific voice and staff contexts, and directly go ahead with the note entry.

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.

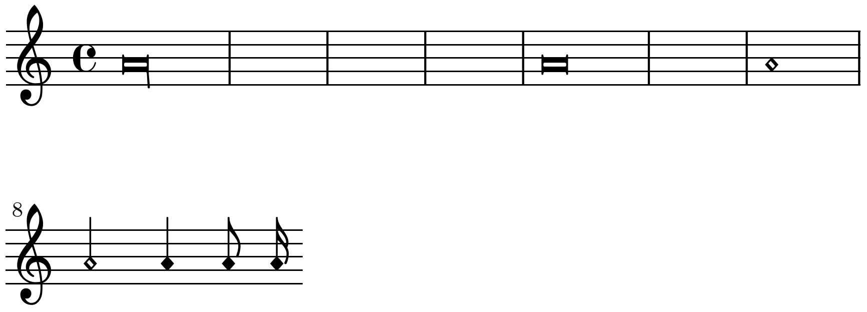
Augmentum dots within ligatures are not handled correctly.

3.15.1 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 the desired value (`baroque`, `neo_mensural` or `mensural`). The `baroque` style differs from the `default` style only in using a square shape for `\breve` note heads. The `neo_mensural` 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 in particular 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 `neo_mensural` style

```
\override NoteHead #'style = #'neo_mensural
a'\longa a'\breve a'1 a'2 a'4 a'8 a'16
```



When typesetting a piece in Gregorian Chant notation, a Gregorian ligature engraver will automatically select the proper note heads, such 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 3.15.9 [Ligatures], page 97 for how ligature engravers work.

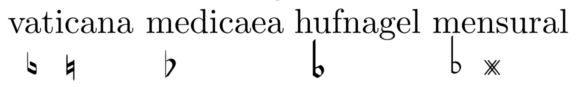
See also

In this manual Section 3.9.3 [Percussion staves], page 65 use note head styles of their own that are frequently used in contemporary music notation.

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

3.15.2 Ancient accidentals

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



As shown, not all accidentals are supported by each style. When trying to access an unsupported accidental, LilyPond will switch to a different style, as demonstrated in '`input/test/ancient-accidentals.ly`'.

Similarly to local accidentals, the style of the key signature can be controlled by the `style` property of the `KeySignature` grob.

See also

In this manual: Section 3.1.2 [Pitches], page 31, Section 3.1.3 [Chromatic alterations], page 32 and Section 3.6 [Accidentals], page 49 give a general introduction into the use of accidentals. Section 3.3.2 [Key signature], page 40 gives a general introduction into the use of key signatures.

Program reference: `KeySignature`.

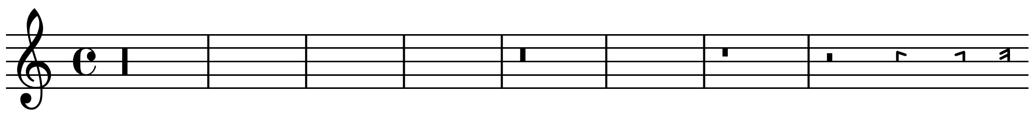
Examples: '`input/test/ancient-accidentals.ly`'.

3.15.3 Ancient rests

Use the `style` property of grob `Rest` to select ancient accidentals. Supported styles are `classical`, `neo_mensural` and `mensural`. `classical` differs from the `default` style only in that the quarter rest looks like a horizontally mirrored 8th rest. The `neo_mensural` 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 `neo_mensural` style

```
\override Rest #'style = #'neo_mensural
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 3.15.8 [Divisiones], page 96.

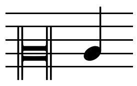
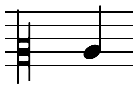
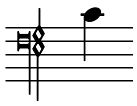
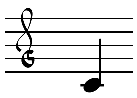
See also

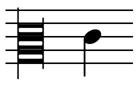
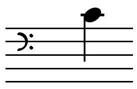
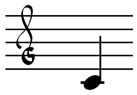
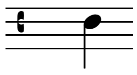
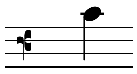
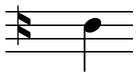
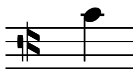
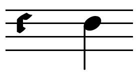
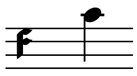
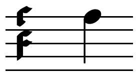
In this manual: Section 3.1.5 [Rests], page 33 gives a general introduction into the use of rests.

3.15.4 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 3.3.3 [Clef], page 40. The note printed to the right side of each clef in the example column denotes the `c'` with respect to that clef.

Glyph Name	**Description**	**Supported Clefs**	**Example**
`clefs-neo_mensural_c`	modern style mensural C clef	`neo_mensural_c1`, `neo_mensural_c2`, `neo_mensural_c3`, `neo_mensural_c4`	
`clefs-petrucci_c1` `clefs-petrucci_c2` `clefs-petrucci_c3` `clefs-petrucci_c4` `clefs-petrucci_c5`	petrucci style mensural C clefs, for use on different staff lines (the examples shows the 2nd staff line C clef).	`petrucci_c1` `petrucci_c2` `petrucci_c3` `petrucci_c4` `petrucci_c5`	
`clefs-petrucci_f`	petrucci style mensural F clef	`petrucci_f`	
`clefs-petrucci_g`	petrucci style mensural G clef	`petrucci_g`	

`clefs-mensural_c`	historic style mensural C clef	`mensural_c1`, `mensural_c2`, `mensural_c3`, `mensural_c4`	
`clefs-mensural_f`	historic style mensural F clef	`mensural_f`	
`clefs-mensural_g`	historic style mensural G clef	`mensural_g`	
`clefs-vaticana_do`	Editio Vaticana style do clef	`vaticana_do1`, `vaticana_do2`, `vaticana_do3`	
`clefs-vaticana_fa`	Editio Vaticana style fa clef	`vaticana_fa1`, `vaticana_fa2`	
`clefs-medicaea_do`	Editio Medicaea style do clef	`medicaea_do1`, `medicaea_do2`, `medicaea_do3`	
`clefs-medicaea_fa`	Editio Medicaea style fa clef	`medicaea_fa1`, `medicaea_fa2`	
`clefs-hufnagel_do`	historic style hufnagel do clef	`hufnagel_do1`, `hufnagel_do2`, `hufnagel_do3`	
`clefs-hufnagel_fa`	historic style hufnagel fa clef	`hufnagel_fa1`, `hufnagel_fa2`	
`clefs-hufnagel_do_fa`	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 3.3.3 [Clef], page 40.

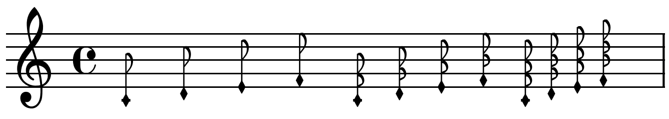
Bugs

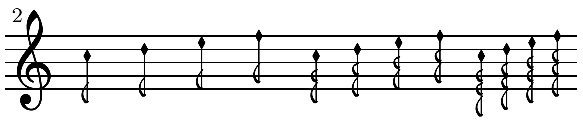
The mensural g clef is mapped to the Petrucci g clef, until a new mensural g clef is implemented.

3.15.5 Ancient flags

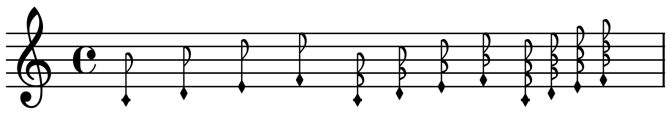
Use the `flag-style` property of grob `Stem` to select ancient flags. Besides the `default` flag style, only `mensural` style is supported

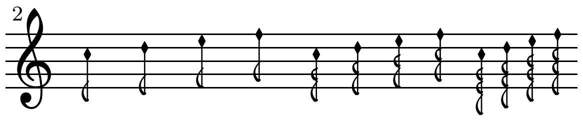
```
\override Stem #'flag-style = #'mensural
\override Stem #'thickness = #1.0
\override NoteHead #'style = #'mensural
\autoBeamOff
c'8 d'8 e'8 f'8 c'16 d'16 e'16 f'16 c'32 d'32 e'32 f'32 s8
c''8 d''8 e''8 f''8 c''16 d''16 e''16 f''16 c''32 d''32 e''32 f''32
```





Note that the innermost flare of each mensural flag always is vertically aligned with a staff line. If you do not like this behavior, you can set the `adjust-if-on-staffline` property of grob `Stem` to `##f`. Then, the vertical position of the end of each flare is different between notes on staff lines and notes between staff lines

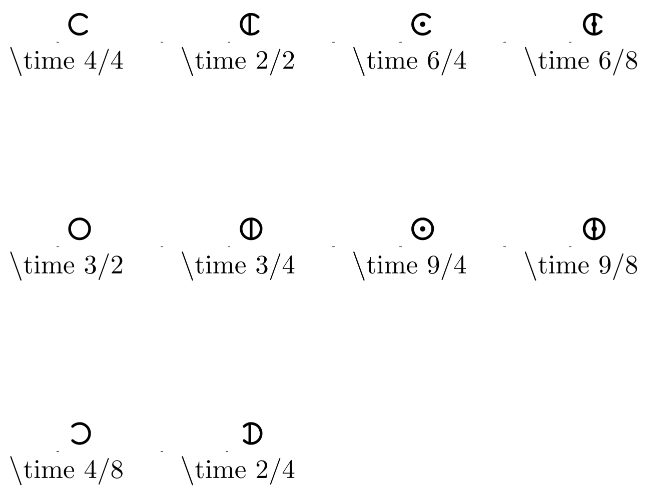




There is no particular flag style for neo-mensural notation. Hence, when typesetting e.g. the incipit of a transcribed piece of mensural music, the default flag style should be used. There are no flags in Gregorian Chant notation.

3.15.6 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 `neo_mensural` and `mensural`. The above table uses the `neo_mensural` style. This style is appropriate e.g. for the incipit of transcriptions of mensural pieces. The `mensural` style mimics the look of historical printings of the 16th century.

'`input/test/time.ly`' gives an overview over all available ancient and modern styles.

See also

Program reference: Section 3.3.5 [Time signature], page 42 gives a general introduction into the use of time signatures.

Bugs

Mensural signature glyphs are mapped to time fractions in a hard-wired way. This mapping is sensible, but still arbitrary: given a mensural time signature, the time fraction represents a modern meter that usually will be a good choice when transcribing a mensural piece of music. For a particular piece of mensural music, however, the mapping may be unsatisfactory. In particular, the mapping assumes a fixed transcription of durations (e.g. brevis = half note in 2/2, i.e. 4:1). Some glyphs (such as the alternate glyph for 6/8 meter) are not at all accessible through the `\time` command.

Mensural time signatures are supported typographically, but not yet musically. The internal representation of durations is based on a purely binary system; a ternary division such as 1 brevis = 3 semibrevis (tempus perfectum) or 1 semibrevis = 3 minima (cum prolatione maiori) is not correctly handled: event times in ternary modes will be badly computed, resulting e.g. in horizontally misaligned note heads, and bar checks are likely to erroneously fail.

The syntax and semantics of the `\time` command for mensural music is subject to change.

3.15.7 Custodes

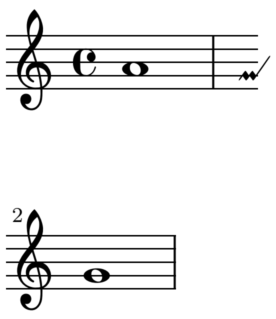
A *custos* (plural: *custodes*; Latin word for 'guard') is a symbol that appears at the end of a staff. It anticipates the pitch of the first note(s) of the following line and thus helps the player or singer to manage line breaks during performance, thus enhancing readability of a score.

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.

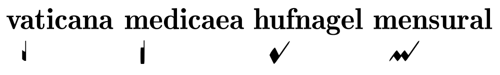
For typesetting custodes, just put a `Custos_engraver` into the `Staff` context when declaring the `\paper` block, as shown in the following example

```
\paper {
  \context {
     \StaffContext
     \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

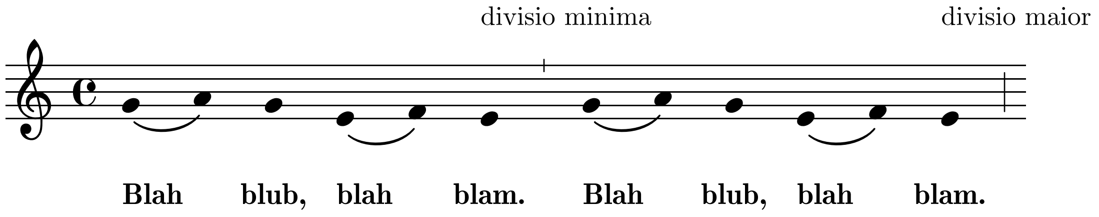
Program reference: `Custos`.

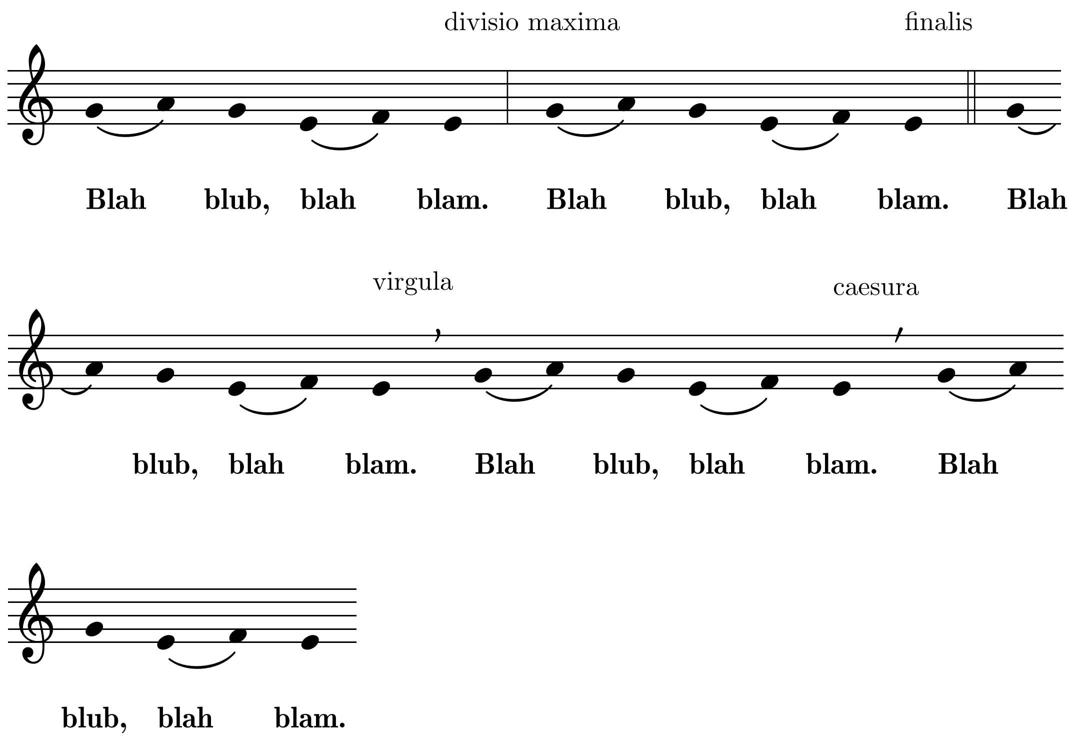
Examples: '`input/regression/custos.ly`'.

3.15.8 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 Section 3.7.3 [Breath marks], page 52. 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`




divisio maxima
finalis
Blah blub, blah blam. Blah blub, blah blam. Blah
virgula
caesura
blub, blah blam. Blah blub, blah blam. Blah
blub, blah blam.

Predefined commands

`\virgula`, `\caesura`, `\divisioMinima`, `\divisioMaior`, `\divisioMaxima`, `\finalis`.

See also

In this manual: Section 3.7.3 [Breath marks], page 52.

Program reference: `BreathingSign`, `BreathingSignEvent`.

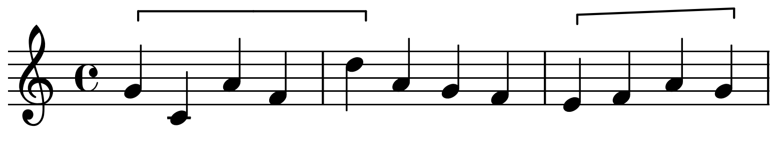
Examples: '`input/test/divisiones.ly`'.

3.15.9 Ligatures

A ligature is a coherent graphical symbol that represents at least two distinct notes. Ligatures originally appeared in the manuscripts of Gregorian chant notation roughly since the 9th century 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

```
\score {
  \notes \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.

3.15.9.1 White mensural ligatures

There is limited support for white mensural ligatures.

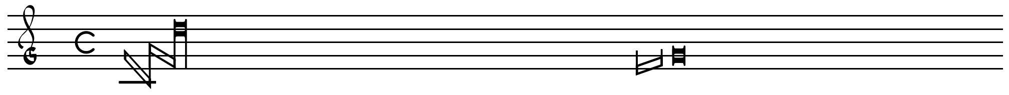
To engrave white mensural ligatures, in the paper block the `Mensural_ligature_engraver` has to be put into the `Voice` context, and remove the `Ligature_bracket_engraver`

```
\paper {
    \context {
        \VoiceContext
        \remove Ligature_bracket_engraver
        \consists Mensural_ligature_engraver
    }
}
```

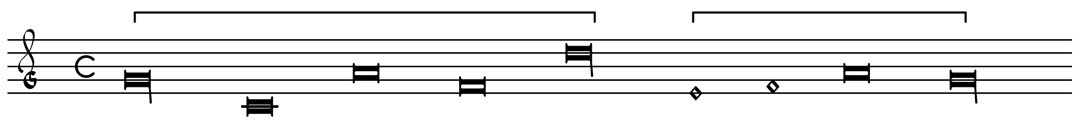
There is no additional input language to describe the shape of a white mensural ligature. The shape is rather determined solely from the pitch and duration of the enclosed notes. While this approach may take a new user a while to get accustomed, it has the great advantage that the full musical information 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 = #'neo_mensural
\override Staff.TimeSignature   #'style = #'neo_mensural
\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



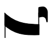
Bugs

The implementation is experimental; it may output strange warnings or even crash in some cases or produce weird results on more complex ligatures.

3.15.9.2 Gregorian square neumes ligatures

Gregorian square neumes notation (following the style of the Editio Vaticana) is under heavy development, but not yet really usable for production purposes. Core ligatures can already be typeset, but essential issues for serious typesetting are still under development, such as (among others) horizontal alignment of multiple ligatures, lyrics alignment and proper accidentals handling. Still, this section gives a sneak preview of what Gregorian chant may look like once it will work.

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	w	x	
10. Torculus	y	z	A
11. Torculus Initio Debilis	B	C	D
12. Porrectus	E	F 	G

	H	I	J
`13. Climacus`			
	K	L	M
`14. Scandicus`			
	N	O	
`15. Salicus`			
	P		
`16. Trigonus`			

Unlike most other neumes notation systems, the input language for neumes does not necessarily reflect directly the typographical appearance, but is designed to solely focuse 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 purely taken from 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`.

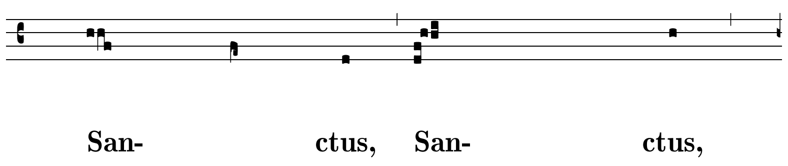
Head prefixes can be accumulated, though restrictions apply. For example, either `\descendens` or `\ascendens` can be applied to a head, but not both to the same head.

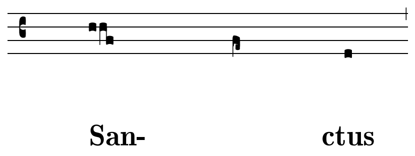
Two adjacent heads can be tied together with the `\pes` and `\flexa` infix commands for a rising and falling line of melody, respectively.

3.15.10 Vaticana style contexts

The predefined `VaticanaVoiceContext` and `VaticanaStaffContext` can be used to easily 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. With these contexts, you can immediately go ahead entering the chant, as the following short excerpt demonstrates

```
\include "gregorian-init.ly"
\score {
  <<
    \context VaticanaVoice = "cantus" {
      \override Score.BarNumber #'transparent = ##t
      \notes {
        \[ c'\melisma c' \flexa a \]
        \[ a \flexa \deminutum g\melismaEnd \]
        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 \lyrics {
      San- ctus, San- ctus, San- ctus
    }
  >>
}
```

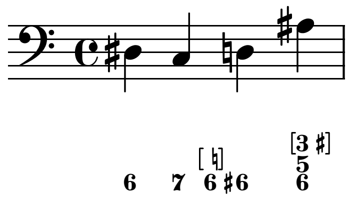




3.15.11 Figured bass

LilyPond has limited support for figured bass

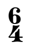
```
<<
  \context Voice \notes { \clef bass dis4  c d ais }
  \context FiguredBass \figures {
    < 6 >4 < 7 >8 < 6+ [_!] >
    < 6 >4 <6 5 [3+] >
  }
>>
```



The support for figured bass consists of two parts: there is an input mode, introduced by `\figures`, where you can enter bass figures as numbers, and there is a context called `FiguredBass` that takes care of making `BassFigure` objects.

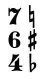
In figures input mode, a group of bass figures is delimited by `<` and `>`. The duration is entered after the `>>`

```
<4 6>
```



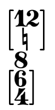
Accidentals are added when you append `-`, `!` and `+` to the numbers

```
<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 `\figures` 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.

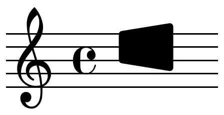
3.16 Contemporary notation

In the 20th century, composers have greatly expanded the musical vocabulary. With this expansion, many innovations in musical notation have been tried. The book by Stone (1980) gives a comprehensive overview (see Appendix B [Literature list], page 182). In general, the use of new, innovative notation makes a piece harder to understand and perform and its use should therefore be avoided if possible. For this reason, support for contemporary notation in LilyPond is limited.

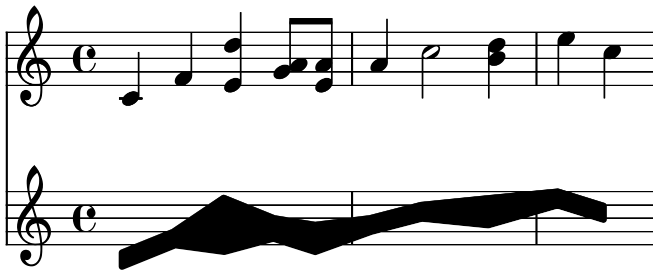
3.16.1 Clusters

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 `notes-to-clusters` to a sequence of chords, e.g.

```
\apply #notes-to-clusters {  <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`.

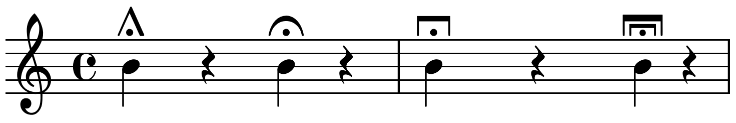
Examples: '`input/regression/cluster.ly`'.

Bugs

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

3.16.2 Fermatas

Contemporary music notation frequently uses special fermata symbols to indicate fermatas of differing lengths. The following fermatas are supported



`shortfermata` `fermata` `longfermata` `verylongfermata`

See Section 3.7.7 [Articulations], page 54 for general instructions how to apply scripts such as fermatas to a `\notes{}` block.

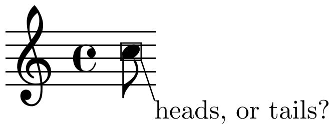
3.17 Special notation

3.17.1 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 where to put the label relative to the object. In the above example, the text “heads or tails?” ends 3 spaces below the ‘balloon.’

See also

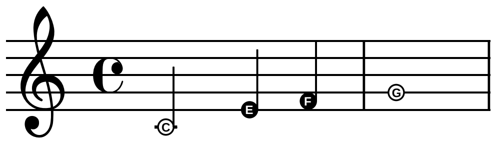
Program reference: `text-balloon-interface`.

Examples: ‘`input/regression/balloon.ly`’.

3.17.2 Easy Notation note heads

The ‘easy play’ note head includes a 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 4.6.1 [Setting global staff size], page 128.

If you view the result with Xdvi, staff lines may show through the letters. Printing the PostScript file obtained does produce the correct result.

Predefined commands

`\setEasyHeads`

3.18 Sound

Entered music can also be converted to MIDI output. The performance is intended for proof-hearing the music for errors.

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 extremities. 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`.

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. Hence, the MIDI file should not have more than 15 staves (or 14 if you do not use drums). Other staves will remain silent.

3.18.1 MIDI block

The MIDI block is analogous to the paper 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 \paper block. Translation modules for sound are called performers. The contexts for MIDI output are defined in '`ly/performer-init.ly`'.

3.18.2 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 A.2 [MIDI instruments], page 159.

Bugs

If the selected string does not exactly match, the default is used, which is the Grand Piano.

4 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 certain effect.

The controls available for tuning are described in a separate document, the `Program reference` manual. This 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, so it is necessary to learn a (very small) subset of Scheme. That is why this chapter starts with a short tutorial on entering numbers, lists, strings and symbols in Scheme.

4.1 Scheme tutorial

LilyPond uses the Scheme programming language, both as part of the input syntax, and as internal mechanism to glue together modules of the program. 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

[1] If you want to know more about Scheme, see `http://www.schemers.org`.

```
"a said \"b\""
```

Newlines and backslashes are escaped with `\n` and `\\` respectively.

In a music file, snippets of Scheme code are introduced with the hash mark `#`. So, the previous examples translated in LilyPond are

```
##t ##f
#1 #-1.5
#"this is a string"
#"this
is
a string"
```

For the rest of this section, we will assume that the data is entered in a music file, so we add `#`s everywhere.

Scheme can be used to do calculations. It uses *prefix* syntax. Adding 1 and 2 is written as `(+ 1 2)` rather than the traditional 1+2.

```
#(+ 1 2)
 ⇒ #3
```

The arrow ⇒ shows that the result of evaluating `(+ 1 2)` is `3`. Calculations may be nested; the result of a function may be used for another calculation.

```
#(+ 1 (* 3 4))
 ⇒ #(+ 1 12)
 ⇒ #13
```

These calculations are examples of evaluations; an expression like `(* 3 4)` is replaced by its value `12`. A similar thing happens with variables. After defining a variable

```
twelve = #12
```

variables can also be used in expressions, here

```
twentyFour = #(* 2 twelve)
```

the number 24 is stored in the variable `twentyFour`.

The *name* of a variable is also an expression, similar to a number or a string. It is entered as

```
#'twentyFour
```

The quote mark `'` prevents 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 a 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 internal variables, we will call the latter “properties.” So, the stem object has a `thickness` property.

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,

[2] In Scheme terminology, the pair is called `cons`, and its two elements are called car and cdr respectively.

```
\override TextScript #'extra-offset = #'(1 . 2)
```

This assigns the pair (1, 2) to `extra-offset` variable 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 evaluation. This is done by quoting the list with a quote `'` symbol. For calculations, do not use a quote.

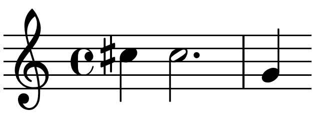
Inside a quoted list or pair, there is no need to quote anymore. The following is a pair of symbols, a list of symbols and a list of lists respectively,

```
#'(stem . head)
#'(staff clef key-signature)
#'((1) (2))
```

4.2 Interpretation contexts

When music is printed, a lot of notation 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 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 stave, 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 accidentals and then the `Staff` context maintains the rule to show or suppress the accidental for the remainder of the measure.

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 define new contexts.

Full description of all available contexts is in the program reference, see Translation ⇒ Context.

4.2.1 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 which do this.

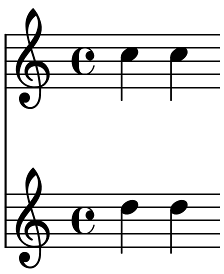
The easiest command is `\new`, and it also the quickest to type. It is prepended to a music expression, for example

```
\new type music expression
```

where *type* is a context name (like `Staff` or `Voice`). This command creates a new context, and starts interpreting *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,

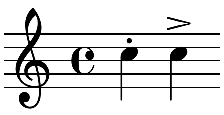
```
\new Lyrics \lyricsto "tenor" lyrics
```

Another possibility is funneling two different music expressions into one context. In the following example, articulations and notes are entered separately,

```
music = \notes { c4 c4 }
arts = \notes  { s4-. s4-> }
```

They are combined by sending both to the same `Voice` context,

```
<< \new Staff \context Voice = "A" \music
   \context Voice = "A" \arts
>>
```



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 C.3.2 [Running a function on all layout objects], page 188). Without an explicit `\context`, it is usually is 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
```

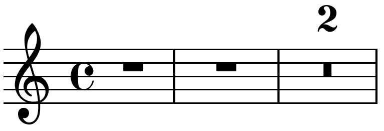
4.2.2 Changing context properties on the fly

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,

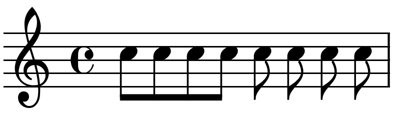
```
R1*2
\set Score.skipBars = ##t
R1*2
```



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, and the current `Voice` is used.

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,

```
\set context.prop
```

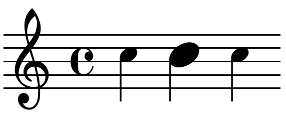
which removes the definition of *prop*. This command removes the definition only if it is set in *context*. In

```
\set Staff.autoBeaming = ##f
\unset Voice.autoBeaming
```

the current `Voice` does not have the property, and the definition at `Staff` level remains intact. Like `\set`, the *context* argument does not have to be specified for a bottom context.

Settings that should only apply to a single time-step can be entered easily 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.

4.2.3 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`.

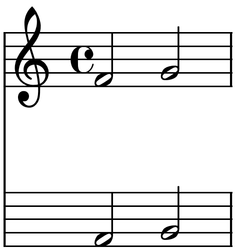
For a full a description of each plug-in, see Program reference ⇒ Translation ⇒ Engravers. Every context described in Program reference ⇒ Translation ⇒ Context. lists the engravers used for that context.

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,

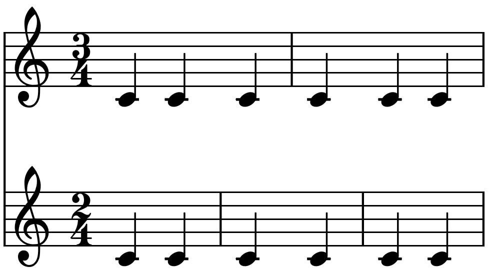
```
<< \new Staff {
    f2 g
  }
  \new Staff \with {
     \remove "Time_signature_engraver"
     \remove "Clef_engraver"
  } {
    f2 g2
  }
>>
```



In the second stave 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 will be adversely influenced too. A more sophisticated methods of blanking objects is shown in Section 4.3.1 [Common tweaks], page 117.

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
  }
>>
```



4.2.4 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
```

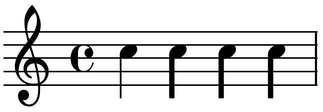
Here *name* is the name of a graphical object, like `Stem` or `NoteHead`, and *property* is an internal variable of the formatting system ('grob property' or 'layout property'). The latter is a symbol, so it must be quoted. The subsection Section 4.3.2 [Constructing a tweak], page 118 explains what to fill in for *name*, *property*, and *value*. Here we only discuss 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:

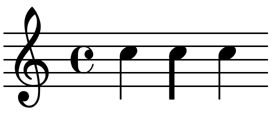
```
c4
\override Staff.Stem #'thickness = #4.0
c4
c4
c4
```



The `\override` command is executed during the interpreting phase, and changes the definition of the `Stem` within `Staff`. After the command all stems are thickened.

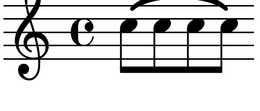
Analogous to `\set`, the *context* argument may be left out, causing it to default to `Voice`, and adding `\once` applies the change during one timestep only

```
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 and 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 a `\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.

4.2.5 Changing context default settings

The adjustments of the previous chapters can also be entered separate from the music, in the `\paper` block,

```
\paper {
   ...
   \context {
      \StaffContext

      \set fontSize = #-2
      \override Stem #'thickness
      \remove "Time_signature_engraver"
    }
 }
```

Here

```
\StaffContext
```

takes the existing definition `Staff` from the identifier `StaffContext`. This works analogously to other contexts, so that the existing definition of `Voice` is in `\VoiceContext`.

The statements

```
      \set fontSize = #-2
      \override Stem #'thickness
      \remove "Time_signature_engraver"
```

affect all staves in the score.

The `\set` keyword is optional within the `\paper` block, so

```
fontSize = #-2
```

will also work.

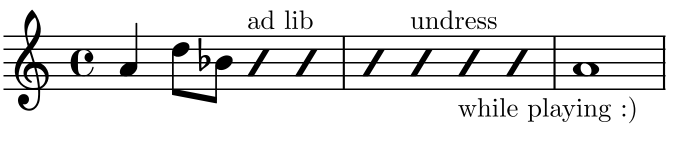
Bugs

It is not possible to collect changes in a variable, and apply them to one `\context` definition by referring to that variable.

4.2.6 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 print 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 `\paper` block,

```
\paper {
  \context {
    ...
  }
}
```

In the following discussion, the example input shown should go on the ... in the previous fragment.

First, name 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"
```

Putting together, we get

```
\context {
  \name ImproVoice
  \type "Engraver_group_engraver"
  \consists "Note_heads_engraver"
  \consists "Text_script_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,[3]

```
\context {
  \StaffContext
  \accepts ImproVoice
}
```

Putting both into a `\paper` block, like

[3] The opposite of `\accepts` is `\denies`, which is sometimes when reusing existing context definitions.

```
    \paper {
      \context {
        \name ImproVoice
        ...
      }
    \context {
      \StaffContext
      \accepts "ImproVoice"
    }
  }
```

Then the output at the start of this subsection can be entered as

```
\score {
  \notes \relative c'' {
     a4 d8 bes8
     \new ImproVoice {
       c4^"ad lib" c
       c4 c^"undress"
       c c_"while playing :)"
     }
     a1
  }
}
```

4.2.7 Which properties to change

There are many different properties. Not all of them are listed in this manual. However, the program reference lists them all in the section `Tunable-context-properties`, and most properties are demonstrated in one of the tips-and-tricks examples.

4.3 Tuning output

In the previous section, we have already touched on a command that changes layout details, the `\override` command. In this section, we will look at in more detail how to use the command in practice. First, we will give a a few versatile commands, which are sufficient for many situations. The next section will discuss general use of `\override`.

4.3.1 Common tweaks

Some overrides are so common that predefined commands are provided as a short-cut, for example, `\slurUp` and `\stemDown`. These commands are described in Chapter 3 [Notation manual], page 31, under the sections for slurs and stems respectively.

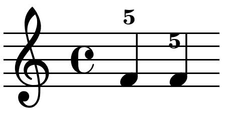
The exact tuning possibilities for each type of layout object are documented in the program reference of the respective object. However, many layout objects share properties, which can be used to apply generic tweaks. We mention a few of these:

- The `extra-offset` property, which has a pair of numbers as value, moves around objects 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:

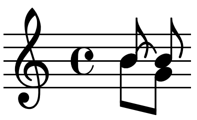
  ```
  \stemUp
  f-5
  ```

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



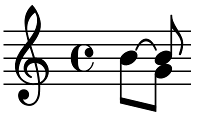
- Setting the `transparent` property will cause an object to be printed in 'invisible ink': the object is not printed, but all its other behavior is retained. The object still takes up space, it takes part in collisions, and slurs, and ties and beams can be attached to it.

  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,



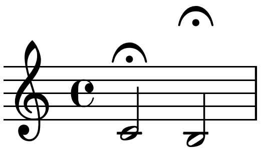
  and blanking a stem in that voice, the tie appears to cross voices:

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



- The `padding` property for objects with `side-position-interface` can be set to increase distance between symbols that are printed above or below notes. We only give an example; a more elaborate explanation is in Section 4.3.2 [Constructing a tweak], page 118:

```
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.

4.3.2 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.

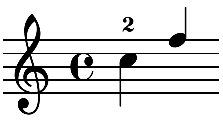
The program reference is a set of HTML pages, which is part of the documentation package. On Unix systems, it is typically in '`/usr/share/doc/lilypond`'. If you have them, it is best to bookmark them in your webbrowser, because you will need them. They are also available on the web: go to the LilyPond website (`http://lilypond.org`), click "Documentation", select the correct version, and then click "Program reference."

If you have them, use the local HTML files. They will load faster, and they are exactly matched to LilyPond version installed.

4.3.3 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 of `Fingering` (in Section 3.7.8 [Fingering instructions], page 56), you will notice that there is written:

> **See also**
>
> Program reference: `FingerEvent` and `Fingering`.

This fragments points to two parts of the program reference: a page on `FingerEvent` and 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 objects of type `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.

4.3.4 Layout interfaces

`Fingering` is a layout object. Such an object is a symbol within the score. It has properties, which 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's 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 a so-called *interface*, 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',

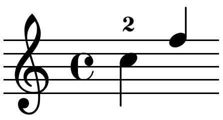
```
(Fingering
  . (
     (print-function . ,Text_item::print)
     (padding . 0.6)
     (staff-padding . 0.6)
     (self-alignment-X . 0)
     (self-alignment-Y . 0)
     (script-priority . 100)
     (font-encoding . number)
     (font-size . -5)
     (meta . ((interfaces . (finger-interface font-interface
            text-script-interface text-interface
            side-position-interface self-alignment-interface
            item-interface))))
))
```

as you can see, `Fingering` is nothing more than a bunch of variable settings, and the webpage is directly generated from this definition.

4.3.5 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

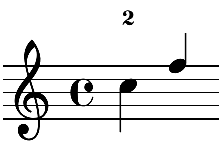
`padding`	(dimension, in staff space)
	add this much extra space between objects that are next to each other.

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 Fingering #'padding = #3
```

Inserting this command before the Fingering object is created, i.e. before `c2`, yields the following result:

```
\once \override Fingering
    #'padding = #3
c-2
\stemUp
f
```



In this case, the context for this tweak is `Voice`, which does not have to be specified for `\override`. This fact can also be deduced from the program reference, for the page for the `Fingering_engraver` plug-in says

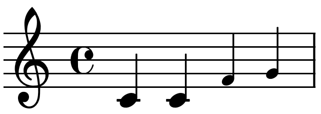
> Fingering_engraver is part of contexts: . . . `Voice`

4.4 Fonts

4.4.1 Selecting font sizes

The most common thing to change about the appearance of fonts is their size. The font size of any context can be easily changed by setting the `fontSize` property for that context. Its value is a number: negative numbers make the font smaller, positive numbers larger. An example is given below:

```
  c4 c4 \set fontSize = #-3
  f4 g4
```



This command will set `font-size` (see below) in all layout objects in the current context. It does not change the size of variable symbols, such as beams or slurs.

The font size is set by modifying the `font-size` property. Its value is a number indicating the size relative to the standard size. 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.

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 `font-size` mechanism does not work for fonts selected through `font-name`. These may be scaled with `font-magnification`.

One of the uses of `fontSize` is to get smaller symbols for cue notes. An elaborate example of those is in '`input/test/cue-notes.ly`'.

Predefined commands

The following commands set `fontSize` for the current voice:

`\tiny`, `\small`, `\normalsize`.

4.4.2 Font selection

Font selection for the standard fonts, TeX's Computer Modern fonts, can also be adjusted with a more fine-grained mechanism. By setting the object properties described below, you can select a different font; all three mechanisms work for every object that supports `font-interface`:

- `font-encoding` is a symbol that sets layout of the glyphs. Choices include `text` for normal text, `braces` (for piano staff braces), `music` (the standard music font, including ancient glyphs), `dynamic` (for dynamic signs) and `number` 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.

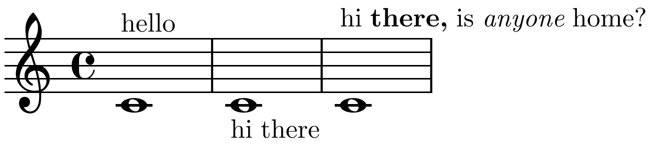
Bugs

No style sheet is provided for other fonts besides the TEX Computer Modern family.

4.5 Text markup

LilyPond has an internal mechanism to typeset texts. You can access it with the keyword `\markup`. Within markup mode, you can enter texts similar to lyrics: simply enter them, surrounded by spaces:

```
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:

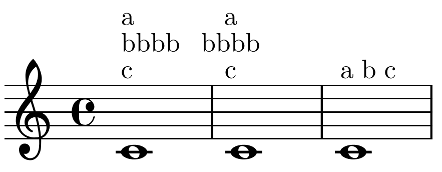
```
            \markup { \bold { hi there } }
```

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. The braces group notes into horizontal lines. Other types of lists also exist: you can stack expressions grouped with `<` and `>` vertically with the command `\column`. Similarly, `\center-align` aligns texts by their center lines:

```
c1^\markup { \column < a bbbb c > }
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 } }
     \notes { 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:

```
            "" \raise #0.5 raised
```

The text `raised` is now raised 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

Text layout is ultimately done by TEX, which does kerning of letters. LilyPond does not account for kerning, so texts will be spaced slightly too wide.

Syntax errors for markup mode are confusing.

Markup texts cannot be used in the titling of the `\header` field. Titles are made by LaTEX, so LaTEX commands should be used for formatting.

4.5.1 Overview of text markup commands

`\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*

`\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)
: This produces a single character, e.g. `\char #65` produces the letter 'A'.

`\column` *args* (list of markups)
: Stack the markups in *args* vertically.

`\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

`\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.

`\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.

`\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*.

`\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.

`\latin-i` *arg* (markup)
: TEST latin1 encoding.

`\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.

`\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 A.3 [The Feta font], page 160 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)
: A superscript which does not use a smaller font.

`\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.

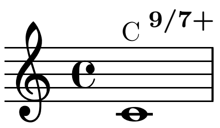
`\number` *arg* (markup)
: Set font family to `number`, which yields the font used for time signatures and fingerings. This font only contains numbers and some punctuation. It doesn't have any letters.

`\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"
```

`\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)
`\roman` *arg* (markup)
: Set font family to `roman`.

`\sans` *arg* (markup)
: Switch to the sans-serif family

`\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

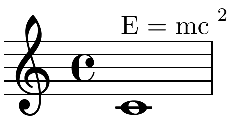
`\strut`
: Create a box of the same height as the space in the current font.

  FIXME: is this working?

`\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 center.

`\word` *str* (string)
: A single word.

4.6 Global layout

The global layout 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, which influences where line breaks are chosen, and thus ultimately how many pages a piece of music takes. This section explains how to tune the algorithm for spacing.

Globally spoken, this procedure happens in three steps: first, flexible distances ("springs") are chosen, based on durations. All possible line breaking combination 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. When the score is processed by TEX, each page is filled with systems, and page breaks are chosen whenever the page gets full.

4.6.1 Setting global staff size

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	17.82	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 size for individual staves. The size of individual staves are relative to the global size, which can be set in the following manner:

```
#(set-global-staff-size 14)
```

This sets the global default size to 14pt staff height, and scales all fonts accordingly.

See also

This manual: Section 4.4.1 [Selecting font sizes], page 121.

4.6.2 Vertical spacing

The height of each system is determined automatically by LilyPond, to keep 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 from its, then you could set

```
\set Staff.minimumVerticalExtent = #'(-4 . 4)
```

This sets the vertical size of the current staff to 4 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)`.

The piano staves are handled a little differently: to make cross-staff beaming work correctly, it is necessary that the distance between staves is fixed beforehand. This is also done with a `VerticalAlignment` object, created in `PianoStaff`. In this object the distance between the staves is fixed by setting `forced-distance`. If you want to override this, use a `\context` block as follows:

```
\paper {
  \context {
    \PianoStaffContext
    \override VerticalAlignment #'forced-distance = #9
  }
  ...
}
```

This would bring the staves together at a distance of 9 staff spaces, measured from the center line of each staff.

See also

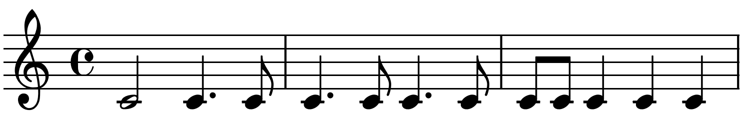
Internals: Vertical alignment of staves is handled by the `VerticalAlignment` object.

4.6.3 Horizontal Spacing

The spacing engine translates differences in durations into stretchable distances ("springs") of differing lengths. Longer durations get more space, shorter durations get less. The shortest durations get a fixed amount of space (which is controlled by `shortest-duration-space` in the `SpacingSpanner` object). The longer the duration, the more space it gets: doubling a duration adds a fixed amount (this amount is controlled by `spacing-increment`) of space to the note.

For example, the following piece contains lots of half, quarter, and 8th notes, the eighth note is followed by 1 note head width (NHW). The quarter note is followed by 2 NHW, the half by 3 NHW, etc.

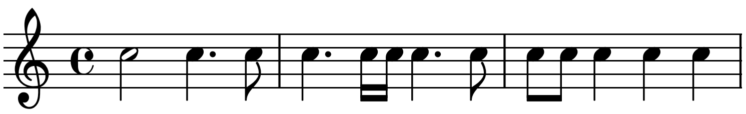
```
c2 c4. c8 c4. c8 c4. c8 c8
c8 c4 c4 c4
```



Normally, `spacing-increment` is set to 1.2, which is the width of a note head, and `shortest-duration-space` is set to 2.0, meaning that the shortest note gets 2 NHW of space. For normal notes, this space is always counted from the left edge of the symbol, so the shortest notes are generally followed by one NHW of space.

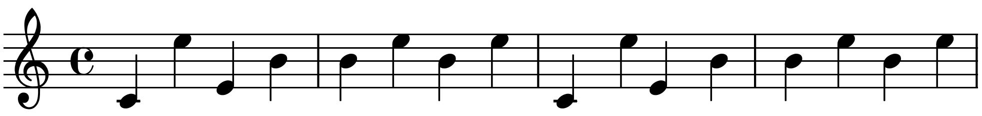
If one would follow the above procedure exactly, then adding a single 32th note to a score that uses 8th and 16th notes, would widen up the entire score a lot. The shortest note is no longer a 16th, but a 32nd, thus adding 1 NHW to every note. To prevent this, the shortest duration for spacing is not the shortest note in the score, but the most commonly found shortest note. Notes that are even shorter this 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
```



The most common shortest duration is determined as follows: in every measure, the shortest duration is determined. The most common short duration, is taken as the basis for the spacing, with the stipulation that this shortest duration should always be equal to or shorter than 1/8th note. The shortest duration is printed when you run lilypond with `--verbose`. 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 1/8th), is set through `base-shortest-duration`.

In the Introduction it was explained that stem directions influence spacing. This is controlled with `stem-spacing-correction` property in `NoteSpacing`, which are generated for every `Voice` context. The `StaffSpacing` object (generated at `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:



Properties of the `SpacingSpanner` must be overridden from the `\paper` block, since the `SpacingSpanner` is created before any property commands are interpreted.

```
\paper { \context  {
  \ScoreContext
  \override SpacingSpanner #'spacing-increment = #3.0
} }
```

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.

4.6.4 Line breaking

Line breaks are normally computed automatically. They are chosen such 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`.

4.6.5 Page layout

The most basic settings influencing the spacing are `indent` and `linewidth`. They are set in the `\paper` 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 `\paper` block, then the lines are justified at their natural length. This 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 paragraphs. In a paragraph, the last line simply takes its natural length.

The page layout process happens outside the LilyPond formatting engine: variables controlling page layout are passed to the output, and are further interpreted by `lilypond` wrapper program. It responds to the following variables in the `\paper` block. The spacing between systems is controlled with `interscoreline`, its default is 16pt. The distance between the score lines will stretch in order to fill the full page `interscorelinefill` is set to a positive number. In that case `interscoreline` specifies the minimum spacing.

If the variable `lastpagefill` is defined, systems are evenly distributed vertically on the last page. This might produce ugly results in case there are not enough systems on the last page. The `lilypond-book` command ignores `lastpagefill`. See Chapter 6 [lilypond-book manual], page 140 for more information.

Page breaks are normally computed by TEX, so they are not under direct control of LilyPond. However, you can insert commands into the '`.tex`' output to instruct TEX where to break pages. This is done by setting the `between-systems-strings` on the `NonMusicalPaperColumn` where the system is broken. An example is shown in '`input/regression/between-systems.ly`'. The predefined command `\newpage` also does this.

To change the paper size, there are two commands,

```
#(set-default-paper-size "a4")
\paper{
   #(set-paper-size "a4")
}
```

The second one sets the size of the `\paper` block that it's in.

Predefined commands

`\newpage`.

See also

In this manual: Section 5.1 [Invoking lilypond], page 133.

Examples: '`input/regression/between-systems.ly`'.

Internals: `NonMusicalPaperColumn`.

Bugs

LilyPond has no concept of page layout, which makes it difficult to reliably choose page breaks in longer pieces.

4.7 Output details

The default output format is LaTEX, which should be run through LaTEX. Using the option '`-f`' (or '`--format`') other output formats can be selected also, but none of them work reliably.

Now the music is output system by system (a 'system' is a single line from the score, consisting of staves belonging together). From TEX's point of view, a system is an `\hbox` which contains a lowered `\vbox` so that it is centered vertically on the baseline of the text. Between systems, `\interscoreline` is inserted vertically to have stretchable space. The horizontal dimension of the `\hbox` is given by the `linewidth` parameter from LilyPond's `\paper` block.

After the last system LilyPond emits a stronger variant of `\interscoreline` only if the macro `\lilypondpaperlastpagefill` is not defined (flushing the systems to the top of the page). You can avoid that by setting the variable `lastpagefill` in LilyPond's `\paper` block.

It is possible to fine-tune the vertical offset further by defining the macro `\lilypondscoreshift`:

```
\def\lilypondscoreshift{0.25\baselineskip}
```

where `\baselineskip` is the distance from one text line to the next.

Here an example how to embed a small LilyPond file `foo.ly` into running LaTEX text without using the `lilypond-book` script (see Chapter 6 [lilypond-book manual], page 140):

```
\documentclass{article}

\def\lilypondpaperlastpagefill{}
\lineskip 5pt
\def\lilypondscoreshift{0.25\baselineskip}

\begin{document}
This is running text which includes an example music file
\input{foo.tex}
right here.
\end{document}
```

The file '`foo.tex`' has been simply produced with

```
  lilypond-bin foo.ly
```

The call to `\lineskip` assures that there is enough vertical space between the LilyPond box and the surrounding text lines.

5 Invoking LilyPond

This chapter details the technicalities of running LilyPond.

5.1 Invoking lilypond

Nicely titled output is created through a separate program: '`lilypond`' is a script that uses the LilyPond formatting engine (which is in a separate program) and LaTeX to create a nicely titled piece of sheet music, in PDF (Portable Document Format) format.

```
lilypond [option]... file...
```

To have `lilypond` 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`.

`-d,--dependencies`
: Write `Makefile` dependencies for every input file.

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

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

`-m,--no-paper`
: Produce MIDI output only.

`--no-lily`
: Do not run '`lilypond-bin`'. Useful for debugging `lilypond`.

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

`--no-pdf` Do not generate (PDF) or PS.

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

`--psgz` Gzip the postscript file.

`--html` Make a .HTML file with links to all output files.

`--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`, `pagenumber`, `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**!)

5.1.1 Titling layout

`lilypond` extracts the following header fields from the LY files to generate titling; an example demonstrating all these fields is in '`input/test/lilypond-testpage.ly`':

`title` The title of the music. Centered on top of the first page.

`subtitle` Subtitle, centered below the title.

`poet` Name of the poet, left flushed below the subtitle.

`composer` Name of the composer, right flushed below the subtitle.

`meter` Meter string, left flushed below the poet.

`opus` Name of the opus, right flushed below the composer.

`arranger` Name of the arranger, right flushed below the opus.

`instrument`
Name of the instrument, centered below the arranger.

`dedication`
To whom the piece is dedicated.

`piece` Name of the piece, left flushed below the instrument.

`head` A text to print in the header of all pages. It is not called `header`, because `\header` is a reserved word in LilyPond.

`copyright`
A text to print in the footer of the first page. Default is to print the standard footer also on the first page. Note that if the score consists of only a single page, the first page is also the last page, and in this case, the tagline is printed instead of the copyright.

`footer` A text to print in the footer of all but the last page.

`tagline` Line to print at the bottom of last page. The default text is "Engraved by LilyPond *version-number*".

5.1.2 Additional parameters

The `lilypond` program responds to several parameters specified in a `\paper` 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 headers file.
Normally read from the `\header` block. Default value: empty.

`latexpackages`
Specify additional LaTeX packages file. 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 `\paper` 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 `\paper` block.

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

`pagenumber`
: If set to `no`, no page numbers will be printed. If set to a positive integer, start with this value as the first page number.

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

5.2 Invoking the lilypond binary

The formatting system consists of two parts: a binary executable (‘`lilypond-bin`’), which is responsible for the formatting functionality, and support scripts, which post-process the resulting output. Normally, the support scripts are called, which in turn invoke the `lilypond-bin` binary. However, `lilypond-bin` may be called directly as follows.

```
lilypond-bin [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]

We strongly advise against making LilyPond formatting available through a web server. That is, processing input from untrusted users, and returning the resulting PDF file. LilyPond is a big and complex program. It was not written with security in mind. Making it available to the outside world is a huge risk; consider the security implications of

```
#(system "rm -rf /")
\score {
  c4^#(ly:export (ly:gulp-file "/etc/passwd"))
}
```

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

[1] The status of GUILE is not reset across invocations, so be careful not to change any system defaults from within Scheme.

`-f,--format=`*`format`*
: Output format for sheet music. Choices are `tex` (for TEX output, to be processed with plain TEX, or through `lilypond`), `pdftex` for PDFTEX input, `ps` (for PostScript), `scm` (for a Scheme dump), `sk` (for Sketch) and `as` (for ASCII-art).

  This option is only for developers. Only the TEX output of these is usable for real work.

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

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

`-i,--init=`*`file`*
: Set init file to *file* (default: '`init.ly`').

`-m,--no-paper`
: Disable TEX output. If you have a `\midi` definition MIDI output will be generated.

`-M,--dependencies`
: Output rules to be included in Makefile.

`-o,--output=`*`FILE`*
: Set the default output file to *FILE*.

`-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**!)

5.4 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 it 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/1.6.0,{!!/usr/share/texmf}}
```

`GS_LIB` For processing PostScript output (obtained with `-f ps`) with Ghostscript you have to set `GS_LIB` to point to the directory containing library PS files.

`GS_FONTPATH`
: For processing PostScript output (obtained with `-f ps`) with Ghostscript you have to set `GS_FONTPATH` to point to the directory containing PFA files.

  When you print direct PS output, remember to send the PFA files to the printer as well.

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.

5.5 Error messages

Different error messages can appear while compiling a file:

Warning Something looks suspect. If you are requesting something out of the ordinary then you will understand the message, and can ignore it. However, warnings usually indicate that something is wrong with the input file.

Error Something is definitely wrong. The current processing step (parsing, interpreting, or formatting) will be finished, but the next step will be skipped.

Fatal error
: Something is definitely wrong, and LilyPond cannot continue. This happens rarely. The most usual cause is misinstalled fonts.

Scheme error
: Errors that occur while executing Scheme code are caught by the Scheme interpreter. If running with the verbose option (`-V` or `--verbose`) then a call trace is printed of the offending function call.

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.

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 offending line to indicate the column where the error was found. For example,

```
test.ly:2:19: error: not a duration: 5:
  \notes { c'4 e'5
                 g' }
```

5.6 Reporting bugs

If you have input that results in a crash or an erroneous output, then that is a bug. We try respond to bug-reports promptly, and fix them as soon as possible. For this, we need to reproduce and isolate the problem. 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 you use, and on which platform you run it. Send the report to `bug-lilypond@gnu.org`.

5.7 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 refer to the installation instructions for more information.

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

  For both editors, there is also a facility to jump in the input file to the source of errors in the graphical output. See Section 5.8 [Point and click], page 138.

JEdit
: There exists a plugin for jEdit (`http://www.jedit.org/`). Refer to the plugin website (`http://lily4jedit.sourceforge.net`) for more information.

5.8 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:
  - Xdvi, version 22.36 or newer. 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`.
  - KDVI. A dvi viewer for KDE. You need KDVI from KDE 3.0 or newer. Enable option *Inverse search* in the menu *Settings*.

    Apparently, KDVI does not process PostScript specials correctly. Beams and slurs will not be visible in KDVI.
- an editor with a client/server interface (or a lightweight GUI editor):
  - Emacs. Emacs is an extensible 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`.

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, you use `(gnuserve-start)` in your '`.emacs`', 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.

If 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:

```
#(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`.

6 lilypond-book manual

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 pictures 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. A tutorial on using lilypond-book is in Section 2.21 [Integrating text and music], page 28. For more information about LaTEX The not so Short Introduction to LaTeX (`http://www.ctan.org/tex-archive/info/lshort/english/`) provides a introduction to using LaTEX.

6.1 Integrating Texinfo and music

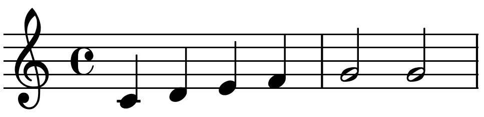
Music is specified like this:

```
@lilypond[options,go,here]
  YOUR LILYPOND CODE
@end lilypond
@lilypond[options,go,here]{ YOUR LILYPOND CODE }
@lilypondfile[options,go,here]{filename}
```

When lilypond-book is run on it, this results in a texinfo file. We show two simple examples here. First a complete block:

```
@lilypond[staffsize=26]
  c' d' e' f' g'2 g'
@end lilypond
```

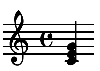
produces



Then the short version:

```
@lilypond[staffsize=11]{<c' e' g'>}
```

produces



When producing texinfo, lilypond-book also generates bitmaps of the music, so you can make a HTML document with embedded music.

6.2 Integrating LaTeX and music

For LaTEX, music is entered using

```
\begin[options,go,here]{lilypond}
  YOUR LILYPOND CODE
\end{lilypond}
\lilypondfile[options,go,here]{filename}
```

or

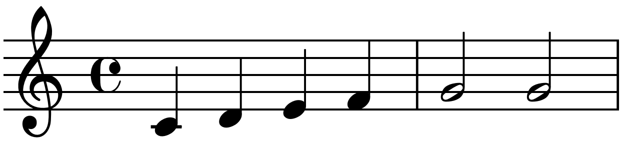
```
\lilypond{ YOUR LILYPOND CODE }
```

Running lilypond-book yields a file that can be processed with LaTeX.

We show some examples here:

```
\begin[staffsize=26]{lilypond}
  c' d' e' f' g'2 g'2
\end{lilypond}
```

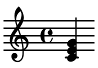
produces



Then the short version:

```
\lilypond[staffsize=11]{<c' e' g'>}
```

produces



The linewidth of the music will be adjust by examining the commands in the document preamble, the part of the document before `\begin{document}`: `lilypond-book` sends these to LaTeX to find out how wide the text is. The line width variable for the music fragments are adjusted to the text width.

After `\begin{document}`, the column changing commands `\onecolumn`, `\twocolumn` commands are also interpreted.

The titling from the `\header` section of the fragments can be imported by adding the following to the top of the LaTeX file:

```
\input titledefs.tex
\def\preLilyPondExample{\def\mustmakelilypondtitle{}}
```

The music will be surrounded by `\preLilyPondExample` and `\postLilyPondExample`, which are defined to be empty by default.

For printing the LaTeX document, you will need to use dvips. For producing PostScript with scalable fonts, add the following options to the dvips command line:

```
 -Ppdf -u +lilypond.map
```

PDF can then be produced with `ps2pdf`.

LilyPond does not use the LaTeX font handling scheme for lyrics and text markups, so if you use characters in your lilypond-book documents that are not included in the standard US-ASCII character set, include `\usepackage[latin1]{inputenc}` in the file header but do not include `\usepackage[[T1]{fontenc}`. Character sets other than latin1 are not supported directly but may be handled by explicitly specifying the `font-name` property in LilyPond and using the corresponding LaTeX packages. Please consult the mailing list for more details.

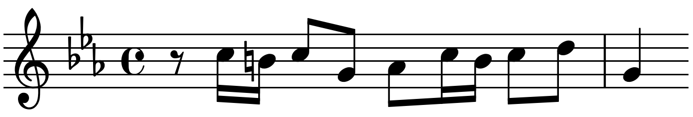
6.3 Integrating HTML and music

Music is entered using

```
<lilypond relative=1 verbatim>
  \key c \minor r8 c16 b c8 g as c16 b c8 d | g,4
</lilypond>
```

of which lilypond-book will produce a HTML with appropriate image tags for the music fragments:

```
<lilypond relative=2 verbatim>
  \key c \minor r8 c16 b c8 g as c16 b c8 d | g,4
</lilypond>
```



For inline pictures, use `<lilypond ... />` syntax, e.g.

```
Some music in <lilypond a b c/> a line of text.
```

A special feature not (yet) available in other output formats, is the `<lilypondfile>` tag, for example,

```
<lilypondfile>trip.ly</lilypondfile>
```

This runs '`trip.ly`' through `lilypond` (see also Section 5.1 [Invoking lilypond], page 133), and substitutes a preview image in the output. The image links to a separate HTML file, so clicking it will take the viewer to a menu, with links to images, midi and printouts.

6.4 Music fragment options

The commands for lilypond-book have room to specify one or more of the following options:

`verbatim`
: *contents* is copied into the source, enclosed in a verbatim block; followed by any text given with the `intertext` option; then the actual music is displayed. This option does not work with the short version of the music blocks:

  `@lilypond{ CONTENTS }` and `\lilypond{ CONTENTS }`

`filename=`*`filename`*
: This names the file for the `printfilename` option. The argument should be unquoted.

`staffsize=`*`ht`*
: Sets the staff height to *ht*, which is measured in points.

`raggedright`
: produces naturally spaced lines (i.e., `raggedright = ##t`); this works well for small music fragments.

`linewidth=`*`size`*`\`*`unit`*
: sets linewidth to *size*, where *unit* = cm, mm, in, or pt. This option affects LilyPond output, not the text layout.

`notime`
: prevents printing time signature.

`fragment`

`nofragment`
: overrides `lilypond-book` auto detection of what type of code is in the LilyPond block, voice contents, or complete code.

`indent=`*`size`*`\`*`unit`*
: sets indentation of the first music system to *size*, where *unit* = cm, mm, in, or pt. This option affects LilyPond, not the text layout. For single-line fragments, the default is to use no indentation.

  For example

```
\begin[indent=5\cm,raggedright]{lilypond}
...
\end{lilypond}
```

`noindent` sets indentation of the first music system to zero. This option affects LilyPond, not the text layout.

`quote` sets linewidth to the width of a quotation and puts the output in a quotation block.

`texidoc` Includes the `texidoc` field, if defined in the file. This is only for Texinfo output.

In Texinfo, the music fragment is normally preceded by the `texidoc` field from the `\header`. The LilyPond test documents are composed from small '`.ly`' files in this way:

```
\header {
  texidoc = "this file demonstrates a single note"
}
\score { \notes { c'4 } }
```

`relative, relative=`*`N`*

uses relative octave mode. By default, notes are specified relative to middle C. The optional integer argument specifies the octave of the

`relative, relative=`*`N`*

uses 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.

6.5 Invoking lilypond-book

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 && latex yourfile.tex
```

`lilypond-book` accepts the following command line options:

'`-f` *`format`*', '`--format=`*`format`*'

Specify the document type to process: `html`, `latex` or `texi` (the default). `lilypond-book` figures this out 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.

'`-F` *`filter`*', '`--filter=`*`filter`*'

Pipe snippets through *filter*.

For example:

```
lilypond-book --filter='convert-ly --from=2.0.0' my-book.tely
```

'`--help`' Print a short help message.

'`-I` *`dir`*', '`--include=`*`dir`*'

Add *DIR* to the include path.

'`-o` *`dir`*', '`--output=`*`dir`*'

Place generated files in *dir*.

'`-P` *`process`*', '`--process=`*`COMMAND`*'

Process lilypond snippets using *command*. The default command is *lilypond-bin*.

'`--verbose`'

Be verbose.

'--version'
: Print version information.

For LaTEX input, the file to give to LaTEX has extension '.latex'. Texinfo input will be written to a file with extension '.texi'.

6.6 Bugs

The LaTEX `\includeonly{...}` command is ignored.

The Texinfo command `pagesize` is not interpreted. Almost all LaTEX commands that change margins and line widths 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`.

7 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.

7.1 Invoking convert-ly

Convert-ly sequentially applies different conversions to upgrade a LilyPond input file. It uses `\version` statements in the file to detect the old version number. For example, to upgrade all LilyPond files in the current directory and its subdirectories, use

```
convert-ly -e --to=1.3.150 ‘find . -name ’*.ly’ -print‘
```

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 level 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.

Bugs

Not all language changes are handled. Only one output option can be specified.

7.2 Invoking midi2ly

Midi2ly translates a MIDI input 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 you want to import files from a program that has no converter for its native format.

Midi2ly will convert 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. 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.

Hackers who know about signal processing are invited to write a more robust midi2ly. midi2ly is written in Python, using a module written in C to parse the MIDI files.

It is invoked 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.

7.3 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 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 sometimes.

7.4 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.

7.5 Invoking musedata2ly

MuseData (see `http://www.musedata.org/`) is an electronic library of classical music scores, comprising at the time of writing about 800 composition dating from 1700 to 1825. The music is encoded in so-called MuseData format. musedata2ly converts a set of MuseData files to one .ly file, and will include a `\header` field if a ‘`.ref`’ file is supplied. It is invoked as follows:

```
musedata2ly [option]... musedata-files
```

The following options are supported by musedata2ly:

`-h,--help`
: print help

`-o,--output=`*`file`*
: set output filename to *file*

`-v,--version`
: version information

`-r,--ref=`*`reffile`*
: read background information from ref-file *reffile*

Bugs

'`musedata2ly`' converts only a small subset of MuseData.

7.6 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 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.

Unified index

#

#(set-accidental-style 'piano-cautionary) ... 50

'

' ... 32

(

(begin * * * *) ... 48
(end * * * *) ... 48

,

, ... 32

.

. ... 34

/

/ ... 79
/+ ... 79

?

? ... 32

[

[... 47

]

] ... 47

_

_ ... 72

\

\"! ... 60
\< ... 60
\> ... 60
\aeolian ... 40
\alternative ... 61
\arpeggio ... 70, 71
\arpeggioBoth ... 71
\arpeggioBracket ... 71
\arpeggioDown ... 71
\arpeggioUp ... 71
\ascendens ... 102
\auctum ... 102
\autoBeamOff ... 48
\autoBeamOn ... 48
\bar ... 43
\break ... 131
\cadenzaOff ... 43
\cadenzaOn ... 43
\caesura ... 97
\clef ... 40
\context ... 26
\deminutum ... 102
\descendens ... 102
\divisioMaior ... 97
\divisioMaxima ... 97
\divisioMinima ... 97
\dorian ... 40
\dotsBoth ... 35
\dotsDown ... 35
\dotsUp ... 35
\dynamicBoth ... 61
\dynamicDown ... 61
\dynamicUp ... 61
\emptyText ... 57
\f ... 60
\fatText ... 57
\ff ... 60
\fff ... 60
\ffff ... 60
\finalis ... 97
\flexa ... 102
\fp ... 60
\germanChords ... 81
\glissando ... 59
\grace ... 57
\header in LaTeX documents ... 141
\hideNotes ... 32
\hideStaffSwitch ... 71
\inclinatum ... 102
\ionian ... 40
\key ... 40
\locrian ... 40
\lydian ... 40
\lyrics ... 71
\lyricsto ... 72
\major ... 40
\mark ... 82
\melisma ... 73
\melismaEnd ... 73
\mf ... 60
\minor ... 40
\mixolydian ... 40
\mp ... 60
\newpage ... 131
\noBreak ... 131
\normalsize ... 122
\notes ... 38
\oneVoice ... 45, 46
\oriscus ... 102
\override ... 119
\p ... 60
\partial ... 43
\pes ... 102
\phrasingSlurBoth ... 52
\phrasingSlurDown ... 52
\phrasingSlurUp ... 52
\phrygian ... 40
\pp ... 60

\ppp . . . 60
\property, in \lyrics . . . 72
\quilisma . . . 102
\relative . . . 37
\repeat . . . 61
\rfz . . . 60
\scriptBoth . . . 56
\scriptDown . . . 56
\scriptUp . . . 56
\semiGermanChords . . . 81
\set . . . 119
\setEasyHeads . . . 105
\sf . . . 60
\sff . . . 60
\sfz . . . 60
\shiftOff . . . 46
\shiftOn . . . 46
\shiftOnn . . . 46
\shiftOnnn . . . 46
\showStaffSwitch . . . 71
\slurBoth . . . 52
\slurDotted . . . 52
\slurDown . . . 52
\slurSolid . . . 52
\slurUp . . . 52
\small . . . 122
\sp . . . 60
\spp . . . 60
\stemBoth . . . 35
\stemDown . . . 35
\stemUp . . . 35
\stropha . . . 102
\super . . . 127
\tempo . . . 53
\tieBoth . . . 36
\tieDotted . . . 36
\tieDown . . . 36
\tieSolid . . . 36
\tieUp . . . 36
\time . . . 42
\times . . . 36
\tiny . . . 122
\transpose . . . 84
\tupletBoth . . . 36
\tupletDown . . . 36
\tupletUp . . . 36
\unHideNotes . . . 32
\virga . . . 102
\virgula . . . 97
\voiceFour . . . 45, 46
\voiceOne . . . 45, 46
\voiceThree . . . 46
\voiceTwo . . . 46

|

| . . . 38, 43

~

~ . . . 35

1

15ma . . . 41

A

ABC . . . 147
AbsoluteDynamicEvent . . . 61
accacciatura . . . 21
accent . . . 55
accents . . . 19
accessing Scheme . . . 107
acciaccatura . . . 21, 57
Accidental . . . 51, 91
Accidental_engraver . . . 51
AccidentalPlacement . . . 51
accidentals . . . 14, 91
Accidentals . . . 49
additions, in chords . . . 78
adjusting output . . . 9
Adjusting slurs . . . 52
adjusting staff symbol . . . 40
All-backend-properties . . . 115
All-layout-objects . . . 115
allowBeamBreak . . . 48
alto clef . . . 41
ambiguity . . . 62, 75
ambitus . . . 75
Ambitus . . . 76
Ambitus_engraver . . . 75
anacruse . . . 21
anacrusis . . . 43
appoggiatura . . . 21, 57
Arkkra . . . 148
Arpeggio . . . 70
Arpeggio . . . 71
ArpeggioEvent . . . 71
articulation . . . 19
articulations . . . 54
Articulations . . . 54
ASCII-art output . . . 136
aug . . . 79
auto-knee-gap . . . 48
autobeam . . . 48
autoBeaming . . . 48
autoBeamSettings . . . 48
AutoChangeMusic . . . 68
automatic beam generation . . . 48
Automatic beams . . . 47
automatic beams, tuning . . . 48
automatic part combining . . . 86
Automatic staff changes . . . 67
automatic syllable durations . . . 72

B

Backend . . . 119
balance . . . 2
balloon . . . 105
Banter . . . 81
bar check . . . 38
Bar check . . . 38
Bar lines . . . 43
bar lines at start of system . . . 45
bar lines, putting symbols on . . . 83

bar lines, symbols on 82
bar numbers 83
bar-line-interface 171
`barCheckSynchronize` 38
baritone clef 41
BarLine 45
BarNumber 83
`base-shortest-duration` 130
bass clef 41
BassFigure 103
BassFigureEvent 103
Basso continuo 103
Beam 47
Beam 63
beams and line breaks 48
beams, by hand 14
beams, kneed 48
beams, manual 47
beats per minute 53
between staves, distance 128
bibliographic information 24
bigger 124
bitmap 133
blackness 2
block comment 21
bold 124
box 124
brace, vertical 82
bracket 124
bracket, vertical 82
bracketed-y-column 124
brackets 54
BreakEvent 131
breaking lines 130
breaking pages 131
BreathingSign 53, 97
BreathingSignEvent 53, 97
broken arpeggio 70
bug report 9
bugs 138

C

call trace 137
caps 124
cautionary accidental 32
CCARH 147
center-align 124
char 124
choral score 73
choral tenor clef 41
chord entry 78
chord mode 78
chord names 23, 80
chordNameExceptions 80
ChordNames 80, 87, 111
chordNameSeparator 81
chordNoteNamer 81
chordRootNamer 81
chords 20, 23, 80
Chords 77
Chords mode 78
chords, jazz 81
Clef 41
clefs 92
cluster 104
Cluster_spanner_engraver 104
ClusterNoteEvent 104
clusters 79
ClusterSpanner 104
ClusterSpannerBeacon 104
coda 55, 83
coda on bar line 82
Coda Technology 146
coloring, syntax 138
column 124
combine 124
command line options 135
comments 21
`common-shortest-duration` 130
Completion_heads_engraver 39, 40
composer 24
condensing rests 86
context definition 106
crescendo 19, 60
CrescendoEvent 61
cross staff 71
cross staff stem 67
cross staff voice, manual 26
cue notes 122
currentBarNumber 83
custodes 95
custos 95
Custos 96
Custos_engraver 96

D

decrescendo 19, 60
DecrescendoEvent 61
defaultBarType 44
defining markup commands 185
`dim` 79
diminuendo 61
dir-column 124
direction, of dynamics 61
distance between staves 128
distance between staves in piano music 67
divisio 96
divisiones 96
documents, adding music to 140
DotColumn 35
Dots 35
doubleflat 124
DoublePercentRepeat 64
doublesharp 124
downbow 55
DrumNoteEvent 65
drums 64, 65
DrumStaff 65, 67
DrumVoice 65, 67
DrumVoices 65
duration 34
DVI driver 12
DVI file 12
dvilj 12
dvips 12, 141
dynamic 124

DynamicLineSpanner . . . 61
dynamics . . . 19
Dynamics . . . 60
DynamicText . . . 61

E

easy notation . . . 105
editor . . . 138
editors . . . 138
emacs . . . 138
Emacs . . . 138, 139
Emacs mode . . . 138
Engraved by LilyPond . . . 24, 25
Engraver_group_engraver . . . 116
engraving . . . 4
enigma . . . 146
entering notes . . . 31
error . . . 137
error messages . . . 137
errors, message format . . . 137
ETF . . . 146
evaluating Scheme . . . 107
exceptions, chord names. . . . 80
expanding repeats . . . 62
expression . . . 17
extender . . . 72
extender line . . . 22
ExtenderEvent . . . 72
extending lilypond . . . 9
`extra-offset` . . . 117

F

fatal error . . . 137
FDL, GNU Free Documentation License . . . 192
fermata . . . 55
fermata on bar line . . . 82
fermata on multi-measure rest . . . 85
fermatas . . . 83, 104
FiguredBass . . . 87, 103
file searching . . . 136
fill-line . . . 124
Finale . . . 146
finalis . . . 96
finding graphical objects . . . 119
finger . . . 124
finger change . . . 56
finger-interface . . . 120
FingerEvent . . . 57, 119
fingering . . . 19, 56
Fingering . . . 57, 119, 120
fingering-event . . . 119
Fingering_engraver . . . 119, 121
flageolet . . . 55
flags . . . 94
flat . . . 125
FoldedRepeatedMusic . . . 63
follow voice . . . 71
`followVoice` . . . 71
font . . . 2
font magnification . . . 122, 123
font selection . . . 123
font size . . . 122
font size, setting . . . 128
font size, texts . . . 123
font switching . . . 123
font-interface . . . 120
`font-interface` . . . 123
font-interface . . . 126
`font-style` . . . 122
fontsize . . . 125
foot marks . . . 55
footer . . . 134
foreign languages . . . 8
four bar music. . . . 130
fraction . . . 125
french clef . . . 41
Frenched scores . . . 87
Frenched staves . . . 48

G

general-align . . . 125
ghostscript . . . 105, 137
Glissando . . . 59, 60
GlissandoEvent . . . 60
grace notes . . . 21, 57
GraceMusic . . . 59, 184
grand staff . . . 82
GrandStaff . . . 50, 82
graphical object descriptions . . . 119
Gregorian square neumes ligatures . . . 98
grob . . . 120
grob-interface . . . 120
GS_FONTPATH . . . 137
GS_LIB . . . 137
GUILE . . . 107
GVim . . . 138

H

Hairpin . . . 60
Hairpin . . . 61
Hal Leonard . . . 105
halign . . . 125
hbracket . . . 125
header . . . 134
hiding objects . . . 118
Hiding staves . . . 87
Horizontal_bracket_engraver . . . 54
HorizontalBracket . . . 54
hspace . . . 125
html . . . 140
HTML, music in . . . 28
hufnagel . . . 90
huge . . . 125
HyphenEvent . . . 72
hyphens . . . 72

I

idiom . . . 8
ImproVoice . . . 116
indent . . . 17
`indent` . . . 131
index . . . 9
installing LilyPond . . . 136
instrument names . . . 106
InstrumentName . . . 84
internal documentation . . . 9, 119
international characters . . . 141
`interscoreline` . . . 131
`interscorelinefill` . . . 131
invisible objects . . . 118
Invisible rest . . . 33
invoking dvips . . . 141
Invoking LilyPond . . . 135
italic . . . 125
item-interface . . . 120

J

jargon . . . 8
jazz chords . . . 81

K

KDE . . . 138
KDVI . . . 138
Key signature . . . 40
key signature, setting . . . 13
KeyChangeEvent . . . 40
`keySignature` . . . 40
KeySignature . . . 40, 91
kneed beams . . . 48

L

LANG . . . 137
language . . . 8
large . . . 125
`lastpagefill` . . . 131
latex . . . 140
LaTeX, music in . . . 28
latin-i . . . 125
latin1 . . . 141
layers . . . 45
lead sheet . . . 23
Lead sheets . . . 23
left-align . . . 125
Ligature_bracket_engraver . . . 98
LigatureBracket . . . 97
Ligatures . . . 97
lilypond-book and titling . . . 141
lilypond-internals . . . 9
lilypond-mode for Emacs . . . 138
LILYPONDPREFIX . . . 137
line . . . 125
line breaks . . . 130
line comment . . . 21
line-column-location . . . 139
line-location . . . 139
`linewidth` . . . 131
LISP . . . 107
lookup . . . 125
lowering text . . . 127
LyricCombineMusic . . . 73
LyricEvent . . . 72
LyricEvent . . . 75
LyricExtender . . . 72
LyricHyphen . . . 72
lyrics . . . 22, 48, 71
Lyrics . . . 22
Lyrics . . . 72, 73, 74, 87, 111, 167
lyrics and melodies . . . 72
LyricText . . . 72
LyricText . . . 75

M

`m` . . . 79
magnification . . . 122
magnify . . . 125
`maj` . . . 79
majorSevenSymbol . . . 80
manual staff switches . . . 68
marcato . . . 55
mark . . . 82
MarkEvent . . . 83
markletter . . . 125
markup . . . 123
markup text . . . 123
measure lines . . . 43
measure numbers . . . 83
measure repeats . . . 64
measure, partial . . . 43
Measure_grouping_engraver . . . 42
MeasureGrouping . . . 42
Medicaea, Editio . . . 90
melisma . . . 22, 72
Melisma_translator . . . 73
mensural . . . 90
Mensural ligatures . . . 98
Mensural_ligature_engraver . . . 98
meter . . . 42
metronome marking . . . 53
MetronomeChangeEvent . . . 53
mezzosoprano clef . . . 41
MIDI . . . 24, 136, 145
MIDI block . . . 106
minimumFret . . . 76
`modern` style accidentals . . . 50
`modern-cautionary` . . . 50
`modern-voice` . . . 50
`modern-voice-cautionary` . . . 50
modes, editor . . . 138
modifiers, in chords. . . . 79
mordent . . . 55
moving text . . . 127
multi measure rests . . . 85
MultiMeasureRest . . . 86
MultiMeasureRestEvent . . . 86
MultiMeasureRestMusicGroup . . . 86
MultiMeasureRestNumber . . . 86
MultiMeasureRestText . . . 86
MultiMeasureTextEvent . . . 86
multiple voices . . . 25
Mup . . . 148

MuseData 147
Music classes 184
Music entry 37
music expression 17
Music expressions 183
Music properties 184
Music Publisher 148
musical symbols 2
musicglyph 126
musicological analysis 54

N

name of singer 74
natural 126
NEdit 138, 139
`no-reset` accidental style 50
non-empty texts 57
Non-guitar tablatures 77
NonMusicalPaperColumn 131
normal-size-sub 126
normal-size-super 126
normalsize 126
notation, explaining 105
note 126
Note entry 31
note grouping bracket 54
note heads 90
note names, Dutch 31
Note specification 31
note-by-number 126
note-collision-interface 176, 177, 178
Note_heads_engraver 39, 116
NoteCollision 45, 46
NoteColumn 46
NoteEvent 32
NoteEvent 183, 184
NoteGroupingEvent 54
NoteHead 32
NoteHead 105, 188
NoteSpacing 130
number 126
number of staff lines, setting 40

O

octavation 41
open 55
optical spacing 2
options, command line 135
organ pedal marks 55
ornaments 54, 57
ottava 41
OttavaBracket 42
outline fonts 141
output format, setting 136
override 126
OverrideProperty 115

P

`padding` 118
padding 121
page breaks 131
page layout 131, 134
page size 131
paper block 24
`paper` file 128
paper size 131
`papersize` 131
parenthesized accidental 32
part combiner 86
PartCombineMusic 87
Partial 43
partial measure 21, 43
PDF 133
PDFTeX output 136
Pedals 69
percent repeats 64
PercentRepeat 64
PercentRepeatedMusic 64
percussion 64, 65
Petrucci 90
phrasing brackets 54
phrasing marks 52
phrasing slurs 20, 52
phrasing, in lyrics 74
PhrasingSlur 52
PhrasingSlurEvent 52
`piano` accidentals 50
PianoPedalBracket 69
PianoStaff 50, 67
PianoStaff 70
PianoStaff 71, 129
pickup 21
picture 133
Pitch names 31
Pitch_squash_engraver 116, 170
pitches 31
pixmap 133
point and click 138
polymetric scores 113
polyphony 25, 45
portato 55
PostScript 137
PostScript output 136
prall 55
prall, down 55
prall, up 55
prallmordent 55
prallprall 55
preview 133
preview image 142
printing chord names 80
printing postscript 137
Program reference 107
Programming error 137
properties 9
PropertySet 115
punctuation 71

Q

quarter tones . . . 31
QuoteMusic . . . 90
quotes, in lyrics . . . 72

R

`r` . . . 35
`R` . . . 85
raise . . . 126
raising text . . . 127
Recording_group_engraver . . . 169
regular line breaks . . . 130
regular rhythms . . . 3
regular spacing . . . 3
Rehearsal marks . . . 82
RehearsalMark . . . 83, 123
Relative . . . 37
relative octave specification . . . 37
reminder accidental . . . 32
removals, in chords . . . 78
RemoveEmptyVerticalGroup . . . 88
removing objects . . . 118
repeat bars . . . 43
repeatCommands . . . 44
`repeatCommands` . . . 62
RepeatedMusic . . . 63
RepeatedMusic . . . 184
repeats . . . 61
RepeatSlash . . . 64
reporting bugs . . . 138
Rest . . . 33, 91
RestCollision . . . 46
RestEvent . . . 33
rests . . . 10, 91
Rests . . . 33
Rests, multi measure . . . 85
reverseturn . . . 55
RevertProperty . . . 115
RhythmicStaff . . . 64
right-align . . . 126
roman . . . 126
root of chord . . . 78

S

`s` . . . 35
sans . . . 126
SATB . . . 73
Scalable fonts . . . 133
Scheme . . . 9, 107
Scheme dump . . . 136
Scheme error . . . 137
Scheme, in-line code . . . 107
Score . . . 42, 109, 111, 164, 166
screen shot . . . 133
Script . . . 56
script on multi-measure rest . . . 85
ScriptEvent . . . 56
scripts . . . 54, 56
search in manual . . . 8
search path . . . 136
segno . . . 55, 83
segno on bar line . . . 82
self-alignment-interface . . . 120
semi-flats, semi-sharps . . . 31
semiflat . . . 127
semisharp . . . 127
SeparatingGroupSpanner . . . 130
SeparationItem . . . 130
SequentialMusic . . . 184
sesquiflat . . . 127
sesquisharp . . . 127
`set-accidental-style` . . . 49
setting object properties . . . 117
sharp . . . 127
shorten measures . . . 43
side-position-interface . . . 120
signature line . . . 25
simple . . . 127
SimultaneousMusic . . . 183
singer's names . . . 74
Sketch output . . . 136
Skip . . . 33
SkipEvent . . . 34
SkipMusic . . . 34
skipTypesetting . . . 39
slur . . . 19
Slur . . . 51
Slur . . . 52
SlurEvent . . . 52
Slurs . . . 51
slurs versus ties . . . 19
small . . . 127
smaller . . . 127
snippets . . . 9
SoloOneEvent . . . 87
SoloTwoEvent . . . 87
Songs . . . 22
soprano clef . . . 41
sound . . . 24
Sound . . . 105
source specials . . . 138
space between staves . . . 128
Space note . . . 33
spaces, in lyrics . . . 72
`spacing` . . . 130
spacing-spanner-interface . . . 179, 180
SpacingSpanner . . . 129, 130
SpacingSpanner, overriding properties . . . 130
SpanBar . . . 45
specials, source . . . 138
Square neumes ligatures . . . 98
staccatissimo . . . 55
staccato . . . 19, 55
Staff . . . 34, 40, 45, 50, 54, 68
Staff . . . 75
Staff . . . 83, 87, 96, 109, 111
Staff . . . 114
Staff . . . 115, 116, 130, 145, 164
staff distance . . . 128
staff group . . . 82
staff lines, setting number of . . . 40
staff lines, setting thickness of . . . 40
Staff notation . . . 40
staff size, setting . . . 128
staff switch, manual . . . 26, 68
staff switching . . . 71
Staff, multiple . . . 82

`Staff.midiInstrument` 106
StaffGroup 44
StaffGroup 82, 83
StaffSpacing 130
StaffSymbol 40, 128
stanza number 74
start of system 45
staves per page 128
Stem 35
Stem 94, 188
stem, cross staff 67
`stem-spacing-correction` 130
`stemLeftBeamCount` 47
`stemRightBeamCount` 47
StemTremolo 63
StemTremolo 64
stopped 55
StringNumberEvent 76
strut 127
sub 127
subbass clef 41
subdivideBeams 48
subscript 56
super 127
superscript 56
`sus` 79
SustainPedal 69
switches 135
syntax coloring 138
SystemStartBar 45
SystemStartBrace 45
SystemStartBracket 45

T

Tab_note_heads_engraver 77
Tablatures basic 76
TabStaff 76, 77
TabVoice 76
tag line 25
teeny 127
Tempo 53
tenor clef 41
tenuto 55
terminology 8
`texi` 140
texinfo 140
Texinfo, music in 28
TEXMF 137
text items, non-empty 57
text markup 123
text on multi-measure rest 85
Text scripts 57
Text spanners 53
text-balloon-interface 105
text-interface 120, 126
text-script-interface 120
`textheight` 131
TextScript 57
TextScriptEvent 57
TextSpanEvent 54
TextSpanner 54
thickness of staff lines, setting 40
thumb marking 55
thumbnail 133, 142
tie 14
Tie 35
Tie 36
TieEvent 36
ties 35
Time signature 42
time signatures 95
TimeScaledMusic 36
TimeSignature 43, 95
Timing_engraver 43, 166
tiny 127
titles 24, 134
titling and lilypond-book 141
titling in THML 142
trace, Scheme 137
translate 127
translating text 127
Translation 119
transparent objects 118
Transpose 84
TransposedMusic 85
transposition of pitches 84
transposition, instrument 85
transposition, MIDI 85
treble clef 41
tremolo beams 63
tremolo marks 63
TremoloEvent 63
TremoloEvent 64
`tremoloFlags` 63
trill 55
triplets 21, 36
Tunable-context-properties 117
tuning automatic beaming 48
tuplet formatting 36
TupletBracket 36
`tupletNumberFormatFunction` 36
tuplets 21, 36
turn 55
tweaking 119
type1 fonts 141
typeset text 123
typewriter 127
typography 3, 4

U

UnfoldedRepeatedMusic 63
UnisonoEvent 87
UntransposableMusic 85
upbow 55
upright 127
URL 9
using the manual 8

V

varbaritone clef 41
varcoda 55
variables 9
Vaticana, Editio 90
VaticanaStaffContext 102
VaticanaVoiceContext 102
vcenter 128
vertical spacing 128, 131
VerticalAlignment 129
Viewing music 12
vim 138
Vim 138
violin clef 41
VocalName 75
Voice 34, 45
Voice 60
Voice 68, 72, 74
Voice 75
Voice 86, 87, 90, 97, 98, 106, 109, 110, 111
Voice 114
Voice 115, 116, 118, 121, 130, 145
VoiceFollower 71
voices, more – on a staff 25
VoltaBracket 63
VoltaRepeatedMusic 63

W

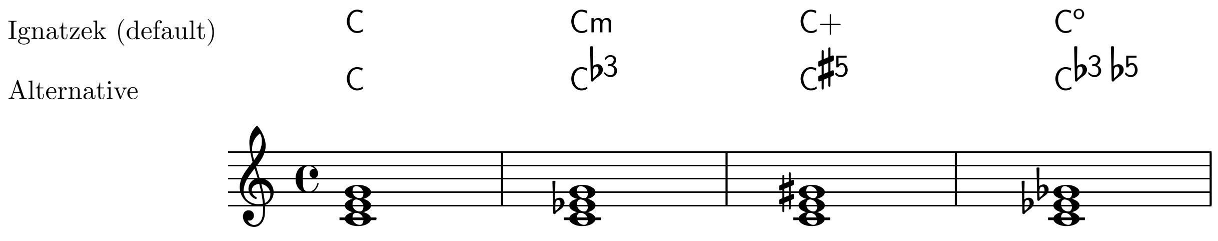
warning 137
website 9
whichBar 44
White mensural ligatures 98
whole rests for a full measure 86
word 128
Writing parts 81

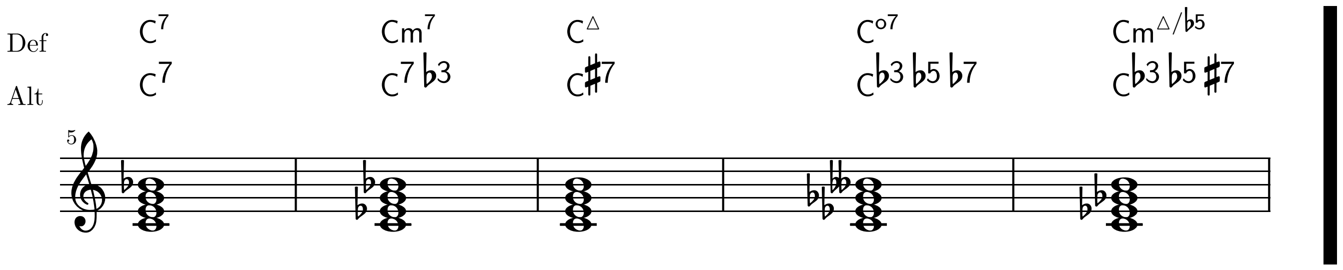
X

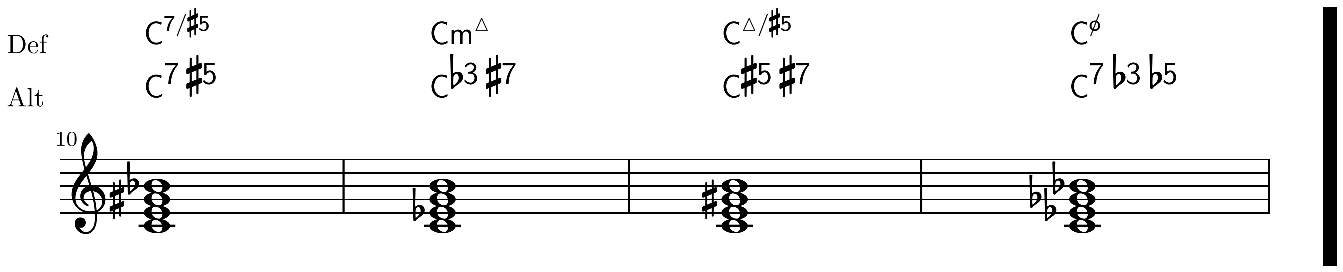
xdvi 12
Xdvi 105, 138
XEDITOR 139
XEmacs 138

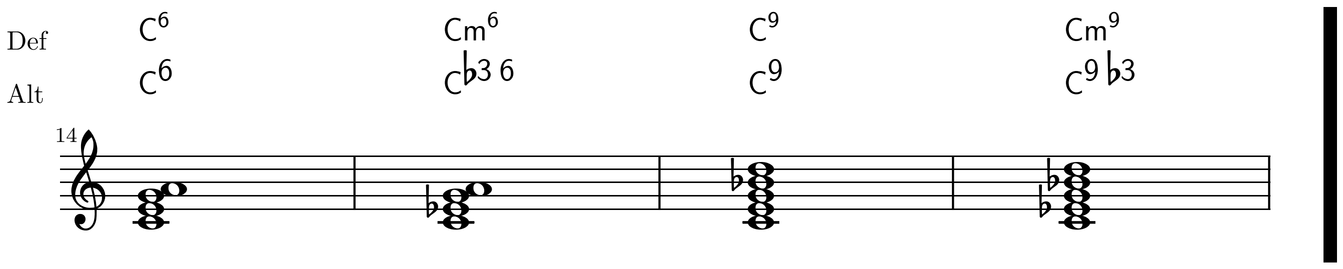
Appendix A Notation manual details

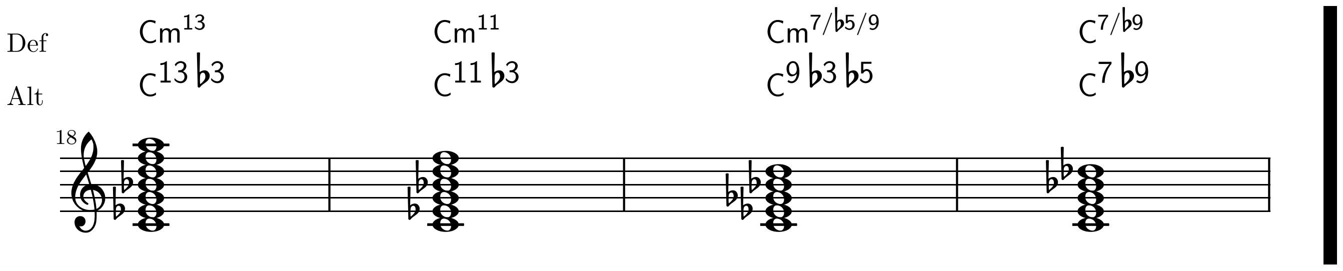
A.1 Chord name chart

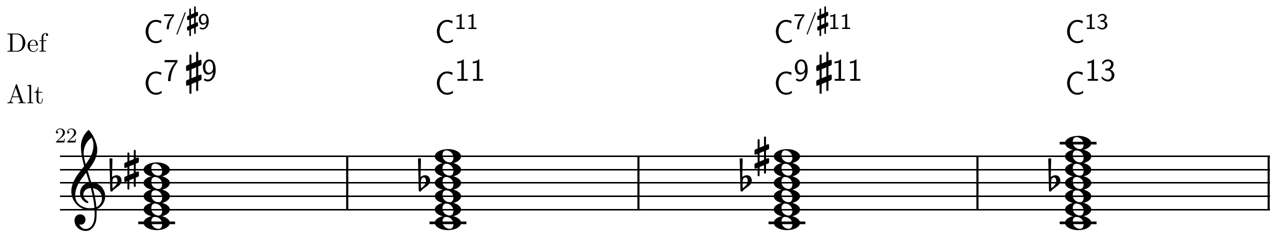


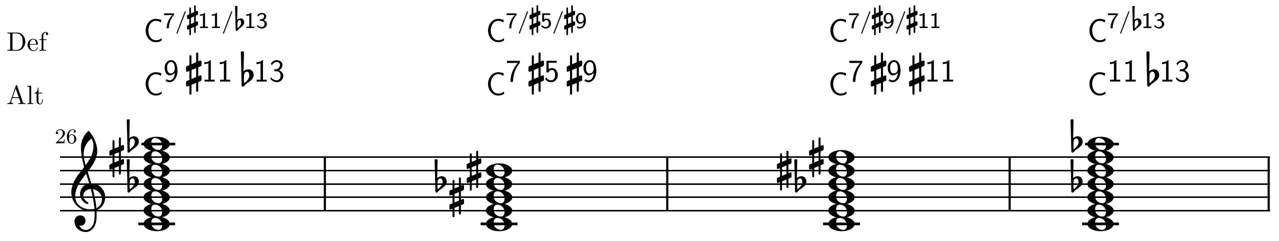


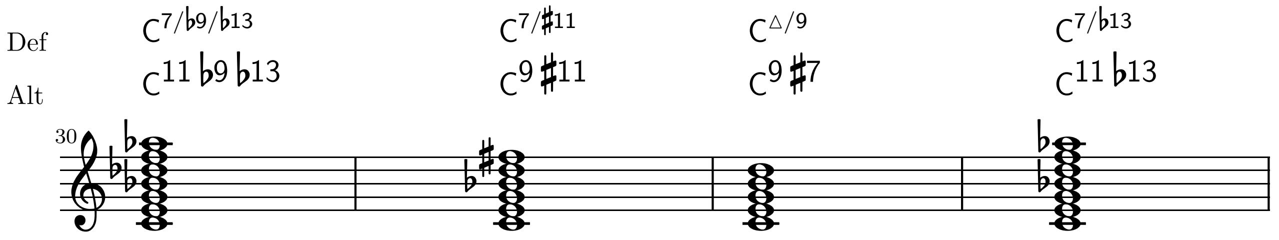


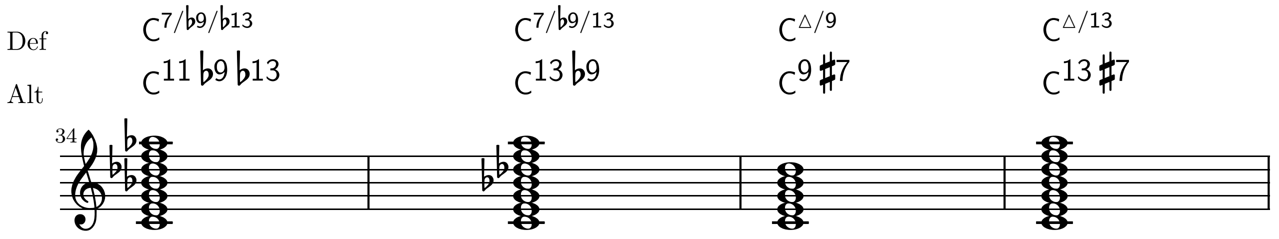


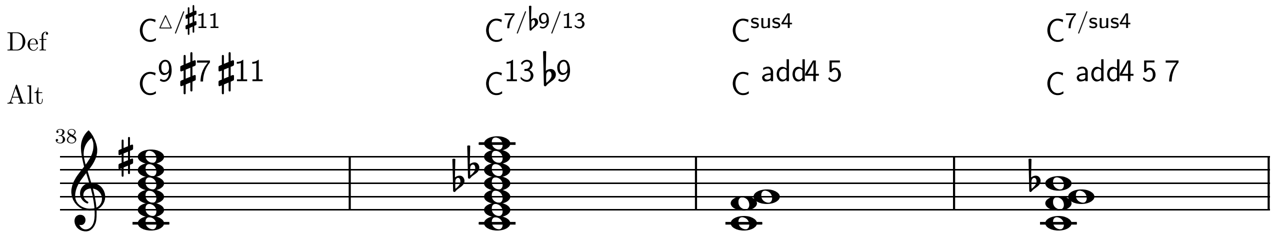


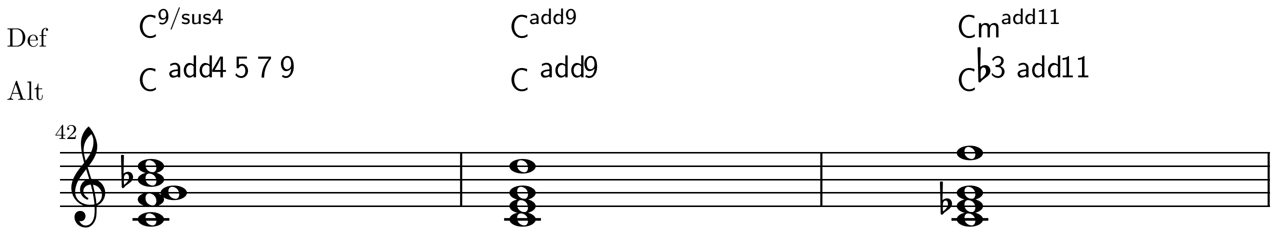












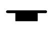
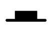
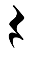
A.2 MIDI instruments

"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 appendixsection"        "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)"
```

A.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 4.5 [Text markup], page 123.

	rests-0		**rests-1**		**rests-0o**
	**rests-1o**		**rests--3**		**rests--2**
	rests--1		**rests-2**		**rests-2classical**

𝄾	rests-3	𝄿	rests-4	𝅀	rests-5
𝅁	rests-6	𝅂	rests-7	𝄰	accidentals-2
𝄲	accidentals-1	𝄰	accidentals-3	♮	accidentals-0
♭	accidentals--2	𝄳	accidentals--1	𝄫	accidentals--4
𝄫	accidentals--3	𝄪	accidentals-4	)	accidentals-rightparen
(	accidentals-leftparen	•	dots-dot	𝅜	noteheads--1
𝅝	noteheads-0	𝅗	noteheads-1	𝅘	noteheads-2
◇	noteheads-0diamond	◇	noteheads-1diamond	◆	noteheads-2diamond
▷	noteheads-0triangle	▷	noteheads-1triangle	▶	noteheads-2triangle
▱	noteheads-0slash	▱	noteheads-1slash	/	noteheads-2slash
✕	noteheads-0cross	✕	noteheads-1cross	×	noteheads-2cross
⊗	noteheads-2xcircle	𝄐	scripts-ufermata	𝄑	scripts-dfermata
Λ	scripts-ushortfermata	V	scripts-dshortfermata	⊓	scripts-ulongfermata
⊔	scripts-dlongfermata	⊓	scripts-uverylongfermata	⊔	scripts-dverylongfermat
ϙ	scripts-thumb	>	scripts-sforzato	•	scripts-staccato
'	scripts-ustaccatissimo	'	scripts-dstaccatissimo	−	scripts-tenuto
∸	scripts-uportato	∸	scripts-dportato	∧	scripts-umarcato
∨	scripts-dmarcato	o	scripts-open	+	scripts-stopped
V	scripts-upbow	⊓	scripts-downbow	𝄢	scripts-reverseturn
𝆗	scripts-turn	𝆖	scripts-trill	∪	scripts-upedalheel
∩	scripts-dpedalheel	V	scripts-upedaltoe	Λ	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--1	↑	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-C4/4	timesig-C2/2	pedal-*
pedal--	pedal-.	pedal-P
pedal-d	pedal-e	pedal-Ped
accordion-accDiscant	accordion-accDot	accordion-accFreebase
accordion-accStdbase	accordion-accBayanbase	accordion-accOldEE
solfa-0do	solfa-1do	solfa-2do
solfa-0re	solfa-1re	solfa-2ro
solfa-0me	solfa-1me	solfa-2me
solfa-0fa	solfa-1fau	solfa-2fau
solfa-1fad	solfa-2fad	solfa-0la
solfa-1la	solfa-2la	solfa-0te
solfa-1te		solfa-2te
rests--3neo_mensural	rests--2neo_mensural	rests--1neo_mensural
rests-0neo_mensural	rests-1neo_mensural	rests-2neo_mensural
rests-3neo_mensural	rests-4neo_mensural	rests--3mensural
rests--2mensural	rests--1mensural	rests-0mensural
rests-1mensural	rests-2mensural	rests-3mensural
rests-4mensural	noteheads-lneo_mensural	noteheads--3neo_mensural
noteheads--2neo_mensural	noteheads--1neo_mensural	noteheads-0neo_mensu[illegible]al
noteheads-0harmonic	noteheads-1neo_mensural	noteheads-2neo_mensural
noteheads-lmensural	noteheads--3mensural	noteheads--2mensural
noteheads--1mensural	noteheads-0mensural	noteheads-1mensural
noteheads-2mensural	noteheads-vaticana_punctum	noteheads-vaticana_punctum_cav[illegible]m
noteheads-vaticana_linea_punctum	noteheads-vaticana_linea_punctum_cavum	noteheads-va
noteheads-vaticana_lpes	noteheads-vaticana_vlpes	noteheads-vaticana_upes
noteheads-vaticana_vupes	noteheads-vaticana_plica	noteheads-vaticana_epipho[illegible]us
noteheads-vaticana_vepiphonus	noteheads-vaticana_reverse_plica	noteheads-vaticana_inne
noteheads-vaticana_cephalicus	noteheads-vaticana_quilisma	noteheads-solesmes_incl_parv

noteheads-solesmes_auct_asc	noteheads-solesmes_auct_desc	noteheads-solesmes_incl_auct
noteheads-solesmes_stropha	noteheads-solesmes_stropha_aucta	noteheads-solesmes_orisc
noteheads-medicaea_inclinatum	noteheads-medicaea_punctum	noteheads-medicaea_rvir
noteheads-medicaea_virga	noteheads-hufnagel_punctum	noteheads-hufnagel_vir a
noteheads-hufnagel_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-neo_mensural_c	clefs-neo_mensural_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-medicaea-1	accidentals-vaticana-1
accidentals-vaticana0	accidentals-mensural1	accidentals-mensural-1
accidentals-hufnagel-1	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-mensural4/4	timesig-mensural2/2
timesig-mensural3/2	timesig-mensural6/4	timesig-mensural9/4
timesig-mensural3/4	timesig-mensural6/8	timesig-mensural9/8
timesig-mensural4/8	timesig-mensural6/8alt	timesig-mensural2/4
timesig-neo_mensural4/4	timesig-neo_mensural2/2	timesig-neo_mensural3/2
timesig-neo_mensural6/4	timesig-neo_mensural9/4	timesig-neo_mensural3/4
timesig-neo_mensural6/8	timesig-neo_mensural9/8	timesig-neo_mensural4/8
timesig-neo_mensural6/8alt	timesig-neo_mensural2/4	scripts-ictus
scripts-uaccentus	scripts-daccentus	scripts-usemicirculus
scripts-dsemicirculus	scripts-circulus	scripts-augmentum
scripts-usignumcongruentiae	scripts-dsignumcongruentiae	

A.4 All context properties

`aDueText` (string)
: Text to print at a unisono passage.

`alignBassFigureAccidentals` (boolean)
: If true, then the accidentals are aligned in bass figure context.

`allowBeamBreak` (boolean)
: If true allow line breaks for beams over bar lines.

`associatedVoice` (string)
: Name of the `Voice` that has the melody for this `Lyrics` line.

`autoAccidentals` (list)
: List of different ways to typeset an accidental.

For determining when to print an accidental, several different rules are tried. The rule that gives the highest number of accidentals is used. Each rule consists of

context: In which context is the rule applied. For example, if *context* is `Score` then all staves share accidentals, and if *context* is `Staff` then all voices in the same staff share accidentals, but staves do not.

octavation:

Whether the accidental changes all octaves or only the current octave. Valid choices are

‘`same-octave:`’

This is the default algorithm. Accidentals are typeset if the note changes the accidental of that note in that octave. Accidentals lasts to the end of the measure and then as many measures as specified in the value. I.e. 1 means to the end of next measure, -1 means to the end of previous measure (that is: no duration at all), etc. #t means forever.

‘`any-octave:`’

Accidentals are typeset if the note is different from the previous note on the same pitch in any octave. The value has same meaning as in same-octave.

laziness

Over how many bar lines the accidental lasts. If *laziness* is `-1` then the accidental is forget immediately, and if *laziness* is `#t` then the accidental lasts forever.

`autoBeamSettings` (list)
Specifies when automatically generated beams should begin and end. See Section 3.5.2 [Setting automatic beam behavior], page 48 for more information.

`autoBeaming` (boolean)
If set to true then beams are generated automatically.

`autoCautionaries` (list)
List similar to `autoAccidentals`, but it controls cautionary accidentals rather than normal ones. Both lists are tried, and the one giving the most accidentals wins. In case of draw, a normal accidental is typeset.

`automaticBars` (boolean)
If set to true then bar lines will not be printed automatically; they must be explicitly created with `\bar` command. Unlike the `\cadenza` keyword, measures are still counted. Bar generation will resume according to that count if this property is unset.

`barAlways` (boolean)
If set to true a bar line is drawn after each note.

`barCheckSynchronize` (boolean)
If true then reset `measurePosition` when finding a barcheck.

`barNumberVisibility` (procedure)
Procedure that takes an int and returns whether the corresponding bar number should be printed

`bassFigureFormatFunction` (procedure)
Procedure that is called to produce the formatting for a `BassFigure` grob. It takes a list of `BassFigureEvents`, a context, and the grob to format.

`beatGrouping` (list)
List of beatgroups, e.g., in 5/8 time `'(2 3)`.

`beatLength` (moment)
The length of one beat in this time signature.

`chordChanges` (boolean)
: Only show changes in chords scheme?

`chordNameExceptions` (list)
: Alist of chord exceptions. Contains (*chord* . *markup*) entries.

`chordNameExceptionsFull` (list)
: Alist of chord exceptions. Contains (*chord* . *markup*) entries.

`chordNameExceptionsPartial` (list)
: Alist of partial chord exceptions. Contains (*chord* . (*prefix-markup suffix-markup*)) entries.

`chordNameFunction` (procedure)
: The function that converts lists of pitches to chord names.

`chordNameSeparator` (markup)
: The markup object used to separate parts of a chord name.

`chordNoteNamer` (procedure)
: Function that converts from a pitch object to a text markup. Used for single pitches.

`chordRootNamer` (procedure)
: Function that converts from a pitch object to a text markup. Used for chords.

`clefGlyph` (string)
: Name of the symbol within the music font.

`clefOctavation` (integer)
: Add this much extra octavation. Values of 7 and -7 are common.

`clefPosition` (number)
: Where should the center of the clef symbol go, measured in half staff spaces from the center of the staff.

`connectArpeggios` (boolean)
: If set, connect arpeggios across piano staff.

`createKeyOnClefChange` (boolean)
: Print a key signature whenever the clef is changed.

`createSpacing` (boolean)
: Create StaffSpacing objects? Should be set for staves.

`crescendoSpanner` (symbol)
: Type of spanner to be used for crescendi. One of: '`hairpin`', '`line`', '`dashed-line`', '`dotted-line`'. If unset, hairpin type is used.

`crescendoText` (markup)
: Text to print at start of non-hairpin crescendo, i.e.: '`cresc.`'

`currentBarNumber` (integer)
: Contains the current barnumber. This property is incremented at every bar line.

`decrescendoSpanner` (symbol)
: See `crescendoSpanner`.

`decrescendoText` (markup)
: Text to print at start of non-hairpin decrescendo, i.e.: '`dim.`'

`defaultBarType` (string)
: Sets the default type of bar line. See `whichBar` for information on available bar types.

  This variable is read by `Timing_engraver` at `Score` level.

`drumPitchTable` (hash table)
: A table mapping percussion instruments (symbols) to pitches.

`drumStyleTable` (hash table)
: A hash table containing mapping drums to layout settings. Predefined values: '`drums-style`', '`timbales-style`', '`congas-style`', '`bongos-style`' and '`percussion-style`'.

  The layout style is a hash table, containing the drum-pitches (e.g. the symbol '`hihat`') as key, and a list (*notehead-style script vertical-position*) as values.

`explicitClefVisibility` (procedure)
: '`break-visibility`' function for clef changes.

`explicitKeySignatureVisibility` (procedure)
: '`break-visibility`' function for explicit key changes. '`\override`' of '`#'break-visibility`' will set the visibility for normal (i.e. at the start of the line) key signatures.

`extraNatural` (boolean)
: Whether to typeset an extra natural sign before accidentals changing from a non-natural to another non-natural.

`extraVerticalExtent` (pair of numbers)
: extra vertical extent, same format as *minimumVerticalExtent*

`fingeringOrientations` (list)
: List of symbols, containing '`left`', '`right`', '`up`' and/or '`down`'. This list determines where fingerings are put relative to the chord being fingered.

`followVoice` (boolean)
: if set, note heads are tracked across staff switches by a thin line

`fontSize` (number)
: The relative size of all grobs in a context.

`forceClef` (boolean)
: Show clef symbol, even if it has not changed. Only active for the first clef after the property is set, not for the full staff.

`harmonicAccidentals` (boolean)
: If set, harmonic notes in chords get accidentals.

`highStringOne` (boolean)
: Whether the 1st string is the string with highest pitch on the instrument. This used by the automatic string selector for tab notation.

`ignoreMelismata` (boolean)
: Ignore melismata for this `Lyrics` line.

`instr` (markup)
: See `instrument`

`instrument` (markup)
: The name to print left of a staff. The `instrument` property labels the staff in the first system, and the `instr` property labels following lines.

`instrumentEqualizer` (procedure)
: Function taking a string (instrument name), and returning a (*min* . *max*) pair of numbers for the loudness range of the instrument.

`instrumentTransposition` (pitch)
: Define the transposition of the instrument. This is used to transpose the MIDI output, and `\quote`s.

`keyAccidentalOrder` (list)
: Alist that defines in what order alterations should be printed. The format is (*name* . *alter*), where *name* is from 0 .. 6 and *alter* from -1, 1.

`keySignature` (list)
: The current key signature. This is an alist containing (*name* . *alter*) or ((*octave* . *name*) . *alter*). where *name* is from 0.. 6 and *alter* from -4 (double flat) to 4 (double sharp).

`majorSevenSymbol` (markup)
: How should the major 7th be formatted in a chord name?

`markFormatter` (procedure)
: Procedure taking as arguments context and rehearsal mark. It should return the formatted mark as a markup object.

`measureLength` (moment)
: Length of one measure in the current time signature.

`measurePosition` (moment)
: How much of the current measure have we had. This can be set manually to create incomplete measures.

`melismaBusyProperties` (list)
: List of properties (symbols) to determine whether a melisma is playing. Setting this property will influence how lyrics are aligned to notes. For example, if set to `#'(melismaBusy beamMelismaBusy)`, only manual melismata and manual beams are considered. Possible values include `melismaBusy`, `slurMelismaBusy`, `tieMelismaBusy`, and `beamMelismaBusy`

`metronomeMarkFormatter` (procedure)
: How to produce a metronome markup. Called with 2 arguments, event and context.

`middleCPosition` (number)
: Place of the middle C, measured in half staffspaces. Usually determined by looking at `clefPosition` and `clefGlyph`.

`midiInstrument` (string)
: Name of the MIDI instrument to use

`midiMaximumVolume` (number)
: Analogous to `midiMinimumVolume`.

`midiMinimumVolume` (number)
: Sets the minimum loudness for MIDI. Ranges from 0 to 1.

`minimumFret` (number)
: The tablature auto string-selecting mechanism selects the highest string with a fret at least *minimumFret*

`minimumVerticalExtent` (pair of numbers)
: minimum vertical extent, same format as *verticalExtent*

`ottavation` (string)
: If set, the text for an ottava spanner. Changing this creates a new text spanner.

`pedalSostenutoStrings` (list)
: See `pedalSustainStrings`.

`pedalSostenutoStyle` (symbol)
: see `pedalSustainStyle`.

`pedalSustainStrings` (list)
: List of string to print for sustain-pedal. Format is (*up updown down*), where each of the three is the string to print when this is done with the pedal.

`pedalSustainStyle` (symbol)
: A symbol that indicates how to print sustain pedals: `text`, `bracket` or `mixed` (both).

`pedalUnaCordaStrings` (list)
: See `pedalSustainStrings`.

`pedalUnaCordaStyle` (symbol)
: see `pedalSustainStyle`.

`printKeyCancellation` (boolean)
: Print restoration alterations before a key signature change.

`printOctaveNames` (boolean)
: Print octave marks for the NoteNames context.

`recordEventSequence` (procedure)
: When `Recording_group_engraver` is in this context, then upon termination of the context, this function is called with current context and a list of music objects. The list of contains entries with start times, music objects and whether they are processed in this context.

`rehearsalMark` (integer)
: The last rehearsal mark printed.

`repeatCommands` (list)
: This property is read to find any command of the form `(volta . x)`, where *x* is a string or `#f`

`restNumberThreshold` (number)
: If a multimeasure rest takes less than this number of measures, no number is printed.

`skipBars` (boolean)
: If set to true, then skip the empty bars that are produced by multimeasure notes and rests. These bars will not appear on the printed output. If not set (the default) multimeasure notes and rests expand into their full length, printing the appropriate number of empty bars so that synchronization with other voices is preserved.

```
@lilypond[fragment,verbatim,center]
r1 r1*3 R1*3  \\property Score.skipBars= ##t r1*3 R1*3
@end lilypond
```

`skipTypesetting` (boolean)
: When true, all no typesetting is done, speeding up the interpretation phase. This speeds up debugging large scores.

`soloADue` (boolean)
: set Solo/A due texts in the part combiner?

`soloIIText` (string)
: text for begin of solo for voice “two” when part-combining.

`soloText` (string)
: text for begin of solo when part-combining.

`squashedPosition` (integer)
: Vertical position of squashing for `Pitch_squash_engraver`.

`stanza` (markup)
: Stanza 'number' to print before the start of a verse. Use in Lyrics context.

`stemLeftBeamCount` (integer)
: Specify the number of beams to draw on the left side of the next note. Overrides automatic beaming. The value is only used once, and then it is erased.

`stemRightBeamCount` (integer)
: See `stemLeftBeamCount`.

`stringOneTopmost` (boolean)
: Whether the 1st string is printed on the top line of the tablature.

`stringTunings` (list)
: The tablature strings tuning. It is a list of the pitch (in semitones) of each string (starting with the lower one).

`subdivideBeams` (boolean)
: If set, multiple beams will be subdivided at beat positions by only drawing one beam over the beat.

`systemStartDelimiter` (symbol)
: Which grob to make for the start of the system/staff? Set to `SystemStartBrace`, `SystemStartBracket` or `SystemStartBar`.

`tablatureFormat` (procedure)
: Function formatting a tab note head; it takes a string number, a list of string tunings and Pitch object. It returns the text as a string.

`timeSignatureFraction` (pair of numbers)
: pair of numbers, signifying the time signature. For example `#'(4 . 4)` is a 4/4 time signature.

`timing` (boolean)
: Keep administration of measure length, position, bar number, etc? Switch off for cadenzas.

`tonic` (pitch)
: The tonic of the current scale

`tremoloFlags` (integer)
: Number of tremolo flags to add if no number is specified.

`tupletNumberFormatFunction` (procedure)
: Function taking a music as input, producing a string. This function is called to determine the text to print on a tuplet bracket.

`tupletSpannerDuration` (moment)
: Normally a tuplet bracket is as wide as the `\times` expression that gave rise to it. By setting this property, you can make brackets last shorter. Example

```
@lilypond[verbatim,fragment]
context Voice \times 2/3 {
  property Voice.tupletSpannerDuration = #(ly:make-moment 1 4)
  c-[8 c c-] c-[ c c-]
}
@end lilypond
```

.

`verticalAlignmentChildCallback` (procedure)
: What callback to add to children of a vertical alignment. It determines what procedure is used on the alignment itself.

`verticalExtent` (pair of numbers)
: Hard coded vertical extent. The format is a pair of dimensions, for example, this sets the sizes of a staff to 10 (5+5) staffspaces high.

```
\set Staff.verticalExtent = #'(-5.0 . 5.0)
```

This does not work for Voice or any other context that doesn't form a vertical group.

`vocNam` (markup)
: Name of a vocal line, short version.

`vocalName` (markup)
: Name of a vocal line.

`voltaOnThisStaff` (boolean)
: Normally, volta brackets are put only on the topmost staff. Setting this variable will create a bracket on this staff as well.

`voltaSpannerDuration` (moment)
: This specifies the maximum duration to use for the brackets printed for `\alternative`. This can be used to shrink the length of brackets in the situation where one alternative is very large.

`whichBar` (string)
: This property is read to determine what type of bar line to create.

Example:

```
\set Staff.whichBar = "|:"
```

This will create a start-repeat bar in this staff only. Valid values are described in `bar-line-interface`.

A.5 Layout properties

`X-extent` (pair of numbers)
: Hard coded extent in X direction.

`X-extent-callback` (procedure)
: Procedure that calculates the extent of this object. If this value is set to `#f`, the object is empty in the X direction. The procedure takes a grob and axis argument, and returns a number-pair.

`X-offset-callbacks` (list)
: A list of functions determining this objects' position relative to its parent. The last one in the list is called first. The functions take a grob and axis argument.

`Y-extent` (pair of numbers)
: See `X-extent`.

`Y-extent-callback` (procedure)
: see `X-extent-callback`.

`Y-offset-callbacks` (list)
: see `X-offset-callbacks`.

`accidentals` (list)
: List of alteration numbers.

`align-dir` (direction)
: Which side to align? `-1`: left side, `0`: around center of width, `1`: right side.

`arch-angle` (number)
: Turning angle of the hook of a system brace

`arch-height` (dimension, in staff space)
: Height of the hook of a system brace.

`arch-thick` (number)
: Thickness of the hook of system brace.

`arch-width` (dimension, in staff space)
: Width of the hook of a system brace.

`arpeggio-direction` (direction)
: If set, put an arrow on the arpeggio squiggly line.

`attachment` (pair)
: Pair of symbols indicating how a slur should be attached at the ends. The format is '(*left-type* . *right-type*), where both *type*s are symbols. The values of these symbols may be `alongside-stem`, `stem`, `head` or `loose-end`.

`attachment-offset` (pair)
: cons of offsets, '(*left-offset* . *right-offset*). This offset is added to the attachments to prevent ugly slurs. [fixme: we need more documentation here].

`auto-knee-gap` (dimension, in staff space)
: If a gap is found between note heads where a horizontal beam fits that is larger than this number, make a kneed beam.

`avoid-note-head` (boolean)
: If set, the stem of a chord does not pass through all note heads, but starts at the last note head.

`axes` (list)
: list of axis numbers. In the case of alignment grobs, this should contain only one number.

`balloon-original-callback` (procedure)
: The original stencil drawer to draw the balloon around.

`balloon-padding` (dimension, in staff space)
: Text to add to help balloon

`balloon-text` (markup)
: Text to add to help balloon

`balloon-text-offset` (pair of numbers)
: Where to put text relative to balloon.

`balloon-text-props` (list)
: Font properties for balloon text.

`bar-size` (dimension, in staff space)
: size of a bar line.

`bar-size-procedure` (procedure)
: Procedure that computes the size of a bar line.

`base-shortest-duration` (moment)
: Spacing is based on the shortest notes in a piece. Normally, pieces are spaced as if notes at least as short as this are present.

`baseline-skip` (dimension, in staff space)
: Distance between base lines of multiple lines of text.

`beam-thickness` (dimension, in staff space)
: thickness, measured in staffspace.

`beam-width` (dimension, in staff space)
: width of the tremolo sign.

`beamed-extreme-minimum-free-lengths` (list)
: list of extreme minimum free stem lengths (chord to beams) given beam multiplicity.

`beamed-lengths` (list)
: list of stem lengths given beam multiplicity .

`beamed-minimum-free-lengths` (list)
: list of normal minimum free stem lengths (chord to beams) given beam multiplicity.

`beamed-stem-shorten` (list)
: How much to shorten beamed stems, when their direction is forced. It is a list, since the value is different depending on the number flags/beams.

`beaming` (pair)
: Pair of number lists. Each number list specifies which beams to make. 0 is the central beam, 1 is the next beam toward the note etc. This information is used to determine how to connect the beaming patterns from stem to stem inside a beam.

`beautiful` (number)
: number that dictates when a slur should be de-uglyfied. It correlates with the enclosed area between noteheads and slurs. A value of 0.1 yields only undisturbed slurs, a value of 5 will tolerate quite high blown slurs.

`before-line-breaking-callback` (procedure)
: This procedure is called before line breaking, but after splitting breakable items at potential line breaks.

`between-cols` (pair)
: Where to attach a loose column to

`between-system-string` (string)
: string to dump between two systems. Useful for forcing page breaks.

`bracket-flare` (pair of numbers)
: A pair of numbers specifying how much edges of brackets should slant outward. Value 0.0 means straight edges

`bracket-thick` (number)
: width of a system start bracket.

`bracket-visibility` (boolean or symbol)
: This controls the visibility of the tuplet bracket. Setting it to false will prevent printing of the bracket. Setting the property to `'if-no-beam` will make it print only if there is no beam associated with this tuplet bracket.

`break-align-orders` (hash table)
: Defines the order in which prefatory matter (clefs, key signatures) appears. The format is a vector of length 3, where each element is one order for end-of-line, middle of line, and start-of-line respectively. An order is a list of symbols.

  For example, clefs are put after key signatures by setting

```
\override Score.BreakAlignment #'break-align-orders = #(make-vector 3
  '(span-bar
  breathing-sign
  staff-bar
  key
  clef
  time-signature))
```

`break-align-symbol` (symbol)
: This key is used for aligning and spacing breakable items.

`break-glyph-function` (procedure)
: This function determines the appearance of a bar line at the line break. It takes a glyph and break-direction and returns the glyph at a line break.

`break-visibility` (procedure)
: A function that takes the break direction and returns a cons of booleans containing (*transparent . empty*). The following variables are predefined: `all-visible`, `begin-of-line-visible`, `end-of-line-visible`, `begin-of-line-invisible`, `end-of-line-invisible`, `all-invisible`.

`breakable` (boolean)
: Can this object appear at a line break, like clefs and bar lines?

`c0-position` (integer)
: An integer indicating the position of middle C.

`cautionary` (boolean)
: Is this a cautionary accidental?

`cautionary-style` (symbol)
: How to print cautionary accidentals. Choices are `smaller` or `parentheses`.

`collapse-height` (dimension, in staff space)
: Minimum height of system start delimiter. If equal or smaller, the bracket is removed.

`common-shortest-duration` (moment)
: The most common shortest note length. This is used in spacing. Enlarging this will set the score tighter.

`concaveness-gap` (dimension, in staff space)
: A beam is printed horizontally if its concaveness-gap is larger than this value. The concaveness-gap is the distance of an inner note head to the line between two outer note heads.

`concaveness-threshold` (number)
: A beam is printed horizontally if its concaveness is bigger than this threshold.

  Concaveness is calculated as the sum of the vertical distances of inner note heads that fall outside the interval of the two outer note heads, to the vertically nearest outer note head, divided by the square of the inner notes involved.

`control-points` (list)
: List of 4 offsets (number-pairs) that form control points for the tie/slur shape.

`damping` (integer)
: Amount of beam slope damping. 0: no, 1: yes, 100000: horizontal beams.

`dash-fraction` (number)
: Size of the dashes, relative to dash-period. Should be between 0.0 (no line) and 1.0 (continuous line).

`dash-period` (number)
: the length of one dash `+` white space. If negative, no line is drawn at all.

`dashed` (number)
: number representing the length of the dashes.

`dir-function` (procedure)
: The function to determine the direction of a beam. Choices include:

  `beam-dir-majority`
  : number count of up or down notes

  `beam-dir-mean`
  : mean center distance of all notes

  `beam-dir-median.`
  : mean center distance weighted per note

`direction` (direction)
: Up or down, left or right?

`dot-count` (integer)
: The number of dots.

`duration-log` (integer)
: The 2-log of the note head duration, i.e. 0=whole note, 1 = half note, etc.

`edge-height` (pair)
: A pair of number specifying the heights of the vertical edges '(*left-height* . *right-height*).

`edge-text` (pair)
: A pair specifying the texts to be set at the edges '(*left-text* . *right-text*).

`enclose-bounds` (number)
: How much of the bound a spanner should enclose: `+`1 = completely, 0 = center, -1 not at all.

`expand-limit` (integer)
: maximum number of measures expanded in church rests.

`extra-X-extent` (pair of numbers)
: A grob is enlarged in X dimension by this much.

`extra-Y-extent` (pair of numbers)
: See `extra-Y-extent`.

`extra-offset` (pair of numbers)
: A pair representing an offset. This offset is added just before outputting the symbol, so the typesetting engine is completely oblivious to it.

`extremity-function` (procedure)
: A function that calculates the attachment of a slur-end. The function takes a slur and direction argument and returns a symbol.

`extremity-offset-alist` (list)
: The offset adds to the centre of the note head, or stem.

  Format: alist (attachment stem-dir*dir slur-dir*dir) `->` offset.

`flag-count` (number)
: The number of tremolo beams.

`flag-style` (symbol)
: a string determining what style of flag-glyph is typeset on a Stem. Valid options include `()` and `mensural`. Additionally, `"no-flag"` switches off the flag.

`flag-width-function` (procedure)
: Procedure that computes the width of a half-beam (a non-connecting beam.).

`font-encoding` (symbol)
: The font encoding is the broadest category for selecting a font. Options include: `music`, `number`, `text`, `math`, `braces`, `dynamic`

`font-family` (symbol)
: The font family is the broadest category for selecting text fonts. Options include: `sans`, `roman`

`font-magnification` (number)
: Magnification of the font, when it is selected with `font-name`.

`font-name` (string)
: Specifies a file name (without extension) of the font to load. This setting override selection using `font-family`, `font-series` and `font-shape`.

`font-series` (symbol)
: Select the series of a font. Choices include `medium`, `bold`, `bold-narrow`, etc.

`font-shape` (symbol)
: Select the shape of a font. Choices include `upright`, `italic`, `caps`.

`font-size` (number)
: The font size, compared the 'normal' size. 0 is style-sheet's normal size, -1 is smaller, +1 is bigger. Each step of 1 is approximately 12% larger, 6 steps are exactly a factor 2 larger. Fractional values are allowed.

`force-hshift` (number)
: This specifies a manual shift for notes in collisions. The unit is the note head width of the first voice note. This is used by `note-collision-interface`.

`forced-distance` (dimension, in staff space)
: A fixed distance between object reference points in an alignment.

`fraction` (pair of numbers)
: Numerator and denominator of a time signature object.

`french-beaming` (boolean)
: Use French beaming style for this stem. The stem will stop at the innermost beams.

`full-size-change` (boolean)
: Don't make a change clef smaller.

`gap` (dimension, in staff space)
: Size of a gap in a variable symbol.

`gap-count` (integer)
: Number of gapped beams for tremolo.

`glyph` (string)
: a string determining what (style) of glyph is typeset. Valid choices depend on the function that is reading this property.

`glyph-name` (string)
: a name of character within font.

`glyph-name-procedure` (procedure)
: Return the name of a character within font, to use for printing a symbol.

`grace-space-factor` (number)
: Space grace notes at this fraction of the `spacing-increment`.

`grow-direction` (direction)
: Crescendo or decrescendo?

`hair-thickness` (number)
: Thickness of the thin line in a bar line.

`height` (dimension, in staff space)
: Height of an object in staffspace.

`height-limit` (dimension, in staff space)
: Maximum slur height: the longer the slur, the closer it is to this height.

`horizontal-shift` (integer)
: An integer that identifies ranking of note-column for horizontal shifting. This is used by `note-collision-interface`.

`kern` (dimension, in staff space)
: Amount of extra white space to add. For bar line, this is the amount of space after a thick line.

`knee` (boolean)
: Is this beam kneed?

`knee-spacing-correction` (number)
: Factor for the optical correction amount for kneed beams. Set between 0 for no correction and 1 for full correction.

`layer` (number)
: The output layer [0..2]: layers define the order of printing objects. Objects in lower layers are overprinted by objects in higher layers.

`ledger-line-thickness` (pair of numbers)
: The thickness of ledger lines: it is the sum of 2 numbers. The first is the factor for line thickness, and the second for staff space. Both contributions are added.

`left-padding` (dimension, in staff space)
: The amount of space that is put left to a group of accidentals.

`left-position` (number)
: Vertical position of left part of spanner.

`length` (dimension, in staff space)
: User override for the stem length of unbeamed stems.

`lengths` (list)
: Default stem lengths. The list gives a length for each flag-count.

`line-count` (integer)
: The number of staff lines.

`measure-count` (integer)
: The number of measures for a multimeasure rest.

`measure-length` (moment)
: Length of a measure. Used in some spacing situations.

`merge-differently-dotted` (boolean)
: Merge note heads in collisions, even if they have a different number of dots. This normal notation for some types of polyphonic music.

`merge-differently-headed` (boolean)
: Merge note heads in collisions, even if they have different note heads. The smaller of the two heads will be rendered invisible. This used polyphonic guitar notation. The value of this setting is used by `note-collision-interface` .

`minimum-X-extent` (pair of numbers)
: Minimum size of an object in X dimension, measured in staff space.

`minimum-Y-extent` (pair of numbers)
: See `minimum-Y-extent`.

`minimum-distance` (dimension, in staff space)
: Minimum distance between rest and notes or beam.

`minimum-length` (dimension, in staff space)
: Try to make a spanner at least this long. This requires an appropriate routine for the `spacing-procedure` property.

`minimum-space` (dimension, in staff space)
: Minimum distance that the victim should move (after padding).

`neutral-direction` (direction)
: Which direction to take in the center of the staff.

`neutral-position` (number)
: Position (in half staff spaces) where to flip the direction of custos stem.

`new-accidentals` (list)
: List of `(`*`pitch`*` . `*`accidental`*`)` pairs.

`no-spacing-rods` (boolean)
: Items with this property do not cause spacing constraints.

`no-stem-extend` (boolean)
: If set, notes with ledger lines do not get stems extending to the middle staff line.

`non-default` (boolean)
: Set for manually specified clefs.

`note-names` (hash table)
: Vector of strings containing names for easy-notation note heads.

`number-visibility` (boolean or symbol)
: Like `bracket-visibility`, but for the number.

`old-accidentals` (list)
: List of `(`*`pitch`*` . `*`accidental`*`) pairs.`

`padding` (dimension, in staff space)
: Add this much extra space between objects that are next to each other.

`penalty` (number)
: Penalty for breaking at this column. 10000 or more means forbid linebreak, -10000 or less means force linebreak. Other values influence linebreaking decisions as a real penalty.

`pitch-max` (pitch)
: Top pitch for ambitus.

`pitch-min` (pitch)
: Bottom pitch for ambitus.

`positions` (pair)
: Pair of staff coordinates `(`*`left`*` . `*`right`*`)`, where both *left* and *right* are in the staff-space unit of the current staff.

`print-function` (procedure)
: Function taking grob as argument, returning a `Stencil` object.

`ratio` (number)
: Parameter for slur shape. The higher this number, the quicker the slur attains it `height-limit`.

`remove-first` (boolean)
: Remove the first staff of a orchestral score?

`right-padding` (dimension, in staff space)
: Space to insert between note and accidentals.

`right-position` (number)
: Vertical position of right part of spanner.

`script-priority` (number)
: A sorting key that determines in what order a script is within a stack of scripts.

`self-alignment-X` (number)
: Specify alignment of an object. The value -1 means left aligned, 0 centered, and 1 right-aligned in X direction. Values in between may also be specified.

`self-alignment-Y` (number)
: like `self-alignment-X` but for Y axis.

`shorten-pair` (pair of numbers)
: The lengths to shorten a text-spanner on both sides, for example a pedal bracket

`shortest-duration-space` (dimension, in staff space)
: Start with this much space for the shortest duration. This is expressed in `spacing-increment` as unit. See also `spacing-spanner-interface`.

`shortest-playing-duration` (moment)
: The duration of the shortest playing here.

`shortest-starter-duration` (moment)
: The duration of the shortest note that starts here.

`side-relative-direction` (direction)
: Multiply direction of `direction-source` with this to get the direction of this object.

`slope` (number)
: The slope of this object.

`slope-limit` (number)
: Set slope to zero if slope is running away steeper than this.

`space-alist` (list)
: A table that specifies distances between prefatory items, like clef and time-signature. The format is an alist of spacing tuples: `(`*`break-align-symbol type . distance`*`)`, where *type* can be the symbols `minimum-space` or `extra-space`.

`space-function` (procedure)
: Calculate the vertical space between two beams. This function takes a beam grob and the maximum number of beams.

`spacing-increment` (number)
: Add this much space for a doubled duration. Typically, the width of a note head. See also `spacing-spanner-interface`.

`spacing-procedure` (procedure)
: Procedure for calculating spacing parameters. The routine is called after `before-line-breaking-callback`.

`stacking-dir` (direction)
: Stack objects in which direction?

`staff-padding` (dimension, in staff space)
: Maintain this much space between reference points and the staff. Its effect is to align objects of differing sizes (like the dynamic **p** and **f**) on their baselines.

`staff-position` (number)
: Vertical position, measured in half staff spaces, counted from the middle line.

`staff-space` (dimension, in staff space)
: Amount of space between staff lines, expressed in global staffspace.

`staffline-clearance` (dimension, in staff space)
: How far away ties keep from staff lines.

`stem-attachment-function` (procedure)
: A function that calculates where a stem attaches to the note head? This is a fallback when this information is not specified in the font. The function takes a grob and axis argument, and returns a $(x\ .\ y)$ pair, specifying location in terms of note head bounding box.

`stem-end-position` (number)
: Where does the stem end (the end is opposite to the support-head.

`stem-shorten` (list)
: How much a stem in a forced direction should be shortened. The list gives an amount depending on the number of flags/beams.

`stem-spacing-correction` (number)
: Optical correction amount for stems that are placed in tight configurations. For opposite directions, this amount is the correction for two normal sized stems that overlap completely.

`stroke-style` (string)
: set to "grace" to turn stroke through flag on.

`style` (symbol)
: This setting determines in what style a grob is typeset. Valid choices depend on the `print-function` that is reading this property.

`text` (markup)
: Text markup. See Section 4.5 [Text markup], page 123.

`thick-thickness` (number)
: Bar line thickness, measured in `linethickness`.

`thickness` (number)
: Bar line thickness, measured in `linethickness`.

`thin-kern` (number)
: The space after a hair-line in a bar line.

`threshold` (pair of numbers)
: (*min* . *max*), where *min* and *max* are dimensions in staff space.

`transparent` (boolean)
: This is almost the same as setting `print-function` to `#f`, but this retains the dimensions of this grob, which means that grobs can be erased individually.

`when` (moment)
: Global time step associated with this column happen?

`width` (dimension, in staff space)
: The width of a grob measured in staff space.

`word-space` (dimension, in staff space)
: space to insert between lyrics or words in texts.

`x-gap` (dimension, in staff space)
: The horizontal gap between note head and tie.

`y-free` (dimension, in staff space)
: The minimal vertical gap between slur and note heads or stems.

`y-offset` (dimension, in staff space)
: Extra vertical offset for ties away from the center line.

`zigzag-length` (dimension, in staff space)
: The length of the lines of a zigzag, relative to `zigzag-width`. A value of 1 gives 60-degree zigzags.

`zigzag-width` (dimension, in staff space)
: The width of one zigzag-squiggle. This number will be adjusted slightly so that the glissando line can be constructed from a whole number of squiggles.

Appendix B Literature list

If you need to know more about music notation, here are some interesting titles to read. The source archive includes a more elaborate BibTeX bibliography of over 100 entries in '`Documentation/bibliography/`'. It is also available online from the website.

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 which covers 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.

Appendix C Interfaces for programmers

C.1 Programmer interfaces for input

C.1.1 Input variables and Scheme

The input format supports the notion of variable: in the following example, a music expression is assigned to a variable with the name `traLaLa`.

```
traLaLa = \notes { c'4 d'4 }
```

There is also a form of scoping: in the following example, the `\paper` block also contains a `traLaLa` variable, which is independent of the outer `\traLaLa`.

```
traLaLa = \notes { c'4 d'4 }
\paper { traLaLa = 1.0 }
```

In effect, each input file is a scope, and all `\header`, `\midi` and `\paper` 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 = \notes { c'4 d'4 }
```

is internally converted to a Scheme definition

```
(define traLaLa Scheme value of ``\notes ... '')
```

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` by means of a second variable `twice`:

```
traLaLa = \notes { c'4 d'4 }

#(define newLa (map ly:music-deep-copy
  (list traLaLa traLaLa)))
#(define twice
  (make-sequential-music newLa))

\score { \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

```
...
\score { #(ly:export (make-sequential-music newLa)) }
```

C.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 interface, 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 in `element` property of `RepeatedMusic`, and the alternatives in `elements`.

C.1.3 Manipulating music expressions

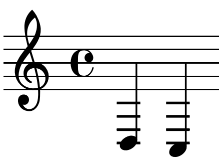
Music objects and their properties can be accessed and manipulated directly, through the `\apply` mechanism. The syntax for `\apply` is

```
\apply #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)
\score { \notes \apply #rev-music-1 { c4 d4 } }
```



The use of such a function is very limited. The effect of this function is void when applied to an argument which is does not have multiple children. The following function application has no effect:

```
\apply #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 `\apply` 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 `\apply` 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/test/to-xml.ly')

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'.

C.2 Markup programmer interface

C.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 C.2.2 [Markup command definition], page 186).

Bugs

One can not feed the `#:line` (resp `#:center`, `#:column`) command with a variable or the result of a function call. E.g.:

```
(markup #:line (fun-that-returns-markups))
```

is illegal. One should use the `make-line-markup` (resp `make-center-markup`, `make-column-markup`) function instead:

```
(markup (make-line-markup (fun-that-returns-markups)))
```

C.2.2 Markup command definition

New markup commands can be defined with the `def-markup-command` scheme macro.

```
(def-markup-command (command-name paper 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
paper	the 'paper' 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 markup type. Therefore, the start of the definition should read

```
(def-markup-command (smallcaps paper props argument) (markup?)
```

What follows is the content of the command: we should interpret the `argument` as a markup, i.e.

```
(interpret-markup paper  ... 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 consing a list with the extra setting.

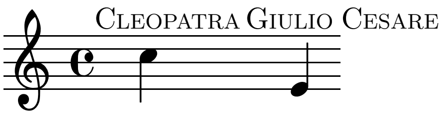
Suppose that we are typesetting a recitative in an opera, and we would like to define a command that will show character names in a custom manner. Names should be printed with small caps and translated a bit to the left and top. We will define a `\character` command that takes into account the needed translation, and uses the newly defined `\smallcaps` command:

```
#(def-markup-command (character paper props name) (string?)
   "Print the character name in small caps, translated to the left and
   top. Syntax: \\character #\"name\""
   (interpret-markup paper props
    (markup "" #:translate (cons -4 2) #: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:

```
\score {
    \notes { \fatText
        c''^\markup \character #"Cleopatra"
        e'^\markup \character #"Giulio Cesare"
    }
}
```



We have used the `caps` font shape, but suppose that our font that 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 paper props str) (string?)
   "Print the string argument in small caps."
   (interpret-markup paper 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 `paper` and `props` arguments.

C.3 Contexts for programmers

C.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)))
```

C.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:

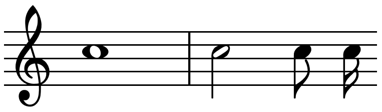
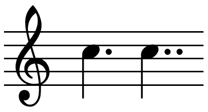
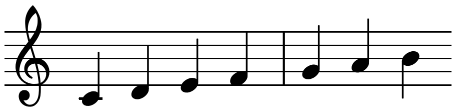
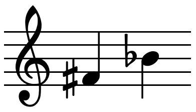
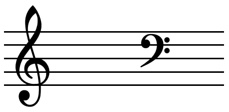
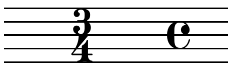
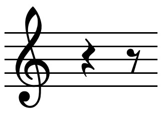
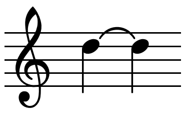
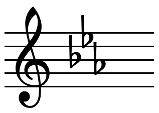
- the layout object itself,
- the context where the layout object was created, and
- the context where `\applyoutput` is processed.

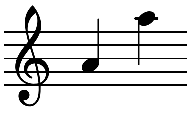
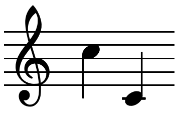
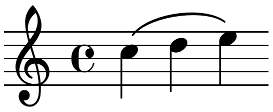
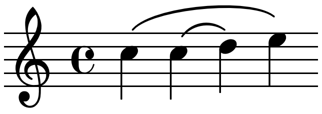
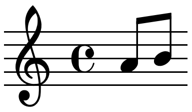
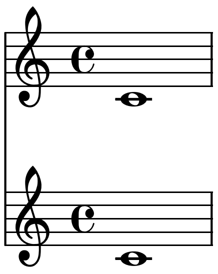
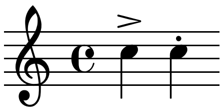
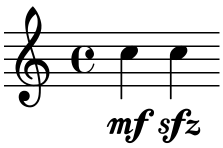
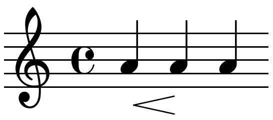
In addition, the cause of the layout object, i.e. the music expression or object that was responsible for creating it, is in the object property `cause`. For example, for a note head, this is a `NoteHead` event, and for a `Stem` object, this is a `NoteHead` object.

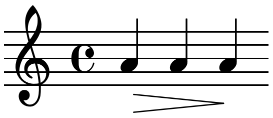
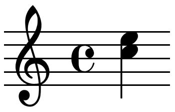
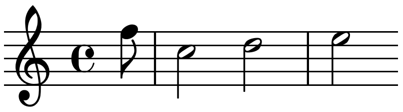
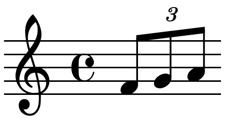
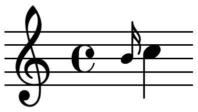
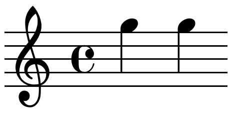
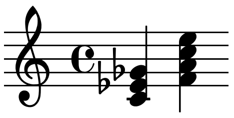
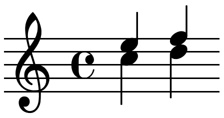
Here is a function to use for `\applyoutput`; it blanks note-heads on the center-line:

```
(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)))
```

Appendix D 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\< b\!`	crescendo	

`a\> b\!`	decrescendo	
`< >`	chord	
`\partial 8`	upstep	
`\times 2/3 {f g a}`	triplets	
`\grace`	grace notes	
`\lyrics { twinkle }`	entering lyrics	
`\new Lyrics`	printing lyrics	twinkle **twinkle**
`twin -- kle`	lyric hyphen	 **twin-kle**
`\chords { c:dim f:maj7 }`	chords	
`\context ChordNames`	printing chord names	C° F△
`<<{e f} \\{c d}>>`	polyphony	
`s4 s8 s16`	spacer rests	

Appendix E GNU Free Documentation License