

GNU LilyPond

The music typesetter

Han-Wen Nienhuys
Jan Nieuwenhuizen
Jürgen Reuter
Rune Zedeler

Copyright © 1999–2003 by the authors

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections. A copy of the license is included in the section entitled “GNU Free Documentation License”.

(For LilyPond version 2.0.3)

Table of Contents

Preface	1
Preface to version 2.0	1
Preface to version 1.8	1
Preface to version 1.6	1
1 Introduction	3
1.1 Notation in LilyPond	3
1.2 Engraving in LilyPond	4
1.3 Typography and program architecture	5
1.4 Music representation	7
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	13
2.4 Octave entry	15
2.5 Combining music into compound expressions	16
2.6 Adding articulation marks to notes	18
2.7 Combining notes into chords	19
2.7.1 Basic rhythmical commands	20
2.7.2 Commenting input files	20
2.8 Printing lyrics	21
2.9 A lead sheet	22
2.10 Listening to output	23
2.11 Titling	23
2.12 Single staff polyphony	24
2.13 Piano staves	24
2.14 Setting variables	25
2.15 Fine tuning layout	25
2.16 Organizing larger pieces	27
2.17 An orchestral part	28
2.18 Integrating text and music	29
3 Notation manual	32
3.1 Note entry	32
3.1.1 Notes	32
3.1.2 Pitches	32
3.1.3 Chromatic alterations	33
3.1.4 Chords	33
3.1.5 Rests	34
3.1.6 Skips	34
3.1.7 Durations	34
3.1.8 Stems	35
3.1.9 Ties	35
3.1.10 Tuplets	36

3.1.11	Easy Notation note heads	37
3.2	Easier music entry	37
3.2.1	Relative octaves	37
3.2.2	Octave check	38
3.2.3	Bar check	39
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	41
3.3.4	Ottava brackets	42
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	46
3.5.2	Setting automatic beam behavior	47
3.6	Accidentals	48
3.6.1	Using the predefined accidental variables	48
3.6.2	Customized accidental rules	51
3.7	Expressive marks	51
3.7.1	Slurs	51
3.7.2	Phrasing slurs	52
3.7.3	Breath marks	53
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	55
3.7.9	Text scripts	56
3.7.10	Grace notes	57
3.7.11	Glissando	59
3.7.12	Dynamics	59
3.8	Repeats	60
3.8.1	Repeat syntax	60
3.8.2	Repeats and MIDI	61
3.8.3	Manual repeat commands	61
3.8.4	Tremolo repeats	62
3.8.5	Tremolo subdivisions	62
3.8.6	Measure repeats	63
3.9	Rhythmic music	63
3.9.1	Percussion staves	63
3.9.2	Percussion MIDI output	65
3.10	Piano music	66
3.10.1	Automatic staff changes	66
3.10.2	Manual staff switches	67
3.10.3	Pedals	67
3.10.4	Arpeggio	68
3.10.5	Staff switch lines	69
3.11	Vocal music	70
3.11.1	Entering lyrics	70

3.11.2	The Lyrics context	71
3.11.3	More stanzas	71
3.11.4	Ambitus	72
3.12	Tablatures	73
3.12.1	Tablatures basic	73
3.12.2	Non-guitar tablatures	74
3.13	Chord names	75
3.13.1	Chords mode	75
3.13.2	Printing chord names	77
3.14	Orchestral music	79
3.14.1	Multiple staff contexts	79
3.14.2	Rehearsal marks	79
3.14.3	Bar numbers	80
3.14.4	Instrument names	80
3.14.5	Transpose	81
3.14.6	Multi measure rests	81
3.14.7	Automatic part combining	82
3.14.8	Hiding staves	83
3.14.9	Different editions from one source	84
3.14.10	Sound output for transposing instruments	85
3.15	Ancient notation	85
3.15.1	Ancient note heads	86
3.15.2	Ancient accidentals	86
3.15.3	Ancient rests	87
3.15.4	Ancient clefs	87
3.15.5	Ancient flags	89
3.15.6	Ancient time signatures	90
3.15.7	Custodes	91
3.15.8	Divisiones	92
3.15.9	Ligatures	92
3.15.9.1	White mensural ligatures	93
3.15.9.2	Gregorian square neumes ligatures	94
3.15.10	Figured bass	98
3.15.11	Vaticana style contexts	99
3.16	Contemporary notation	99
3.16.1	Clusters	100
3.16.2	Fermatas	100
3.17	Tuning output	101
3.17.1	Tuning objects	101
3.17.2	Constructing a tweak	103
3.17.3	Applyoutput	104
3.17.4	Font selection	105
3.17.5	Text markup	106
3.18	Global layout	109
3.18.1	Vertical spacing	109
3.18.2	Horizontal Spacing	109
3.18.3	Font size	111
3.18.4	Line breaking	111
3.18.5	Page layout	111
3.19	Sound	112
3.19.1	MIDI block	112
3.19.2	MIDI instrument names	113

4 Literature list 114

5	Technical manual	116
5.1	Interpretation context	116
5.1.1	Creating contexts	116
5.1.2	Default contexts	117
5.1.3	Context properties	117
5.1.4	Context evaluation	118
5.1.5	Defining contexts	118
5.1.6	Engravers and performers	119
5.1.7	Defining new contexts	119
5.2	Scheme integration	120
5.2.1	Inline Scheme	120
5.2.2	Input variables and Scheme	120
5.2.3	Scheme datatypes	121
5.2.4	Assignments	121
5.3	Music storage format	122
5.3.1	Music expressions	122
5.3.2	Internal music representation	123
5.3.3	Manipulating music expressions	123
5.4	Lexical details	124
5.5	Output details	125
6	Invoking LilyPond	126
6.1	Invoking lilypond	126
6.1.1	Titling layout	127
6.1.2	Additional parameters	127
6.2	Invoking the lilypond binary	128
6.3	Command line options	128
6.4	Environment variables	129
6.5	Error messages	130
6.6	Reporting bugs	130
6.7	Point and click	131
7	lilypond-book manual	133
7.1	Integrating Texinfo and music	133
7.2	Integrating LaTeX and music	134
7.3	Integrating HTML and music	134
7.4	Music fragment options	135
7.5	Invoking lilypond-book	137
7.6	Bugs	138
8	Converting from other formats	139
8.1	Invoking convert-ly	139
8.2	Invoking midi2ly	139
8.3	Invoking etf2ly	140
8.4	Invoking abc2ly	141
8.5	Invoking pmx2ly	141
8.6	Invoking musedata2ly	142
8.7	Invoking mup2ly	142
	Unified index	143

Appendix A	Reference manual details	151
A.1	Chord name chart	151
A.2	MIDI instruments	152
A.3	The Feta font	153
Appendix B	Cheat sheet	157
Appendix C	GNU Free Documentation License	160
C.0.1	ADDENDUM: How to use this License for your documents	165

Preface

Preface to version 2.0

Due to personal circumstances, Han-Wen was able to do a lot more on LilyPond during the past months. A testament to that is the quick release of version 2.0, less than two months after 1.8. We have taken the opportunity to make a few radical changes to the syntax: note attributes, like articulation, dynamics and fingerings are now post-fix exclusively. This makes entering scores easier: you never have to think about the order of the attributes. With version 2.0, we have a new and improved platform for working on notation and typography features for coming versions,

Due to other personal circumstances, Jan was not able to do more than packaging for Cygwin. The good news is that we now have a nearly fool-proof installation for Windows. He will be back for serious hacking in 2.1.

Han-Wen and Jan

Utrecht/Eindhoven, The Netherlands, September 2003.

Preface to version 1.8

If you are familiar with LilyPond version 1.6, then version 1.8 will no offer no big surprises. The only conspicuous change is in the way that formatted text is entered. There is now a new syntax that is more friendly, more versatile and extensible. We hope you like it. In general, development on version 1.8 has been focused on improving the design of various internal mechanisms. This includes chord name formatting and entry code, music expression storage, and integration between LilyPond and Scheme. These changes may not be evident directly, but they make the program more robust and more flexible, which translates into fewer bugs and more adjustment options.

Special thanks for version 1.8 go out to Juergen Reuter for lots of work on the ancient notation engine, and to Amy Zapf for pushing us to rewrite the chord name support.

Han-Wen and Jan,

Utrecht/Eindhoven, The Netherlands, April/May 2003.

Preface to version 1.6

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 e-mail 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 still 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 bugreports. So first and foremost, we would like to thank all users that sent us bugreports, gave suggestions or contributed in any other way to LilyPond.

We would also like to thank the following people: Mats Bengtsson for the incountable number of questions he answered on the mailing list, and Rune Zedeler for his energy in finding and fixing bugs. Nicola Bernardini for inviting us to his workshop on music publishing, which was truly a masterclass, and Heinz Stolba and James Ingram for teaching us there.

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.

1 Introduction

There are a lot of programs that let you print sheet music with a computer, but most of them do not do good job. Most computer printouts have a bland, mechanical look, and are unpleasant to play from. If you agree with us on that, then you will like LilyPond: we have tried to capture the original look of hand-engraved music. We have tuned our algorithms, font-designs, and program settings to make the program produce prints that match the quality of the old editions we love to see and love to play from.

1.1 Notation in LilyPond

Printing sheet music consists of two non-trivial tasks. First, one has to master music notation: the science of knowing which symbols to use for what. Second, one has to master music engraving: the art of placing symbols such that the result looks pleasing.

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 (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? Our solution is to make a strict distinction between notation, *what* symbols to use, and engraving, *where* to put them. Anything related to the second question is considered “engraving” (i.e. typography).

For tackling the first problem, notation, we have broken up the problem into digestible (and programmable) chunks: every type of symbol is handled by a separate program module, a so-called plug-in. Each plug-in are completely modular and independent, so each can be developed and improved separately. When put together, the plug-ins can solve the music notation program in cooperation. People that put graphics to musical ideas are called copyists or engravers, so by analogy, each plug-in is also **engraver**.

In the following example, we see how we start out with a note head 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 heads is seen, a stem object is created, and attached to the note head.

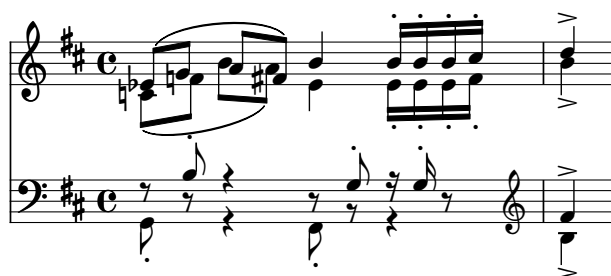
By adding engravers for beams, slurs, accents, accidentals, bar lines, time signature, and key signature, we get a complete piece of notation.



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 head, 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.



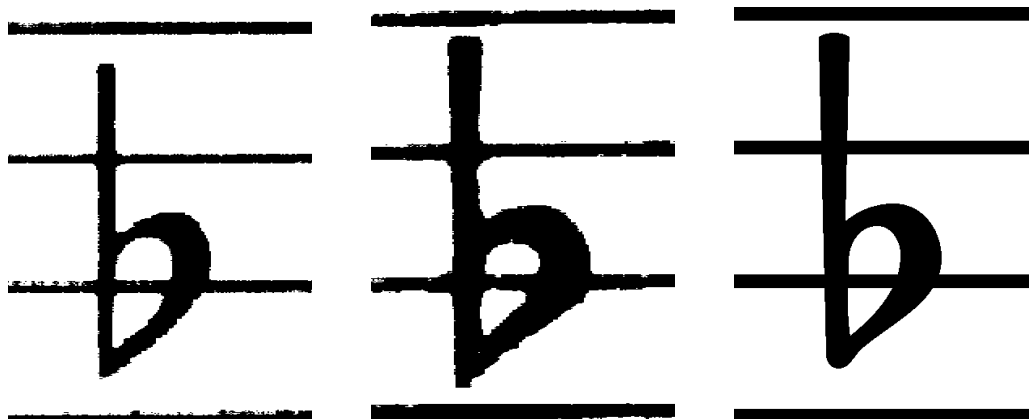
1.2 Engraving in LilyPond

The term music engraving derives from the traditional process of music printing. Only a few decades ago, sheet music was made by cutting and stamping the music into zinc or pewter plates, in mirror image. The plate would be inked, and 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 corrections was cumbersome, so engraving had to be done correctly in one go. Of course, this was a highly specialized skill, much more so than the traditional process of printing books. In the traditional German system of craftsmanship six years of full-time training, more than any other craft, were required before a student could call himself a master of the art. After that many more years of practical experience were needed to become an established music engraver. Even today, with the use of high-speed computers and advanced software, music requires lots of manual fine tuning before it is acceptable for publication.

Sheet music is performance material: everything is done to aid the musician in letting him perform better. Music often is far away from its reader—it might be on a music stand. To make it clearly readable, traditionally printed sheet music always uses bold symbols, on heavy staff lines, and is printed on large sheets of paper. This “strong” look is also present in the horizontal spacing. To minimize the number of page breaks, (hand-engraved) sheet music is spaced very tightly. Yet, by a careful distribution of white space, the feeling of balance is retained, and a clutter of symbols is avoided.

We have used these observations in designing LilyPond. The images below show the flat symbol. On the left, a scan from a Henle edition, which was made by a computer, and in the center is the flat from a hand engraved Bärenreiter edition of the same music. The left scan illustrates typical flaws of computer print: the symbol is much lighter, the staff lines are thinner,

and the glyph has a straight layout with sharp corners. By contrast, the Bärenreiter has a bold and almost voluptuous rounded look. Our flat symbol is designed after, among others, this one. It is tuned it to harmonize 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, adhering with mathematical precision to the duration will lead to a poor result. Shown here is an example of a motive, printed twice. It is printed using exact mathematical spacing, and with some 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-stems clumps of notes.

1.3 Typography and program architecture

Producing good engraving requires skill and knowledge. As the previous examples show, there is a lot of subtlety involved in music engraving, and unfortunately, only a small fraction of these details are documented. Master engravers must learn all these details from experience or from other engravers, which is why it takes so long to become a master. As an engraver gets older and wiser, he will be able to produce better and more complex pieces. A similar situation is present when putting typographical knowledge into a computer program. It is not possible to come up with a definitive solution for a problem at the first try. Instead, we start out with simple solution that might cover 75% of the cases, and gradually refine that solution over the course of months or years, so 90 or 95 % of the cases are handled.

This has an important implication for the design of the program: at any time, almost every piece of formatting code must be considered as temporary. When the need arises, it is to be replaced a solution that will cover even more cases. A “plug-in” architecture is a clean way to accomplish this. This is an architecture where new pieces of code can be inserted in the program dynamically. In such a program, a new solution can be developed along-side the existing code.

For testing, it is plugged in, but for production use, the old solution is used. The new module can be perfected separately until it is better than the existing solution, at which point it replaces the old one.

Until that time, users must have a way to deal with imperfections: these 25%, 10% or 5% of the cases that are not handled automatically. In these cases, a user must be able to override formatting decisions. To accomplish this we store decisions in generic variables, and let the user manipulate those. For example, consider the following fragment of notation:



The position of the forte symbol is slightly awkward, because it is next to the low note, whereas dynamics should be below notes in general. This may be remedied by inserting extra space between the high note and the ‘f’, as shown in this example:



This was achieved with the following input statement:

```
\once \property Voice. DynamicLineSpanner \override #'padding = #4.0
```

It increases the amount of space (`padding`) between the note and the dynamic symbol to 4.0 (which is measured in staff space, so 4.0 equals the height of a staff). The keyword `\once` indicates that this is a tweak: it is only done one time.

Both design aspects, a plug-in architecture, and formatting variables, are built on top of `GUILE`, an interpreter for the programming language Scheme, which is a member of the LISP family. Variables are stored as Scheme objects, and attached to graphical objects such as note heads and stems. The variables are a means to adjust formatting details in individual cases, but they are used in a more general manner.

Consider the case of a publisher that is not satisfied with the in the default layout, and wants heavier stems. Normally, they are 1.3 times the thickness of staff lines, but suppose that their editions require them to be twice the thickness of the staff lines. The same mechanism can be used to adjust a setting globally. By issuing the following command, the entire piece is now formatted with thicker stems:

```
\property Score.Stem \override #'thickness = #2.0
```



In effect, by setting these variables, users can define their own layout styles.

“Plug-ins” are also implemented using Scheme. A formatting “plug-in” takes the form of a function written in Scheme (or a C++ function made available as a Scheme function), and it is also stored in a variable. For example, the placement of the forte symbol in the example above is calculated by the function `Side_position_interface::aligned_side`. If we want to replace this function by a more advanced one, we could issue

```
\property Voice.DynamicLineSpanner \override #'Y-offset-callbacks
= #'(,gee-whiz-gadget)
```

Now, the formatting process will trigger a call to our new `gee-whiz-gadget` function when the position of the `f` symbol has to be determined.

The full scope of this functionality certainly is intimidating, but there is no need to fear: normally, it is not necessary to define style-sheets or rewrite formatting functions. In fact, LilyPond gets a lot of formatting right automatically, so adjusting individual layout situations is not needed often at all.

1.4 Music representation

Our premise is that LilyPond is a system that does music formatting completely automatically. Under this assumption, the output does not have to be touched up. Consequently, an interactive display of the output, where it is possible to reposition notation elements, is superfluous. This implies that the program should be a batch program: the input is entered in a file, which then is *compiled*, i.e. put through the program. The final output is produced as a file ready to view or print. The compiler fills in all the details of the notation, those details should be left out of the input file. In other words, the input should mirror the content as closely as possible. In the case of music notation the content is the music itself, so that is what the input should consist of.

On paper this theory sounds very good. In practice, it opens a can of worms. What really *is* music? Many philosophical treatises must have been written on the subject. Instead of losing ourselves in philosophical arguments over the essence of music, we have reversed the question to yield a more practical approach. Our assumption is that the printed score contains all of the music of piece. We build a program that uses some input format to produce such a score. Over the course of time, the program evolves. While this happens, we can remove more and more elements of the input format: as the program improves, it can fill in irrelevant details of the input by itself. At some (hypothetical) point, the program is finished: there is no possibility to remove any more elements from the syntax. What we have left is by definition exactly the musical meaning of the score.

There are also more practical concerns. Our users have to key in the music into the file directly, so the input format should have a friendly syntax: a quarter note C is entered as `c4`, the code `r8.` signifies a dotted eighth rest.

Notes and rests form the simplest musical expressions in the input syntax. More complex constructs are produced by combining them into compound structures. This is done in much the same way that complex mathematical formulas are built from simple expressions such as numbers and operators.

In effect, the input format is a language, and the rules of that language can be specified succinctly with a so-called context-free grammar. The grammar formally specifies what types of input form valid ‘sentences’. Reading such languages, and splitting them into grammatical structures is a problem with standard solutions. Moreover, rigid definitions make the format easier to understand: a concise formal definition permits a simple informal description.

The user-interface of LilyPond is its syntax. That part is what users see most. As a result, some users think that music representation is a very important or interesting problem. In reality, less than 10% of the source code of the program handles reading and representing the input, and they form the easy bits of the program. In our opinion, producing music notation, and formatting it prettily are much more interesting and important than music representation: solving these problems takes up most of the bulk of the code, and they are the most difficult things to get right.

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:

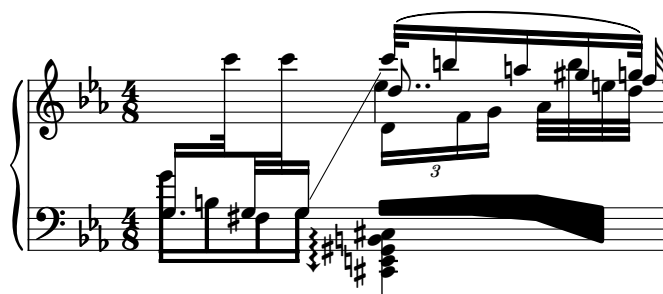


By adding chord names and lyrics we obtain a lead sheet:

C C F C
twin kle twin kle lit tle star



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 plain 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 32* discusses topics grouped by notation construct. Once you master the basics, this is the place to look up details.
- *Chapter 4 [Literature list], page 114* contains a set of useful reference books, for those who wish to know more on notation and engraving.
- *Chapter 5 [Technical manual], page 116* discusses the general design of the program, and how to extend its functionality.
- *Chapter 6 [Invoking LilyPond], page 126* explains how to run LilyPond and its helper programs.
- *Chapter 7 [lilypond-book manual], page 133* explains the details behind creating documents with in-line music examples (like this manual).
- *Chapter 8 [Converting from other formats], page 139* 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.

Once you are an experienced user, you can use the manual as reference: there is an extensive index¹, 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), then 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 crosslinked 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 headstart, we have 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.

These small files show various tips and tricks, and are 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 like 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/>.

¹ If you are looking for something, and you cannot find it by using the index, that is considered a bug. In that case, please file a bug report.

2 Tutorial

Using LilyPond comes down to encoding music in an input file. After entering the music, the program is run on the file producing output which can be viewed or printed. In this tutorial, we will show step by step how to enter such files, and illustrate the process with fragments of input and the corresponding output. At the end of every section, a paragraph will list where to find further information on the topics discussed.

Many people learn programs by trying and fiddling around with the program. This is also possible with LilyPond. If you click on a picture in the HTML version of this manual, you will see the exact LilyPond input that was used to generate that image. 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 B [Cheat sheet], page 157, which is a table listing all commands for quick reference.

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. You should then be able to create and print your first sheets of music.

2.1 First steps

We start off by showing how very simple music is entered in LilyPond: you get a note simply by typing its note name, from ‘a’ through ‘g’. So if you enter

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

then the result looks like this:



The length of a note is specified by adding a number, ‘1’ for a whole note, ‘2’ for a half note, and so on:

```
a1 a2 a4 a16 a32
```



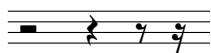
If you do not specify a duration, the previous one is used:

```
a4 a a2 a
```



Rests are entered just like notes, but with the name “r”:

```
r2 r4 r8 r16
```



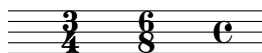
Add a dot ‘.’ after the duration to get a dotted note:

```
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
```



Notes and commands like `\clef` and `\time`, are enclosed in `\notes {...}`. This indicates that music (as opposed to lyrics) follows:

```
\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. The final step is to combine the music with a printing command.

The printing command is the so-called `\paper` block:

```
\paper { }
```

The `\paper` block is used to customize printing specifics. The customization commands go between `{` and `}`, but for now, we accept the defaults. The music and the `\paper` block are combined by enclosing them in `\score { ... }`, so the following is a complete and valid input file:

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



In the rest of the tutorial we will often leave out `\score` and `\paper` for clarity. However, both 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 32 and Section 3.1.7 [Durations], page 34.

Clefs see Section 3.3.3 [Clef], page 41

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 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 One of the following commands should put the PDF on your screen:

```
gv test.pdf
ghostview test.pdf
ggv test.pdf
kghostview test.pdf
xpdf test.pdf
gpdf test.pdf
acroread test.pdf
gsview32 test.pdf
```

If the music on your screen looks good, you can print it by clicking File/Print inside your viewing program.

On Windows, the same procedure should work, the terminal is started by clicking on the LilyPond or Cygwin icon. Any text editor (such as NotePad, Emacs or Vim) may be used to edit the LilyPond file.

To view the PDF file, try the following:

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

² For T_EX 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.

- If your system has a PDF viewer installed, open ‘C:\Cygwin\home\your-name’ in the explorer and double-click ‘test.pdf’.
- If you prefer the keyboard, you can try to enter one of the commands from the list shown before in the terminal. If none work, go to <http://www.cs.wisc.edu/~ghost/> to install the proper software.

The commands for formatting and printing music on all platforms are detailed in Chapter 6 [Invoking LilyPond], page 126.

2.3 More about pitches

A sharp (#) pitch is made by adding ‘is’ to the name, a flat (b) pitch by adding ‘es’. As you might expect, a double sharp or double flat is made by adding ‘isis’ or ‘eses’:³

```
cis1 ees fisis aeses
```



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

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



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

For example, 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 gets an accidental:



```
\key as \major
```

³ This syntax derived from note naming conventions in Nordic and Germanic languages, like German and Dutch.

d

Adding all alterations explicitly might require some 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 48 for some examples how accidentals can be printed according to different rules.

A tie is created by adding a tilde “~” to the first note being tied:

g4~ g a2~ a4



This example shows the key signature, accidentals and ties in action:

```
\score {
  \notes {
    \time 4/4
    \key g \minor
    \clef violin
    r4 r8 a8 gis4 b
    g8 d4.~ d e'8
    fis4 fis8 fis8 eis4 a8 gis~
    gis2 r2
  }
  \paper { }
```



Key signature

see Section 3.3.2 [Key signature], page 40

Beams

see Section 3.5 [Beaming], page 46

2.4 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
}
```



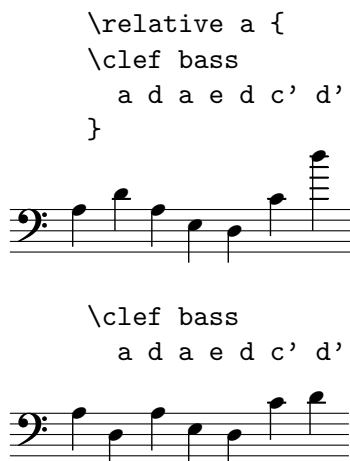
Larger intervals are made by adding octavation quotes.

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



Quotes or commas do not determine the absolute height of a note; the height of a note is relative to the previous one. For example: `c f`, goes down; `f, f` are both the same; `c' c` are the same; and `c g'` goes up:

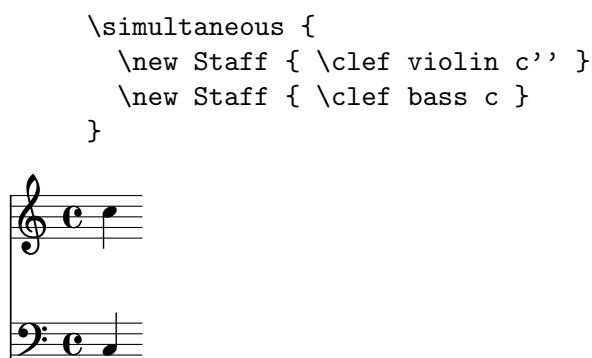
Here is an example of the difference between relative mode and “normal” (non-relative) mode:



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

2.5 Combining music into compound expressions

To print more than one staff, each piece of music that makes up a staff is marked by adding `\context Staff` before it. These `Staff`'s are then grouped inside `\simultaneous {` and `}`, as is demonstrated here:



In this example, `\simultaneous` indicates that both music fragments happen at the same time, and must be printed stacked vertically. The notation `<< .. >>` can also be used as a shorthand for `\simultaneous { .. }`.

The command `\new` introduces a “notation context”. To understand this concept, imagine that you are performing a piece of music. When you are playing, you combine the symbols printed at a certain point with contextual information. For example, without knowing the current clef, and the accidentals in the last measure, it would be impossible to determine the pitch of a note. In other words, this information forms context that helps you decipher a score. LilyPond produces notation from music, so in effect, it does the inverse of reading scores. Therefore, it

also needs to keep track of contextual information. This information is maintained in “notation contexts.” There are several types of contexts, e.g. **Staff**, **Voice** and **Score**, but also **Lyrics** and **ChordNames**. Prepending `\new` to a chunk of music indicates what kind of context to use for interpreting it, and ensures that the argument is interpreted with a fresh instance of the context indicated.

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.
    }
  >>
  \paper {}
}
```



The example shows how small chunks of music, for example, the notes `c2`, `e4`, etc. of the second staff, are combined to form a larger chunk by enclosing it in braces. Again, a larger chunk is formed by prefix `\new Staff` to it, and that chunk is combined with `<< >>`. This mechanism is similar with 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. In LilyPond input, a similar mechanism is used. Here, the simplest expressions are notes and rests. By enclosing expressions in `<< >>` and `{ }`, more complex music is formed. The `\new` command also forms new expressions; prepending it to a music expression yields a new expression.

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 amount of closing braces at the end of an expression. For example,

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

For more information on context see the Technical manual description in Section 5.1 [Interpretation context], page 116.

2.6 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 to the note:

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



Crescendi and decrescendi are started with the commands \< and \>. The command \! finishes a crescendo on the note it is attached to:

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



A slur is drawn across many notes, and indicates bound articulation (legato). 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:

If you need two slurs at the same time (one for articulation, one for phrasing), you can also make a phrasing slur with \ (and \).

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



For more information on

- fingering see Section 3.7.8 [Fingering instructions], page 55
- 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 59
- fingering

2.7 Combining notes into chords

Chords can be made by surrounding pitches with < and >:

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



You can combine beams and ties with chords. Beam and tie markings must be placed outside the chord markers:

```
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\!)
```



2.7.1 Basic rhythmical commands

A pickup (or upstep) 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 note, or a set of notes with a keyword. In this case, the keywords are `\appoggiatura` and `\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,

upsteps see Section 3.3.6 [Partial measures], page 43.

2.7.2 Commenting input files

Comments are pieces of the input that are ignored. There are two types of comments. A line comments is introduced by `%`: after that, the rest of that line is ignored. Block comments span larger sections of input. Anything that is enclosed in `%{` and `%}` is ignored too. 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
%}
```

2.8 Printing lyrics

Lyrics are entered by separating each syllable with a space, and surrounding them with `\lyrics { ... }`, for example,

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

Like notes, lyrics are also a form of music, but they must not be printed on a staff, which is the default way to print music. To print them as lyrics, they must be marked with `\new Lyrics`:

```
\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 `\addlyrics` keyword:

```
\addlyrics
  \notes { ... }
  \new Lyrics ...
```

The final result is

```
\score {
  \notes {
    \addlyrics
    \relative c' {
      \partial 8
      c8
      \times 2/3 { f g g } \times 2/3 { g4( a2) }
    }
    \new Lyrics \lyrics { I want to break free }
  }
  \paper{ }
}
```

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.

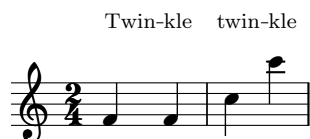
```
\lyrics { I want to break free __ }
```

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 70.

2.9 A lead sheet

In popular music, it is common to denote accompaniment as chord-names. Using them in LilyPond has two parts, just like lyrics: entering the chords (with `\chords`), and printing them (with `\new ChordNames`).

Chord names are entered by starting chords mode (with `\chords`). In chords mode, you can enter chords with a letter (indicating the root of the chord), and a durations following that:

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



The result of `\chords` is a list of chords, and is equivalent to entering chords with `<...>`.

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 }
```



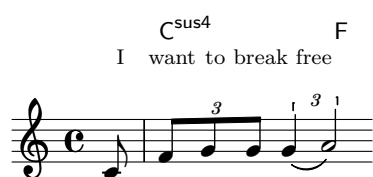
Printing chords is done by adding `\context ChordNames` before the chords thus entered:

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

```
C Fm GΔ G#7
```

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

```
\score {
  <<
    \context ChordNames \chords { chords }
    \addlyrics
    \notes the melody
    \context Lyrics \lyrics { the text }
  >>
  \paper { }
}
```



A complete list of modifiers, and other options for layout are in the reference manual section Section 3.1.4 [Chords], page 33.

2.10 Listening to output

MIDI (Musical Instrument Digital Interface) is a standard for connecting and recording digital instruments. A MIDI file is like a tape recording of a MIDI instrument. The `\midi` block makes the music go to 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.

`\midi` can be used in similarly to `\paper { }`, for example,

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

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 is in the Section 3.19 [Sound], page 112 section in the notation manual.

2.11 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 by the `lilypond` wrapper script, then the title and composer specified 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.”⁴

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), then the header can be put into 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 6.1 [Invoking lilypond], page 126.

⁴ Nicely printed parts are good PR for us, so do us a favor, and leave the tagline if you can.

2.12 Single staff polyphony

When different melodic lines are combined on a single staff, these 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:

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



Again, these expressions can be nested arbitrarily:



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

2.13 Piano staves

Piano music is always typeset in two staves connected by a brace. Printing such a staff is done similar to the polyphonic example in Section 2.5 [Combining music into compound expressions], page 16:

```
<< \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 66.

2.14 Setting variables

When the music is converted from notes to print, it is interpreted from left-to-right order, similar to what happens when we read music. During this step, context-sensitive information, such as the accidentals to print, and where barlines must be placed, are stored in variables. These variables are called *context properties*. The properties can also be manipulated from input files. Consider this input:

```
\property Staff.autoBeaming = ##f
```

It sets the property named `autoBeaming` in the current staff at this point in the music to `##f`, which means ‘false’. This property controls whether beams are printed automatically:

```
c8 c c c
\property Staff.autoBeaming = ##f
c8 c c c
```



LilyPond includes a built-in programming language, namely, a dialect of Scheme. The argument to `\property`, `##f`, is an expression in that language. The first hash-mark signals that a piece of Scheme code follows. The second hash character is part of the boolean value true (`#t`). Values of other types may be entered as follows:

- a string, enclosed in double quotes, for example,

```
\property Staff.instrument = #"French Horn"
```

- a boolean: either `#t` or `#f`, for true and false respectively, e.g.

```
\property Voice.autoBeaming = ##f
\property Score.skipBars = ##t
```

- a number, such as

```
\property Score.currentBarNumber = #20
```

- a symbol, which is introduced by a quote character, as in

```
\property Staff.crescendoSpanner = #'dashed-line
```

- a pair, which is also introduced by a quote character, like in the following statements, which set properties to the pairs `(-7.5, 6)` and `(3, 4)` respectively:

```
\property Staff.minimumVerticalExtent = #'(-7.5 . 6)
\property Staff.timeSignatureFraction = #'(3 . 4)
```

- a list, which is also introduced by a quote character. In the following example, the `breakAlignOrder` property is set to a list of symbols:

```
\property Score.breakAlignOrder =
  #'(left-edge time-signature key-signatures)
```

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

2.15 Fine tuning layout

Sometimes it is necessary to change music layout by hand. When music is formatted, layout objects are created for each symbol. For example, every clef and every note head is represented by a layout object. These layout objects also carry variables, which we call *layout properties*. By changing these variables from their values, we can alter the look of a formatted score:

```
c4
\property Voice.Stem \override #'thickness = #3.0
c4 c4 c4
```



In the example shown here, the layout property **thickness** (a symbol) is set to 3 in the **Stem** layout objects of the current **Voice**. As a result, the notes following `\property` have thicker stems.

In most cases of manual overrides, only a single object must be changed. This can be achieved by prefixing `\once` to the `\property` statement, i.e.

```
\once \property Voice.Stem \set #'thickness = #3.0
```



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 32, 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 couple 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 unit 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 example, the second fingering is moved a little to the left, and 1.8 staff space downwards:

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

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

The following example demonstrates how to connect different voices using ties. Normally ties only happen between notes of the same voice. By introducing a tie in a different voice, and blanking a stem in that voice, the tie appears to cross voices:

```
c4 << {
  \once \property Voice.Stem \set #'transparent = ##t
  b8~ b8
} \ {
  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 3.17.2 [Constructing a tweak], page 103:

```
c2\fermata
\property Voice.Script \set #'padding = #3
b2\fermata
```



More specific overrides are also possible. The notation manual discusses in depth how to figure out these statements for yourself, in Section 3.17 [Tuning output], page 101.

2.16 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 and/or orchestral 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 {
  dis'8 e'8
}
\score { \notes {
  \seufzer \seufzer
} }
```



The name of an identifier should have alphabetic characters only, and 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 {
```

```

\begin{document}
\begin{music}
\set \aFivePaper
\set \linewidth = \width
}
}

```

More information on the possible uses of identifiers is in the technical manual, in Section 5.2.3 [Scheme datatypes], page 121.

2.17 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 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.lyinc"
\header {
  instrument = "Horn in F"
}
\score {
  \notes \transpose f c' \hornNotes
}

```

The `\include` command substitutes the contents of the file at this position in the file, so that `hornNotes` is defined afterwards. The code `\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, and 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, the next rest takes 3 measures in 2/4 time:

```
R2*3
```

When printing the part, the following `skipBars` property must be set to false, to prevent the rest from being expanded in three one bar rests:

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

Prepending the rest and the property setting 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’:

```

\include "bassoon-music.lyinc"
\include "horn-music.lyinc"

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

```

This would lead to the simple score depicted below:



More in-depth information on preparing parts and scores is in the notation manual, in Section 3.14 [Orchestral music], page 79.

2.18 Integrating text and music

Sometimes you might want to use music examples in a text that you are writing (for example, a musicological treatise, a songbook, or (like us) the LilyPond manual). You can make such texts by hand, simply by importing a PostScript figure into your word processor. However, there is an automated procedure to reduce the amount of work.

If you use HTML, LaTeX, or Texinfo, you can mix text and LilyPond code. A script called `lilypond-book` will extract the music fragments, run LilyPond on them, and put back the resulting notation. This program is fully described in Chapter 7 [lilypond-book manual], page 133. Here we show a small example; since the example contains also explanatory text, we will not comment it further:

```

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

```

In a `lilypond-book` document, you can 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 you have no `\verb+\score+` block in the fragment, `\texttt{lilypond-book}` will supply one:

```

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

```

In the example you see here, two things happened: a `\verb+\score+` block was added, and the line width was set to natural length. You can specify options by putting them in brackets:

```
\begin[26pt,verbatim]{lilypond}
  c'4 f16
\end{lilypond}
```

If you want to include large examples into the text, it is more convenient to put it in a separate file:

```
\lilypondfile{screech-boink.ly}

\end{document}
```

Under Unix, you can view the results as follows:

```
$ cd input/tutorial
$ mkdir -p out/
$ lilypond-book --outdir=out/ lilbook.tex
lilypond-book (GNU LilyPond) 1.7.23
Reading 'input/tutorial/lilbook.tex'
Reading 'input/screech-boink6.ly'
lots of stuff deleted
Writing 'out/lilbook.latex'
$ cd out
$ latex lilbook.latex
lots of stuff deleted
$ xdvi lilbook
```

Running `lilypond-book` and running `latex` creates a lot of temporary files, and you would not want those to clutter up your working directory. The `outdir` option to `lilypond-book` creates the temporary files in a separate subdirectory ‘out’.

The result looks more or less like this:

In a `lilypond-book` document, you can freely mix music and text. For example:

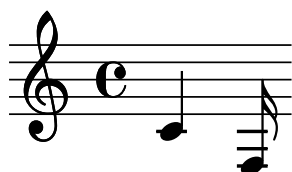


If you have no `\score` block in the fragment, `lilypond-book` will supply one:

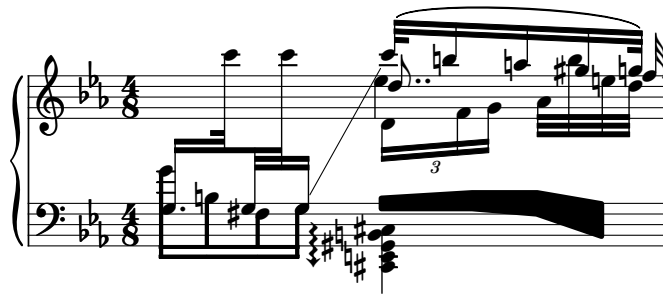


In the example you see here, two things happened: a `\score` block was added, and the line width was set to natural length. You can specify options by putting them in brackets:

```
c'4 f16
```



If you want to include large examples into the text, it is more convenient to put it in a separate file:



3.1 Note entry

The basic elements of any piece of music are the notes. This section is about basic notation elements notes, rests and related constructs, such as stems, tuplets and ties.

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

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



The most common syntax for pitch entry is used in `\chords` and `\notes` mode. In Note and Chord mode, 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''
```



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
cih
cis
cisih
cisis



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.

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

`bla`

`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 (an accidental within parentheses) can be obtained by adding the question mark ‘?’ after the pitch:

`cis’ cis’ cis’! cis’?`



The automatic production of accidentals can be tuned in many ways. For more information, refer to Section 3.6 [Accidentals], page 48.

3.1.4 Chords

A chord is formed by 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`:

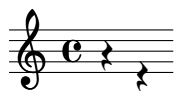
```
r1 r2 r4 r8
```



Whole bar rests, centered in middle of the bar, must be done with multi measure rests. They are discussed in Section 3.14.6 [Multi measure rests], page 81.

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

`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.

See also

`SkipEvent`.

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:

```
\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

Dots, and DotColumn.

Bugs

In dense chords, dots can overlap.

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

`TieEvent`, `NewTieEvent`, `Tie`, and Section 3.2.5 [Automatic note splitting], page 39.

If you want less ties created for a chord, see `'input/test/tie-sparse.ly'`.

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 triplet 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:

`\property Voice.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

`TupletBracket`, and `TimeScaledMusic`.

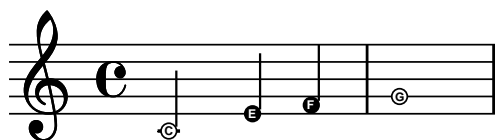
Bugs

Nested tuplets are not formatted automatically. In this case, outer tuplet brackets should be moved manually.

3.1.11 Easy Notation note heads

The ‘easy play’ note head includes a note name inside the head. It is used in music aimed at beginners:

```
\score {
  \notes { c'2 e'4 f' | g'1 }
  \paper { \translator { \EasyNotation } }
}
```



The `EasyNotation` variable overrides a `Score` context. You probably will want to print it with magnification or a large font size to make it more readable. To print with magnification, you must create a DVI file (with ‘`lilypond`’) and then enlarge it with something like ‘`dvips -x 2000 file.dvi`’. See the `dvips` documentation for more details. To print with a larger font, see Section 3.18.3 [Font Size], page 111.

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

3.2 Easier music entry

When entering music it is easy to introduce errors. This section deals with tricks and features of the input language that were added solely to help entering music, and find and correct mistakes.

It is also possible to use external programs, for example GUI interfaces, or MIDI transcription programs, to enter or edit music. Refer to the website for more information. Finally, there are tools make debugging easier, by linking the input file and the output shown on screen. See Section 6.7 [Point and click], page 131 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 pitch as a note name, you have to be in note mode, so there must be a surrounding `\notes` keyword (which is not shown here).

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. The syntax is

```
\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 central 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. Depending on the value of `barCheckSynchronize`, the beginning of the measure will be relocated.

In the next example, the second bar check will signal an error:

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

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
\property Score.skipTypesetting = ##t
  e f g a g c, f e d
\property 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`:

```
\paper { \translator {
  \ThreadContext
  \remove "Note_heads_engraver"
  \consists "Completion_heads_engraver"
} }
```

which will make long notes tied in the following example:

```
\time 2/4
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.

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.

This object is created whenever a `Staff` context is created. The appearance of the staff symbol cannot be changed by using `\override` or `\set`. At the moment that `\property Staff` is interpreted, a `Staff` context is made, and the `StaffSymbol` is created before any `\override` is effective. Properties can be changed in a `\translator` definition, or by using `\applyoutput`.

Bugs

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

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.

Syntax

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. The tutorial explains why this is so in Section 2.3 [More about pitches], page 13.

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

KeyChangeEvent, and KeySignature.

3.3.3 Clef

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

Syntax

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

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

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 }
```



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

`OttavaSpanner`.

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.

Syntax

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 {
  raggedright = ##t
  \translator { \StaffContext
    \consists "Measure_grouping_engraver"
  }
}
```



See also

TimeSignature, and Timing_engraver.

Bugs

Automatic beaming does not use 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

```
\property 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.

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 barlines.

Syntax

Special types of barlines 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 "|."

```



For allowing linebreaks, there is a special com-

mand,

```
\bar "empty"
```

This will insert an invisible barline, and allow linebreaks at this point.

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

```

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

```



The command `\bar bartype` is a short cut for doing `\property Score.whichBar = bartype`. Whenever `whichBar` is set to a string, a bar line of that type is created. At the start of a measure it is set to `defaultBarType`. The contents of `repeatCommands` are used to override default measure bars.

Property `whichBar` can also be set directly, using `\property` or `\bar`. These settings take precedence over the automatic `whichBar` settings.

You are encouraged to use `\repeat` for repetitions. See Section 3.8 [Repeats], page 60.

See also

Section 3.8 [Repeats], page 60.

The bar line objects that are created at **Staff** level are called **BarLine**, the bar lines that span staves are **SpanBars**.

The barlines at the start of each system are **SystemStartBar**, **SystemStartBrace**, and **SystemStartBracket**. They are spanner objects and typically must be tuned from a `\translator` block.

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:

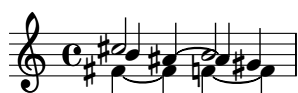
```
\context Staff \relative c'' {
  c4 << { f d e } \ \ { b c2 } >>
  c4 << g' \ \ b, \ \ f' \ \ d >>
}
```



The separator causes `Voice` contexts² 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 } >>
```



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:

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



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

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



LilyPond also vertically shifts rests that are opposite of a stem:

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

```
\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

The objects responsible for resolving collisions are `NoteCollision` and `RestCollision`. 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 8th or a shorter note, and a downstem half note, the 8th note gets the wrong offset.

3.5 Beaming

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

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



When these automatic decisions are not good enough, beaming can be entered explicitly. It is also possible to define beaming patterns that differ from the defaults.

See also

`Beam`.

3.5.1 Manual beams

In some cases it may be necessary to override the automatic beaming algorithm. For example, the auto beamer 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. When this mechanism fouls up, the properties `Voice.stemLeftBeamCount` and `Voice.stemRightBeamCount` can be used to control the beam subdivision on a stem. 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 \property Voice.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. This accomplishes the same effect as twiddling with `stemLeftBeamCount` and `stemRightBeamCount`, but it takes less typing:

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



Kneaded 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 kneaded 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 `Voice.autoBeamSettings`. The defaults for `autoBeamSettings` are defined in `'scm/auto-beam.scm'`.

The value of `autoBeamSettings` is changed using `\override` and restored with `\revert`:

```
\property Voice.autoBeamSettings \override #'(BE P Q N M) = dur
\property Voice.autoBeamSettings \revert #'(BE P Q N M)
```

Here, *BE* is the symbol `begin` or `end`. 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).

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

```
\property Voice.autoBeamSettings \override
  #'(end * * * *) = #(ly:make-moment 1 4)
```

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:

```
\property Voice.autoBeamSettings \override
  #'(end * * * *) = #(ly:make-moment 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

```
\property Voice.autoBeamSettings \override
  #'(begin * * 6 8) = ...
```

If a rule should be 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 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.

For melodies that have lyrics, you may want to switch off automatic beaming. This is done by setting `Voice.autoBeaming` to `#f`.

Predefined commands

`\autoBeamOff`, `\autoBeamOn`.

Bugs

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 eighth beams and sixteenth beams, a beam that contains both eighth and sixteenth notes will use the rules for the sixteenth beam.

In the example below, the autobeamer makes eighth beams and sixteenth end at 3 eighths; the third beam can only be corrected by specifying manual beaming.



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

3.6 Accidentals

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

3.6.1 Using the predefined accidental variables

The constructs for describing the accidental typesetting rules are quite hairy, so non-experts should stick to the variables defined in `'ly/property-init.ly'`.

The variables set properties in the “Current” context (see Section 5.1.3 [Context properties], page 117). This means that the variables should normally be added right after the creation of the context in which the accidental typesetting described by the variable is to take effect. For example, if you want to use piano-accidentals in a piano staff then issue `\pianoAccidentals` first thing after the creation of the piano staff:

```

\score {
  \notes \relative c'' <<
    \new Staff { cis4 d e2 }
    \context GrandStaff <<
      \pianoAccidentals
      \new Staff { cis4 d e2 }
      \new Staff { es2 c }
    >>
    \new Staff { es2 c }
  >>
}

```



The variables are:

`\defaultAccidentals`

This is the default typesetting behaviour. 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.

`\voiceAccidentals`

The normal behaviour 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 `\defaultAccidentals`.

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

```

\context Staff <<
  \voiceAccidentals
  <<
    { es g } \\\
    { c, e }
  >> >>

```



Hence you should only use `\voiceAccidentals` 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 `\modernVoiceAccidentals` or `\modernVoiceCautionaries` instead.

`\modernAccidentals`

This rule corresponds to the common practice in the 20th century. The rule is more complex than `\defaultAccidentals`. You get all the same accidentals, but temporary accidentals also get cancelled in other octaves. Furthermore, in the same octave, they also get cancelled in the following measure:

```
\modernAccidentals
cis' c'' cis'2 | c'' c'
```



`\modernCautionaries`

This rule is similar to `\modernAccidentals`, but the “extra” accidentals (the ones not typeset by `\defaultAccidentals`) are typeset as cautionary accidentals. They are printed in reduced size or with parentheses:

```
\modernCautionaries
cis' c'' cis'2 | c'' c'
```



`\modernVoiceAccidentals`

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* cancelled across voices in the same `Staff`.

`\modernVoiceCautionaries`

is the same as `\modernVoiceAccidentals`, but with the extra accidentals (the ones not typeset by `\voiceAccidentals`) typeset as cautionaries. Even though all accidentals typeset by `\defaultAccidentals` *are* typeset by this variable then some of them are typeset as cautionaries.

`\pianoAccidentals`

20th century practice for piano notation. Very similar to `\modernAccidentals` but accidentals also get cancelled across the staves in the same `GrandStaff` or `PianoStaff`.

`\pianoCautionaries`

As `\pianoAccidentals` but with the extra accidentals typeset as cautionaries.

`\noResetKey`

Same as `\defaultAccidentals` but with accidentals lasting “forever” and not only until the next measure:

```
\noResetKey
c1 cis cis c
```



`\forgetAccidentals`

This is sort of the opposite of `\noResetKey`: 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:

```
\forgetAccidentals
\key d\major c4 c cis cis d d dis dis
```



3.6.2 Customized accidental rules

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.

lazyness: Over how many barlines the accidental lasts. If *lazyness* is -1 then the accidental is forget immediately, and if *lazyness* is #t then the accidental lasts forever.

Predefined commands

`\defaultAccidentals`, `\voiceAccidentals`, `\modernAccidentals`, `\modernCautionaries`,
`\modernVoiceAccidentals`, `\modernVoiceCautionaries`, `\pianoAccidentals`,
`\pianoCautionaries`, `\noResetKey`, `\forgetAccidentals`.

See also

`Accidental_engraver`, `Accidental`, and `AccidentalPlacement`.

Bugs

Currently the 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 there are simultaneous notes whose accidentals depend on each other. The problem only occurs when using non-default accidentals. In the default scheme, accidentals only depend on other accidentals with the same pitch on the same staff, so no conflicts possible.

This example shows two examples of the same music giving different accidentals depending on the order in which the notes occur in the input file:

```
\property Staff.autoAccidentals = #'( Staff (any-octave . 0) )
cis'4 <c'' c'> r2 | cis'4 <c' c''> r2
| <cis' c''> r | <c'' cis'> r |
```



This problem can be solved by manually inserting ! and ? for the problematic notes.

3.7 Expressive marks

3.7.1 Slurs

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

Syntax

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` in `Voice` context. Its value is a pair of symbols, specifying the attachment type of the left and right end points:

```
\slurUp
\property Voice.Stem \set #'length = #5.5
g'8(g g4)
\property Voice.Slur \set #'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..)
\property Voice.Slur \set #'attachment = #'(stem . stem)
d,32( d'4 d8..)
```



Predefined commands

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

See also

internals document, `Slur`, and `SlurEvent`.

Bugs

Producing nice slurs is a difficult problem, and LilyPond currently 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

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 tweaked by overriding the `text` property of the `BreathingSign` layout object with any markup text. For example,

```
c'4
\property Voice.BreathingSign \override #'text
= #(make-musicglyph-markup "scripts-rvarcomma")
\breathe
d4
```



See also

`BreathingSign`, `BreathingSignEvent`, and `'input/regression/breathing-sign.ly'`.

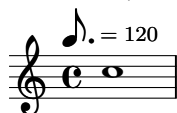
3.7.4 Metronome marks

Metronome settings can be entered as follows:

```
\tempo duration = perminute
```

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

`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
  \property Voice.TextSpanner \set #'direction = #-1
  \property Voice.TextSpanner \set #'edge-text = #'("rall " . "")
  c2\startTextSpan b c\stopTextSpan a }
```



See also

TextSpanEvent, TextSpanner, and ‘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 { \translator {
    \StaffContext \consists "Horizontal_bracket_engraver"
  }}}}
```



See also

HorizontalBracket, NoteGroupingEvent, and ‘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.

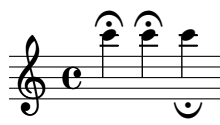
The script is automatically placed, but if you need to force directions, you can use `_` to force them down, or `^` to put them up:

```
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 `_`, eg.


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




`accent marcato staccatissimo staccato tenuto portato upbow downbow`




`flageolet thumb lheel rheel ltoe rtie open stopped turn reverseturn`



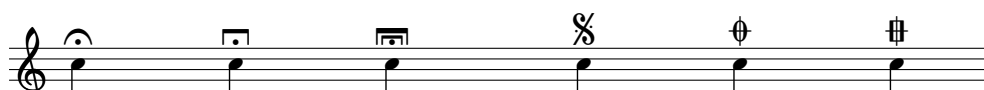
`trill prall mordent prallprall prallmordent upprall downprall upmordent`



`downmordent pralldown prallup lineprall signumcongruentiae shortfermata`



`fermata longfermata verylongfermata segno coda varcoda`



Predefined commands

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

See also

`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 { \fontsize #-3 \number "2-3" }
```



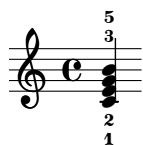
You can use the thumb-script to indicate that a note should be played with your thumb (used in cello music):

```
<a' a''-3>8(_\thumb <b' b''-3>)\thumb
<c'' c'''-3>(_\thumb <d'' d'''-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
```



Setting `fingeringOrientations` will put fingerings next to note heads:

```
\property Voice.fingeringOrientations = #'(left down)
<c-1 es-2 g-4 bes-5 > 4
\property Voice.fingeringOrientations = #'(up right down)
<c-1 es-2 g-4 bes-5 > 4
```



See also

`FingerEvent`, and `Fingering`.

3.7.9 Text scripts

It is possible to place arbitrary strings of text or markup text (see Section 3.17.5 [Text markup], page 106) 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.

See also

`TextScriptEvent`, `TextScript`, and Section 3.17.5 [Text markup], page 106.

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:



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

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

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



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

```
\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 {
    \property Voice.Stem \override #'direction = #-1
    f16->
    \property Voice.Stem \revert #'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 grace section, 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

`GraceMusic`.

Bugs

Grace notes cannot be used in the smallest size (`'paper11.ly'`).

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, barlines, 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 } >>
```



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.

Overriding settings cannot be done in separate styles for appoggiatura and acciaccatura.

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.

Syntax

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

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



See also

Glissando, and GlissandoEvent.

Bugs

Adding additional texts (such as *gliss.*) is not supported.

3.7.12 Dynamics

Absolute dynamic marks are specified using an variable 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''\decr e''\rced
<< 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:

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

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

```
c4 \cresc c4 c c c \endcresc c4
```



You can also supply your own texts:

```

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

```



Predefined commands

\dynamicUp, \dynamicDown, \dynamicBoth.

See also

CrescendoEvent, DecrescendoEvent, and AbsoluteDynamicEvent.

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

If you want to adjust padding or vertical direction of the dynamics, you must set properties for 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 (voltas) 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

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, then 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

If you do a nested repeat like

```
\repeat ...
\repeat ...
\alternative
```

then it is ambiguous 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 unfold 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,

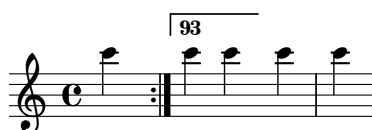
the symbol `end-repeat`,
which prints a :| bar line,

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 3.17.5 [Text markup], page 106. 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
\property Score.repeatCommands = #'((volta "93") end-repeat)
c4 c4
\property Score.repeatCommands = #'((volta #f))
c4 c4
```



See also

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 }
    \repeat "tremolo" 4 c16
  }
}
```



See also

Tremolo beams are `Beam` objects. Single stem tremolos are `StemTremolos`. The music expression is `TremoloEvent`.

Bugs

The single stem tremolo must be entered without `{` and `}`.

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, then then the last value (stored in `Voice.tremoloFlags`) is used:

```
c'2:8 c':32 | c': c': |
```



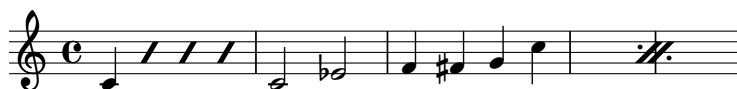
Bugs

Tremolos in this style do not carry over into the MIDI output.

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

`RepeatSlash`, `PercentRepeat`, `PercentRepeatedMusic`, and `DoublePercentRepeat`.

3.9 Rhythmic music

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 |
}
```



3.9.1 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.

Syntax

Percussion staves are typeset with help of a set of Scheme functions. The system is based on the general MIDI drum-pitches. Include `'drumpitch-init.ly'` to use drum pitches. This file defines the pitches from the Scheme variable `drum-pitch-names`, the definition of which can be read in `'scm/drums.scm'`. Each piece of percussion has a full name and an abbreviated name, and either the full name or the abbreviation may be used in input files.

To typeset the music on a staff apply the function `drums->paper` to the percussion music. This function takes a list of percussion instrument names, notehead scripts and staff positions (that is: pitches relative to the C-clef) and transforms the input music by moving the pitch, changing the notehead and (optionally) adding a script:

```
\include "drumpitch-init.ly"
up = \notes { crashcymbal4 hihat8 halfopenhihat hh hh hh openhihat }
down = \notes { bassdrum4 snare8 bd r bd sn4 }
\score {
  \apply #(drums->paper 'drums) \context Staff <<
    \clef percussion
    \new Voice { \voiceOne \up }
```

```

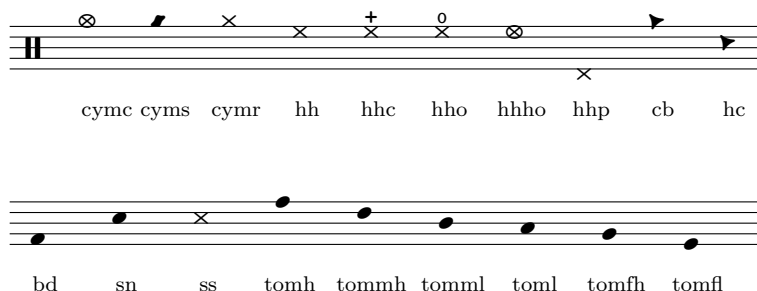
\new Voice { \voiceTwo \down }
>>
}

```



In the above example the music was transformed using the list `'drums`. The following lists are defined in `'scm/drums.scm`:

`'drums` to typeset a typical drum kit on a five-line staff:

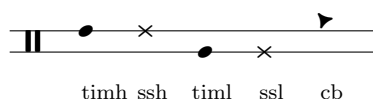


The drum scheme supports six different toms. When there are fewer toms, simply select the toms that produce the desired result, i.e. to get toms on the three middle lines you use `tommh`, `tomml` and `tomfh`.

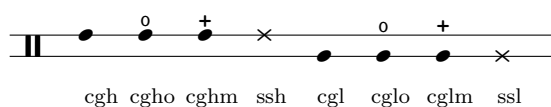
Because general MIDI does not contain rimshots the sidestick is used for this purpose instead.

`'timbales`

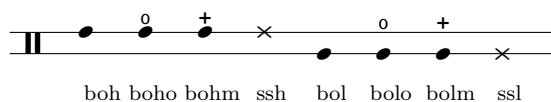
to typeset timbales on a two line staff:



`'congas` to typeset congas on a two line staff:



`'bongos` to typeset bongos on a two line staff:



`'percussion`

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:

```

#(set-drum-kit 'mydrums '(
  (bassdrum default #f ,(ly:make-pitch -1 2 0))
  (snare default #f ,(ly:make-pitch 0 1 0))

```

```
#(set-drum-kit 'mydrums (append '(
  (bassdrum default #f ,(ly:make-pitch -1 2 0))
  (lowtom    diamond #f ,(ly:make-pitch -1 6 0))
) (get-drum-kit 'drums)))
```

```
\include "drumpitch-init.ly"
up = \notes { crashcymbal4 hihat8 halfopenhihat hh hh hh openhihat }
down = \notes { bassdrum4 snare8 bd r bd sn4 }
\include "nederlands.ly"
bass = \notes \transpose c c,, { a4. e8 r e g e }
\score {
  <<
    \apply #(drums->paper 'drums) \new Staff <<
      \clef percussion
      \new Voice { \voiceOne \up }
      \new Voice { \voiceTwo \down }
    >>
    \new Staff { \clef "F_8" \bass }
  >>
}
```



3.9.2 Percussion MIDI output

In order to produce correct MIDI output you need to produce two score blocks—one for the paper and one for the MIDI output. To use the percussion channel you set the property `instrument` to `'drums'`. Because the drum-pitches themselves are similar to the general MIDI pitches all you have to do is to insert the voices with none of the scheme functions to get the correct MIDI output:

```
\score {
  \apply #(drums->paper 'mydrums) \context Staff <<
    \clef percussion
    { \up } \\\
    { \down }
  >>
  \paper{}
}
\score {
  \context Staff <<
    \property Staff.instrument = #'drums
    \up \down
  >>
  \midi{}
}
```

Bugs

Chords entered with `< ... >` do not work. This scheme is a temporary implementation.

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 staff so it reaches the stem in the upper staff, 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'`. ■

3.10.1 Automatic staff changes

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

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

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

The autochanger switches on basis of pitch (central 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 Staff \context Voice = VA << \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

`AutoChangeMusic`.

Bugs

The staff switches often do not end up in optimal places. For high quality output, staff switches should be specified manually.

3.10.2 Manual staff switches

Voices can be switched between staves manually, using the following command:

```
\translator 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".

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.

Syntax

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 the generated documentation of `SustainPedal` for more information.

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

```

\property Staff.pedalSustainStyle = #'bracket
c''4\sustainDown d''4 e''4
a'4\sustainUp\sustainDown
f'4 g'4 a'4\sustainUp

```



A third style of pedal notation is a mixture of text and brackets, obtained by setting `pedal-type` to `mixed`:

```
\property Staff.pedalSustainStyle = #'mixed
c''4\sustainDown d''4 e''4
c'4\sustainUp\sustainDown
f'4 g'4 a'4\sustainUp
```



The default ‘*Ped’ style for sustain and damper pedals corresponds to `\pedal-type = #'text`. However, `mixed` is the default style for a `sostenuto` pedal:

```
c''4\sostenutoDown d''4 e''4 c'4 f'4 g'4 a'4\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:

```
\property Staff.PianoPedalBracket \override
  #'shorten-pair = #'(0 . -1.0)
c''4\sostenutoDown d''4 e''4 c'4
f'4 g'4 a'4\sostenutoUp
```



3.10.4 Arpeggio

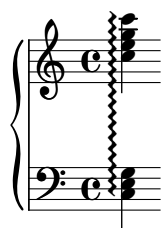
You can specify an arpeggio sign on a chord by attaching an `\arpeggio` to a chord:

```
<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 <<
  \property 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. This can be typeset by setting `arpeggio-direction`:

```
\context Voice {
  \property Voice.Arpeggio \set #'arpeggio-direction = #1
  <c e g c>\arpeggio
  \property Voice.Arpeggio \set #'arpeggio-direction = #-1
  <c e g c>\arpeggio
}
```



A square bracket on the left indicates that the player should not arpeggiate the chord. To draw these brackets, set the `molecule-callback` property of `Arpeggio` or `PianoStaff.Arpeggio` objects to `\arpeggioBracket`, and use `\arpeggio` statements within the chords as before:

```
\property PianoStaff.Arpeggio \override
  #'molecule-callback = \arpeggioBracket
  <c' e g c>\arpeggio
```



Predefined commands

`\arpeggioBracket`, `\arpeggio`.

See also

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

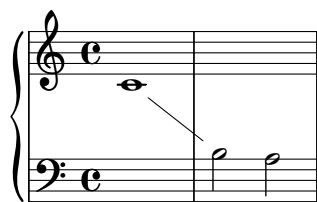
Bugs

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



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 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 an alphabetic sequence, is included in the resulting string. As a consequence, spaces must be inserted around `\property` commands:

```
\property Lyrics . LyricText \set #'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\""} }
```

Hyphens can be entered as ordinary hyphens at the end of a syllable, i.e.

```
soft- ware
```

These will be attached to the end of the first syllable.

Centered hyphens are entered using the special `--` lyric as a separate word 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

`LyricEvent`, `HyphenEvent`, and `ExtenderEvent`.

Bugs

The definition of lyrics mode is too complex.

3.11.2 The Lyrics context

Lyrics are printed by interpreting them in `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 `\addlyrics` expression:

```
\addlyrics
  \notes ...
  \context Lyrics ...
```

Normally, this will put the lyrics below the staff. 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 = sopr { s1 }
  \context Staff = soprStaff { s1 }
  \context Lyrics = tenor { s1 }
  \context Staff = tenorStaff { s1 }
>>
```

and then combine the appropriate melodies and lyric lines:

```
\addlyrics
  \context Staff = soprStaff the music
  \context Lyrics = sopr the lyrics
```

putting both together, you would get

```
\context ChoirStaff \notes <<
  \context Lyrics = ...
  \context Staff = ...
  \addlyrics ...
>>
```

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

See also

`LyricCombineMusic`, `Lyrics`, and ‘`input/template/satb.ly`’.

Bugs

`\addlyrics` is not automatic enough: melismata are not detected automatically, and melismata are not stopped when they hit a rest. A melisma on the last note in a melody is not printed.

3.11.3 More stanzas

When multiple stanzas are printed underneath each other, the vertical groups of syllables should be aligned around punctuation. This can be done automatically when corresponding lyric lines and melodies are marked.

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

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

Then set the `LyricsVoice` 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 `LyricsVoices` are marked `duet-1` and `duet-2`:

```

\context LyricsVoice = "duet-1" {
  Hi, my name is bert. }
\context LyricsVoice = "duet-2" {
  Ooooo, ch\’e -- ri, je t’aime. }

```

The convention for naming `LyricsVoice` and `Voice` must also be used to get melismata correct in conjunction with rests.

The complete example is shown here:

```

\score {
\addlyrics
\notes \relative c'' \context Voice = duet { \time 3/4
  g2 e4 a2 f4 g2. }
\lyrics \context Lyrics <<
\context LyricsVoice = "duet-1" {
  \property LyricsVoice . stanza = "Bert"
  Hi, my name is bert. }
\context LyricsVoice = "duet-2" {
  \property LyricsVoice . stanza = "Ernie"
  Ooooo, ch\’e -- ri, je t’aime. }
>>
}

```



Bert Hi, my name is bert.
 Ernie Ooooo, ché – ri, je t’aime.

Stanza numbers, or the names of the singers can be added by setting `LyricsVoice.Stanza` (for the first system) and `LyricsVoice.stz` for the following systems. Notice how dots are surrounded with spaces in `\lyrics` mode:

```

\property LyricsVoice . stanza = "Bert"
...
\property LyricsVoice . stanza = "Ernie"

```

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

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* (plural: *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. Most musical instruments have their ambitus standardized (or at least there is agreement upon the minimal ambitus of a particular type of instrument), such that a composer or arranger of a piece of music can easily meet the ambitus constraints of the targeted instrument. However, the ambitus of the human voice depends on individual physiological state, including education and training of the voice. Therefore, a singer potentially has to check for each piece of music if

the ambitus of that piece meets his individual capabilities. This is why the ambitus of a piece may be of particular value to vocal performers.

The ambitus is typically notated on a per-voice basis at the very beginning of a piece, e.g. nearby the initial clef or time signature of each staff. The range is graphically specified by two noteheads, that represent the minimum and maximum pitch. Some publishers use a textual notation: they put the range in words in front of the corresponding staff. LilyPond only supports the graphical ambitus notation.

To apply, add the `Ambitus_engraver` to the `Voice` context, i.e.

```
\paper {
  \translator {
    \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, then 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

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

Bugs

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

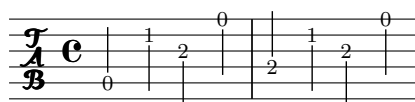
3.12 Tablatures

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:

```
e8 fis gis a b cis' dis' e'
\property TabStaff.minimumFret = #8
e8 fis gis a b cis' dis' e'
```

See also

`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` (the `line-count` property of `TabStaff` can only be changed using `\applyoutput`, for more information, see Section 3.17.1 [Tuning objects], page 101).

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 central 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 <<

\applyoutput #(outputproperty-compatibility (make-type-checker 'staff-symbol-inter:
'line-count 4)
\property TabStaff.stringTunings = #'(-5 -10 -15 -20)

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



It is possible to change the Scheme function to format the tablature note text. The default is `fret-number-tablature-format`, which uses the fret number. For instruments that do not use this notation, you can create a special tablature formatting function. This function takes three argument: string number, string tuning and note pitch.

Bugs


No guitar special effects have been implemented.

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 >> }
  C      Fsus4  Bb/F  C      Fsus4  F6/sus4
```



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 }
```



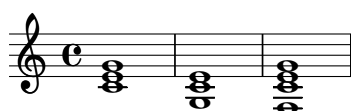
Since an unaltered 11 does not sound good when combined with an unaltered 13, the 11 is removed in this case (unless it is added explicitly):

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



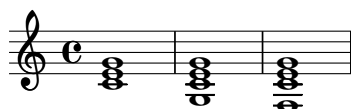
An inversion (putting one pitch of the chord on the bottom), as well as bass notes, can be specified by appending */pitch* to the chord:

```
\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` 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 `>`:

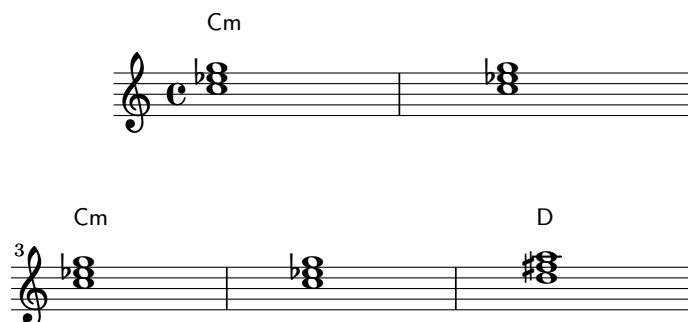
```
scheme = \notes {
  \chords {a1 b c} <d' f' a'> <e' g' b'>
}
\score {
  \notes<<
    \context ChordNames \scheme
    \context Staff \scheme
  >>
}
```

A B C Dm Em



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 {
      \property ChordNames.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 Chapter 4 [Literature list], page 114). 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
  \property ChordNames.chordNameSeparator
    = \markup { \typewriter "|" }
  c:7sus4 }
```

`C7/sus4C7|sus4`

`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 pre-defined 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 151. Turning on these styles is described in the input file ‘input/test/chord-names-jazz.ly’.

Predefined commands

`\germanChords`, `\semiGermanChords`.

See also

‘input/regression/chord-name-major7.ly’, ‘input/regression/chord-name-exceptions.ly’, ‘input/test/chord-names-jazz.ly’, ‘input/test/chord-names-german.ly’, ‘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

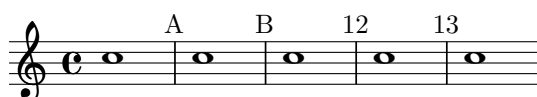
Polyphonic scores consist of many staves. These staves can be constructed in three different ways:

- The group is started with a brace at the left. This is done with the `GrandStaff` context.
- The group is started with a bracket. This is done with the `StaffGroup` context
- The group is started with a vertical line. 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 "A"
  c1 \mark "B"
  c1 \mark "12"
  c1 \mark "13"
  c1
}
```



The mark is incremented automatically if you use `\mark \default`. The value to use is stored in the property `rehearsalMark` is used and automatically incremented.

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

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



In this case, during line breaks, marks must also be printed at the end of the line, and not at the beginning. Use the following to force that behavior:

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

See `'input/test/boxed-molecule.ly'` for putting boxes around the marks.

See also

`MarkEvent`, `RehearsalMark`, and `'input/test/boxed-molecule.ly'`.

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

`BarNumber`, `'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:

```
\property Staff.instrument = "ploink " { c''4 }
```



You can also use markup texts to construct more complicated instrument names:

```
\notes {
  \property Staff.instrument = \markup {
    \column < "Clarineti"
    { "in B"
      \smaller \musicglyph #"accidentals--1"
    }
    >
  }
  { c''1 }
}
```



See also

`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 *from* and *to*.

`\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

TransposedMusic, and UntransposableMusic.

Bugs

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

3.14.6 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
\property 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:

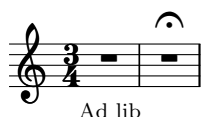
```
\property Score.skipBars = ##t
\time 3/4
R2. | R2.*2
\time 13/8
R1*13/8
R1*13/8*12
```



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 3.17.5 [Text markup], page 106). 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

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, then the result will look OK, but the bar numbering will be off.

3.14.7 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.

Syntax

The syntax for part combining is

```
\partcombine context musicexpr1 musicexpr2
```

where the pieces of music *musicexpr1* and *musicexpr2* will be combined into one context of type *context*. The music expressions must be interpreted by contexts whose names should start with *one* and *two*.

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


```

\context Staff <<
  \context Voice=one \partcombine Voice
  \context Thread=one \relative c'' {
    g a( b) r
  }
  \context Thread=two \relative c'' {
    g r4 r f
  }
>>

```



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:

```

\context Staff <<
  \property Staff.soloADue = ##f
  \context Voice=one \partcombine Voice
  \context Thread=one \relative c'' {
    b4 a c g
  }
  \context Thread=two \relative c'' {
    d,2 a4 g'
  }
>>

```



See also

PartCombineMusic, *Thread_devnull_engraver*, and *Voice_devnull_engraver* and *A2_engraver*.

Bugs

The syntax for naming contexts is inconsistent with the syntax for combining stanzas.

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:



The part combiner is rather buggy, and it will be replaced by a better mechanism in the near future.

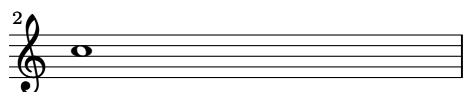
3.14.8 Hiding staves

In orchestral scores, staff lines that only have rests are usually removed. This saves some space. This style is called ‘French Score’. For *Lyrics*, *LyricsVoice*, *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
    \translator { \RemoveEmptyStaffContext }
  }
}
```



The first page shows all staves in full. If they should be removed from the first page too, set `remove-first` to `false` in `RemoveEmptyVerticalGroup`.

3.14.9 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 \\\
    {
      \property Voice.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 image shows a musical score with three staves. The top staff is labeled 'both', the middle 'part', and the bottom 'score'. All staves are in treble clef and common time (C). The 'both' and 'part' staves have a 'cue' annotation under the second measure. The 'score' staff has a whole rest in the second measure. The music is in 4/4 time.

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

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

See also

`'input/regression/tag-filter.ly'`

3.14.10 Sound output for transposing instruments

When you want to make a MIDI file from a score containing transposed and untransposed instruments, you have to instruct LilyPond the pitch offset (in semitones) for the transposed instruments. This is done using the `transposing` property. It does not affect printed output:

```
\property Staff.instrument = #"Cl. in B-flat"
\property Staff.transposing = #-2
```

3.15 Ancient notation

Support for ancient notation is still under heavy development. Regardless of all of the current limitations (see the bugs section below for details), it 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 86, Section 3.15.2 [Ancient accidentals], page 86, Section 3.15.3 [Ancient rests], page 87, Section 3.15.4 [Ancient clefs], page 87, Section 3.15.5 [Ancient flags], page 89 and Section 3.15.6 [Ancient time signatures], page 90. By manipulating such a grob property, the typographical appearance of the affected graphical objects can be accommodated for a specific notation flavour 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 91, Section 3.15.8 [Divisiones], page 92, Section 3.15.9 [Ligatures], page 92, and Section 3.15.10 [Figured bass], page 98.

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, then you may have a look at the predefined contexts (see Section 3.15.11 [Vaticana style contexts], page 99). 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

Syntax

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:

```
\property Voice.NoteHead \set #'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 92 for how ligature engravers work.

See also

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

Section 3.9.1 [Percussion staves], page 63 use note head styles of their own that are frequently used in contemporary music notation.

3.15.2 Ancient accidentals

Syntax

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

```
vaticana medicaea hufnagel mensural
b  ♯  ♮  ♭  ♮  ♯
```

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

Section 3.1.2 [Pitches], page 32, Section 3.1.3 [Chromatic alterations], page 33 and Section 3.6 [Accidentals], page 48 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.

3.15.3 Ancient rests

Syntax

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 mimicks the appearance of rests as in historic prints of the 16th century.

The following example demonstrates the `neo_mensural` style:

```
\property Voice.Rest \set #'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 92.

See also

Section 3.1.5 [Rests], page 34 gives a general introduction into the use of rests.







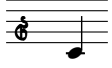



3.15.4 Ancient clefs


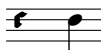


Syntax

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 41. 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
------------	-------------	-----------------	---------

<code>clefs-neo_mensural_c</code>	modern style mensural C clef	<code>neo_mensural_c1</code> , <code>neo_mensural_c2</code> , <code>neo_mensural_c3</code> , <code>neo_mensural_c4</code>	
<code>clefs-petrucchi_c1</code> <code>clefs-petrucchi_c2</code> <code>clefs-petrucchi_c3</code> <code>clefs-petrucchi_c4</code> <code>clefs-petrucchi_c5</code>	petrucci style mensural C clefs, for use on different stafflines (the examples shows the 2nd staffline C clef).	<code>petrucci_c1</code> <code>petrucci_c2</code> <code>petrucci_c3</code> <code>petrucci_c4</code> <code>petrucci_c5</code>	
<code>clefs-petrucchi_f</code>	petrucci style mensural F clef	<code>petrucci_f</code>	
<code>clefs-petrucchi_g</code>	petrucci style mensural G clef	<code>petrucci_g</code>	
<code>clefs-mensural_c</code>	historic style mensural C clef	<code>mensural_c1</code> , <code>mensural_c2</code> , <code>mensural_c3</code> , <code>mensural_c4</code>	
<code>clefs-mensural_f</code>	historic style mensural F clef	<code>mensural_f</code>	
<code>clefs-mensural_g</code>	historic style mensural G clef	<code>mensural_g</code>	
<code>clefs-vaticana_do</code>	Editio Vaticana style do clef	<code>vaticana_do1</code> , <code>vaticana_do2</code> , <code>vaticana_do3</code>	
<code>clefs-vaticana_fa</code>	Editio Vaticana style fa clef	<code>vaticana_fa1</code> , <code>vaticana_fa2</code>	
<code>clefs-medicaea_do</code>	Editio Medicaea style do clef	<code>medicaea_do1</code> , <code>medicaea_do2</code> , <code>medicaea_do3</code>	

<code>clefs-medicaea_fa</code>	Editio Medicaea style fa clef	<code>medicaea_fa1,</code> <code>medicaea_fa2</code>	
<code>clefs-hufnagel_do</code>	historic style hufnagel do clef	<code>hufnagel_do1,</code> <code>hufnagel_do2,</code> <code>hufnagel_do3</code>	
<code>clefs-hufnagel_fa</code>	historic style hufnagel fa clef	<code>hufnagel_fa1,</code> <code>hufnagel_fa2</code>	
<code>clefs-hufnagel_do_fa</code>	historic style hufnagel combined do/fa clef	<code>hufnagel_do_fa</code>	

Modern style means “as is typeset in contemporary editions of transcribed mensural music”.

Petrucchi 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”.

Petrucchi used C clefs with differently balanced left-side vertical beams, depending on which staffline it is printed.

See also

For modern clefs, see Section 3.3.3 [Clef], page 41. For the percussion clef, see Section 3.9.1 [Percussion staves], page 63. For the TAB clef, see Section 3.12 [Tablatures], page 73.

3.15.5 Ancient flags

Syntax

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

```
\property Voice.Stem \set #'flag-style = #'mensural
\property Voice.Stem \set #'thickness = #1.0
\property Voice.NoteHead \set #'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 behaviour, 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

Syntax

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:

C	C	C	C
<code>\time 4/4</code>	<code>\time 2/2</code>	<code>\time 6/4</code>	<code>\time 6/8</code>

O	O	O	O
<code>\time 3/2</code>	<code>\time 3/4</code>	<code>\time 9/4</code>	<code>\time 9/8</code>

O	D
<code>\time 4/8</code>	<code>\time 2/4</code>

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 mimicks 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

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.

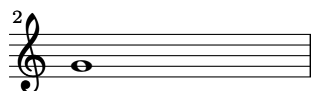
Custodes were frequently used in music notation until the 17th century. Nowadays, they have survived only in a few particular forms of musical notation such as contemporary editions of Gregorian chant like the *editio vaticana*. There are different custos glyphs used in different flavours of notational style.

Syntax

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

```
\paper {
  \translator {
    \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:

```
vaticana medicaea hufnagel mensural
  |         |         ✓         ✗
```

If the boolean property `adjust-if-on-staffline` is set to `#t` (which it is by default), lily typesets slightly different variants of the custos glyph, depending on whether the custos, is typeset on or between stafflines. The glyph will optically fit well into the staff, with the appendage on the right of the custos always ending at the same vertical position between two stafflines regardless of the pitch. If you set `adjust-if-on-staffline` to `#f`, then a compromise between both forms is used.

Just like stems can be attached to noteheads in two directions *up* and *down*, each custos glyph is available with its appendage pointing either up or down. If the pitch of a custos is above a selectable position, the appendage will point downwards; if the pitch is below this position, the appendage will point upwards. Use property `neutral-position` to select this position. By default, it is set to 0, such that the neutral position is the center of the staff. Use property `neutral-direction` to control what happens if a custos is typeset on the neutral position itself. By default, this property is set to -1, such that the appendage will point downwards. If set to 1, the appendage will point upwards. Other values such as 0 are reserved for future extensions and should not be used.

See also

Custos and ‘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 53. 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.

Syntax

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`:

The image shows two musical staves illustrating the use of divisiones and finalis. The first staff shows three measures of music, each ending with a different division symbol: a vertical line for *divisio minima*, a vertical line with a horizontal bar for *divisio maior*, and a vertical line with a horizontal bar and a small circle for *divisio maxima*. The second staff shows four measures of music, each ending with a different division symbol: a double bar line for *finalis*, a vertical line with a horizontal bar for *virgula*, a vertical line with a horizontal bar and a small circle for *caesura*, and a vertical line with a horizontal bar and a small circle for *caesura*. The lyrics "Blah blub, blah blam." are written below each measure.

Predefined commands

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

See also

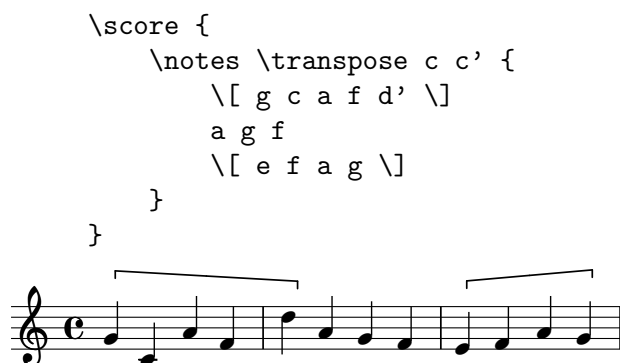
`BreathingSign`, `BreathingSignEvent`, ‘`input/test/divisiones.ly`’, and Section 3.7.3 [Breath marks], page 53.

3.15.9 Ligatures

In musical terminology, 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 as an allusion to the accent symbols of greek lyric poetry to denote ascending or descending sequences of notes. Both, the shape and the exact meaning of ligatures changed tremendously during the following centuries: In early notation, ligatures were used for monophonic tunes (Gregorian chant) and very soon denoted also the way of performance in the sense of articulation. With upcoming multiphony, the need for a metric system arised, since multiple voices of a piece have to be synchronized some way. New notation systems were invented that used the manifold shapes of ligatures to now denote rhythmical patterns (e.g. black mensural notation, mannered notation, ars nova). With the invention of the metric system of the white mensural notation, the need for ligatures to denote such patterns disappeared. Nevertheless, ligatures were still in use in the mensural system for a couple of decades until they finally disappeared during the late 16th / early 17th century. Still, ligatures have survived in contemporary editions of Gregorian chant such as the Editio Vaticana from 1905/08.

Syntax

Syntactically, ligatures are simply enclosed by `\[` and `\]`. Some ligature styles (such as *Editio Vaticana*) 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:



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. Support for *Editio Vaticana* will be added in the future.

3.15.9.1 White mensural ligatures

There is limited support for white mensural ligatures. The implementation is still experimental; it may output strange warnings or even crash in some cases or produce weird results on more complex ligatures.

Syntax

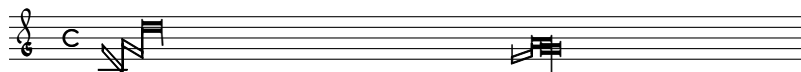
To engrave white mensural ligatures, in the paper block the `Mensural_ligature_engraver` has to be put into the `Voice` context, and remove the `Ligature_bracket_engraver`:

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

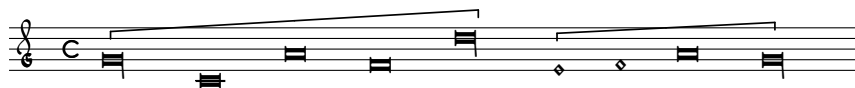
There is no additional input language to describe the shape of a white mensural ligature. The shape is rather determined solely from the pitch and duration of the enclosed notes. While this approach may take a new user a while to get accustomed, it has the great advantage that the full musical information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

```
\property Score.timing = ##f
\property Score.defaultBarType = "empty"
\property Voice.NoteHead \set #'style = #'neo_mensural
\property Staff.TimeSignature \set #'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:



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 Strophæ	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 ▮	

	u	v	
8. Quilisma Pes			
	w	x	
9. Podatus Initio Debilis			
	y	z	A
10. Torculus			
	B	C	D
11. Torculus Initio Debilis			
	E	F	G
12. Porrectus			
	H	I	J
13. Climacus			
	K	L	M
14. Scandicus			
	N	O	
15. Salicus			
	P		
16. Trigonus			

Syntax

Unlike most other neumes notation systems, the input language for neumes does not necessarily reflect directly the typographical appearance, but is designed to solely focus on musical meaning. For example, `\[a \pes b \flexa g \]` produces a Torculus consisting of three Punctum heads, while `\[a \flexa g \pes b \]` produces a Porrectus with a curved flexa shape and only a single Punctum head. There is no command to explicitly typeset the curved flexa shape; the decision of when to typeset a curved flexa shape is 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 such as Hufnagel (also known as German gothic neumes) or Medicaea (kind of a very simple forerunner of the Editio Vaticana). As soon as Hufnagel ligature engraver and Medicaea ligature engraver will have been implemented, it will be as simple as replacing the ligature engraver in the **Voice** context to get the desired notation style from the same input.

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	<code>\[b \]</code>
b	Punctum Inclinatorum	<code>\[\inclinatorum b \]</code>
c	Punctum Auctum Ascendens	<code>\[\auctum \ascendens b \]</code>
d	Punctum Auctum Descendens	<code>\[\auctum \descendens b \]</code>
e	Punctum Inclinatorum Auctum	<code>\[\inclinatorum \auctum b \]</code>
f	Punctum Inclinatorum Parvum	<code>\[\inclinatorum \deminutum b \]</code>
g	Virga	<code>\[\virga b \]</code>
h	Stropha	<code>\[\stropha b \]</code>
i	Stropha Aucta	<code>\[\stropha \auctum b \]</code>
j	Oriscus	<code>\[\oriscus b \]</code>
k	Clivis vel Flexa	<code>\[b \flexa g \]</code>
l	Clivis Aucta Descendens	<code>\[b \flexa \auctum \descendens g \]</code>
m	Clivis Aucta Ascendens	<code>\[b \flexa \auctum \ascendens g \]</code>
n	Cephalicus	<code>\[b \flexa \deminutum g \]</code>
o	Podatus vel Pes	<code>\[g \pes b \]</code>
p	Pes Auctus Descendens	<code>\[g \pes \auctum \descendens b \]</code>
q	Pes Auctus Ascendens	<code>\[g \pes \auctum \ascendens b \]</code>
r	Epiphonus	<code>\[g \pes \deminutum b \]</code>
s	Pes Quassus	<code>\[\oriscus g \pes \virga b \]</code>
t	Pes Quassus Auctus Descendens	<code>\[\oriscus g \pes \auctum \descendens b \]</code>
u	Quilisma Pes	<code>\[\quilisma g \pes b \]</code>
v	Quilisma Pes Auctus Descendens	<code>\[\quilisma g \pes \auctum \descendens b \]</code>

w	Pes Initio Debilis	<code>\[\deminutum g \pes b \]</code>
x	Pes Auctus Descendens Initio Debilis	<code>\[\deminutum g \pes \auctum \descendens b \]</code>
y	Torculus	<code>\[a \pes b \flexa g \]</code>
z	Torculus Auctus Descendens	<code>\[a \pes b \flexa \auctum \descendens g \]</code>
A	Torculus Deminutus	<code>\[a \pes b \flexa \deminutum g \]</code>
B	Torculus Initio Debilis	<code>\[\deminutum a \pes b \flexa g \]</code>
C	Torculus Auctus Descendens Initio Debilis	<code>\[\deminutum a \pes b \flexa \auctum \descendens g \]</code>
D	Torculus Deminutus Initio Debilis	<code>\[\deminutum a \pes b \flexa \deminutum g \]</code>
E	Porrectus	<code>\[a \flexa g \pes b \]</code>
F	Porrectus Auctus Descendens	<code>\[a \flexa g \pes \auctum \descendens b \]</code>
G	Porrectus Deminutus	<code>\[a \flexa g \pes \deminutum b \]</code>
H	Climacus	<code>\[\virga b \inclinatum a \inclinatum g \]</code>
I	Climacus Auctus	<code>\[\virga b \inclinatum a \inclinatum \auctum g \]</code>
J	Climacus Deminutus	<code>\[\virga b \inclinatum a \inclinatum \deminutum g \]</code>
K	Scandicus	<code>\[g \pes a \virga b \]</code>
L	Scandicus Auctus Descendens	<code>\[g \pes a \pes \auctum \descendens b \]</code>
M	Scandicus Deminutus	<code>\[g \pes a \pes \deminutum b \]</code>
N	Salicus	<code>\[g \oriscus a \pes \virga b \]</code>
O	Salicus Auctus Descendens	<code>\[g \oriscus a \pes \auctum \descendens b \]</code>
P	Trigonus	<code>\[\stroph a b \stroph a b \stroph a a \]</code>

Predefined commands

The following head prefixes are supported:

`\virga`, `\strophæ`, `\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.

Bugs

Trigonus: apply equal spacing, regardless of pitch.

3.15.10 Figured bass

Syntax

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+] >
    }
  >>
```



6 7 6#6 6

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>

6
4

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

<4- 6+ 7!>

7 b
6 #
4 b

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

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

[12]
b
8
[6]
4

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

BassFigureEvent music, BassFigure object, and FiguredBass context.

Bugs

Slash notation for alterations is not supported.

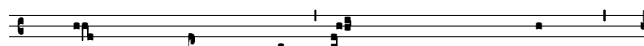
3.15.11 Vaticana style contexts

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:

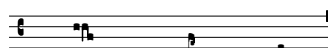
```

\include "gregorian-init.ly"
\score {
  \addlyrics
  \context VaticanaVoice {
    \property Score.BarNumber \set #'transparent = ##t
    \notes {
      \[ c'( c' \flexa a \] \[ a \flexa \deminutum g) \] f \divisioMinima
      \[ f( \pes a c' c' \pes d') \] c' \divisioMinima \break
      \[ c'( c' \flexa a \] \[ a \flexa \deminutum g) \] f \divisioMinima
    }
  }
  \context Lyrics \lyrics {
    San- ctus, San- ctus, San- ctus
  }
}

```



San-ctus, San-ctus,



San-ctus

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 Chapter 4 [Literature list], page 114). 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

In musical terminology, a *cluster* denotes a range of simultaneously sounding pitches that may change over time. The set of available pitches to apply usually depends on the acoustic source. Thus, in piano music, a cluster typically consists of a continuous range of the semitones as provided by the piano's fixed set of a chromatic scale. In choral music, each singer of the choir typically may sing an arbitrary pitch within the cluster's range that is not bound to any diatonic, chromatic or other scale. In electronic music, a cluster (theoretically) may even cover a continuous range of pitches, thus resulting in coloured noise, such as pink noise.

Clusters can be denoted in the context of ordinary staff notation by engraving simple geometrical shapes that replace ordinary notation of notes. Ordinary notes as musical events specify starting time and duration of pitches; however, the duration of a note is expressed by the shape of the note head rather than by the horizontal graphical extent of the note symbol. In contrast, the shape of a cluster geometrically describes the development of a range of pitches (vertical extent) over time (horizontal extent). Still, the geometrical shape of a cluster covers the area in which any single pitch contained in the cluster would be notated as an ordinary note. From this point of view, it is reasonable to specify a cluster as the envelope of a set of notes.

Syntax

A cluster is engraved as the envelope of a set of cluster-notes. Cluster notes are created 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:



By default, `Cluster_spanner_engraver` is in the `Voice` context. This allows putting ordinary notes and clusters 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

`ClusterSpanner`, `ClusterSpannerBeacon`, `'input/regression/cluster.ly'`, `Cluster_spanner_engraver`, and `ClusterNoteEvent`.

Bugs

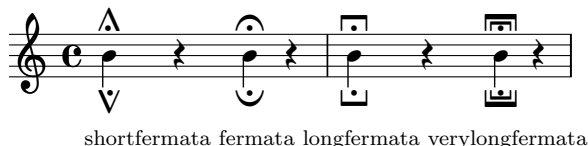
Music expressions like `<< { g8 e8 } a4 >>` are not printed accurately. Use `<g a>8 <e a>8` instead.

3.16.2 Fermatas

Contemporary music notation frequently uses special fermata symbols to indicate fermatas of differing lengths.

Syntax

The following 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 Tuning output

There are situations where default layout decisions are not sufficient. In this section we discuss ways to override these defaults.

Formatting is internally done by manipulating so called objects (graphic objects). Each object carries with it a set of properties (object or layout properties) specific to that object. For example, a stem object has properties that specify its direction, length and thickness.

The most direct way of tuning the output is by altering the values of these properties. There are two ways of doing that: first, you can temporarily change the definition of one type of object, thus affecting a whole set of objects. Second, you can select one specific object, and set a layout property in that object.

Do not confuse layout properties with translation properties. Translation properties always use a mixed caps style naming, and are manipulated using `\property`:

```
\property Context.propertyName = value
```

Layout properties are use Scheme style variable naming, i.e. lower case words separated with dashes. They are symbols, and should always be quoted using `#'`. For example, this could be an imaginary layout property name:

```
#'layout-property-name
```

3.17.1 Tuning objects

The definition of an object is a list of default object properties. For example, the definition of the Stem object (available in `'scm/define-grobs.scm'`), includes the following definitions for Stem:

```
(thickness . 1.3)
(beamed-lengths . (0.0 2.5 2.0 1.5))
(Y-extent-callback . ,Stem::height)
...
```

Adding variables on top of this existing definition overrides the system default, and alters the resulting appearance of the layout object.

Syntax

Changing a variable for only one object is commonly achieved with `\once`:

```
\once \property context.objectname
  \override symbol = value
```

Here *symbol* is a Scheme expression of symbol type, *context* and *objectname* is a string and *value* is a Scheme expression. This command applies a setting only during one moment in the score.

In the following example, only one Stem object is changed from its original setting:

```
c4
\once \property Voice.Stem \set #'thickness = #4
c4
c4
```



For changing more objects, the same command, without `\once` can be used:

```
\property context.objectname \override symbol = value
```

This command adds `symbol = value` to the definition of `objectname` in the context `context`, and this definition stays in place until it is removed.

An existing definition may be removed by the following command:

```
\property context.objectname \revert symbol
```

All `\override` and `\revert` commands should be balanced. The `\set` shorthand performs a revert followed by an override, and is often more convenient to use

```
\property context.objectname \set symbol = value
```

Some examples:

```
c'4 \property Voice.Stem \override #'thickness = #4.0
c'4
c'4 \property Voice.Stem \revert #'thickness
c'4
```



The following example gives exactly the same result as the previous one (assuming the system default for stem thickness is 1.3):

```
c'4 \property Voice.Stem \set #'thickness = #4.0
c'4
c'4 \property Voice.Stem \set #'thickness = #1.3
c'4
```



Reverting a setting which was not set in the first place has no effect. However, if the setting was set as a system default, this may remove the default value, and this may give surprising results, including crashes. In other words, `\override` and `\revert` must be carefully balanced. The following are examples of correct nesting of `\override`, `\set`, `\revert`:

- a clumsy but correct form:

```
\override \revert \override \revert \override \revert
```

- shorter version of the same:

```
\override \set \set \revert
```

- a short form, using only `\set`. This requires you to know the default value:

```
\set \set \set \set to default value
```

- if there is no default (i.e. by default, the object property is unset), then you can use

```
\set \set \set \revert
```

The object description is an Scheme association list. Since a Scheme list is a singly linked list, we can treat it as a stack, and `\override` and `\revert` are push and pop operations. The association list is stored in a normal context property, hence

```
\property Voice.NoteHead = #'()
```

will effectively erase `NoteHeads` from the current `Voice`. Typically, this will blank the object. However, this mechanism should not be used: it may cause crashes or other anomalous behavior.

See also

`OverrideProperty`, `RevertProperty`, `PropertySet`, `All-backend-properties`, and `All-layout-objects`.

Bugs

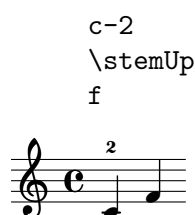
The backend is not very strict in type-checking object properties. Cyclic references in Scheme values for properties cause hangs and/or crashes. Reverting properties that are system defaults may also lead to crashes.

3.17.2 Constructing a tweak

Three pieces of information are required to use `\override` and `\set`: the name of the layout object, the context and the name of the property. We demonstrate how to glean this information from the notation manual and the generated documentation.

The generated documentation is a set of HTML pages which should be included if you installed a binary distribution, typically in `/usr/share/doc/lilypond`. They are also available on the web: go to the LilyPond website (<http://lilypond.org>), click “Documentation”, select the correct version, and click then “Program reference.” It is advisable to bookmark the local HTML files. They will load faster than the ones on the web. If you use the version from the web, you must check whether the documentation matches the program version: it is generated from the definitions that the program uses, and therefore it is strongly tied to the LilyPond version.

Suppose we want to move the fingering indication in the fragment below:



If you visit the documentation of **Fingering** (in Section 3.7.8 [Fingering instructions], page 55), you will notice that there is written:

See also

`FingerEvent` and `Fingering`.

In other words, the fingerings once entered, are internally stored as `FingerEvent` music objects. When printed, a `Fingering` layout object is created for every `FingerEvent`.

The `Fingering` object has a number of different functions, and each of those is captured in an interface. The interfaces are listed under `Fingering` in the program reference.

The `Fingering` object has a fixed size (`item-interface`), the symbol is a piece of text (`text-interface`), whose font can be set (`font-interface`). It is centered horizontally (`self-alignment-interface`), it is placed next to other objects (`side-position-interface`) vertically, and its placement is coordinated with other scripts (`text-script-interface`). It also

has the standard **grob-interface** (grob stands for Graphical object) with all the variables that come with it. Finally, it denotes a fingering instruction, so it has **finger-interface**.

For the vertical placement, we have to look under **side-position-interface**:

side-position-interface

Position a victim object (this one) next to other objects (the support). In this case, 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.
Default value: 0.6

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 \property Voice.Fingering \set #'padding = #3
```

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

```
\once \property Voice.Fingering
  \set #'padding = #3
c-2
\stemUp
f
  2
```



The context name **Voice** in the example above can be determined as follows. In the documentation for **Fingering**, it says

Fingering grobs are created by: **Fingering_engraver**

Clicking **Fingering_engraver** shows the documentation of the module responsible for interpreting the fingering instructions and translating them to a **Fingering** object. Such a module is called an *engraver*. The documentation of the **Fingering_engraver** says

Fingering_engraver is part of contexts: **Voice**

so tuning the settings for Fingering should be done with

```
\property Voice.Fingering \set ...
```

Of course, the tweak may also be done in a larger context than **Voice**, for example, **Staff** or **Score**.

See also

The program reference also contains alphabetical lists of **Contexts**, **All-layout-objects** and **Music-expressions**, so you can also find which objects to tweak by browsing the internals document.

3.17.3 Applyoutput

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 simple example of `\applyoutput`; it blanks note-heads on the center-line:

```
(define (blanker grob grob-origin context)
  (if (and (memq (ly:get-grob-property grob 'interfaces)
                note-head-interface)
        (eq? (ly:get-grob-property grob 'staff-position) 0))
      (ly:set-grob-property! grob 'transparent #t)))
```

3.17.4 Font selection

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 an integer: negative numbers make the font smaller, positive numbers larger. An example is given below:

```
c4 c4 \property Voice.fontSize = #-1
f4 g4
```



This command will set `font-relative-size` (see below), and does not change the size of variable symbols, such as beams or slurs.

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

The size of the font may be scaled with the object property `font-magnification`. For example, 2.0 blows up all letters by a factor 2 in both directions.

The font used for printing a object can be selected by setting `font-name`, e.g.

```
\property Staff.TimeSignature
\set #'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.

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-family`

is a symbol indicating the general class of the typeface. Supported are `roman` (Computer Modern), `braces` (for piano staff braces), `music` (the standard music font, including ancient glyphs), `dynamic` (for dynamic signs) 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`.

font-relative-size

is a number indicating the size relative the standard size. For example, with 20pt staff height, relative size -1 corresponds to 16pt staff height, and relative size +1 corresponds to 23 pt staff height.

There are small differences in design between fonts designed for different sizes, hence `font-relative-size` is preferred over `font-magnification` for changing font sizes.

font-design-size

is a number indicating the design size of the font.

This is a feature of the Computer Modern Font: each point size has a slightly different design. Smaller design sizes are relatively wider, which enhances readability.

For any of these properties, the value `*` (i.e. the symbol `*`, entered as `#'*`), acts as a wildcard. This can be used to override default setting, which are always present. For example:

```
\property Lyrics . LyricText \override #'font-series = #'bold
\property Lyrics . LyricText \override #'font-family = #'typewriter
\property Lyrics . LyricText \override #'font-shape = #'*
```

Predefined commands

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

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

Bugs

Relative size is not linked to any real size.

There is no style sheet provided for other fonts besides the T_EX family, and the style sheet cannot be modified easily.

3.17.5 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` only apply to the first following word; 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 }
```

The following size commands set absolute sizes:

```
\teeny
```


`\tiny`

`\small`

`\large`

`\huge`

You can also make letter larger or smaller relative to their neighbors, with the commands `\larger` and `\smaller`.

The following font change commands are defined:

`\dynamic` changes to the font used in dynamic signs. This font does not contain all characters of the alphabet, so when producing “piu f”, the “piu” should be done in a different font.

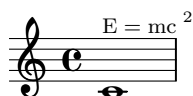
`\number` changes to the font used in time signatures. It only contains numbers and a few punctuation marks.

`\italic` changes font-shape to italic.

`\bold` changes font-series to bold.

Raising and lowering texts can be done with `\super` and `\sub`:

```
c1^\markup { E "=" mc \super "2" }
```



If you want to give an explicit amount for lowering or raising, use `\raise`. This command takes a Scheme valued first argument, and a markup object as second argument:

```
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. They cannot be used to move a single text up or down, when it is above or below a note, since the mechanism that positions it next to the note cancels any vertical shift. For vertical positioning, use the `padding` and/or `extra-offset` properties.

Other commands taking single arguments include

`\bracket`, `\hbracket`

Bracket the argument markup with normal and horizontal brackets respectively.

`\musicglyph`

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 153 for a complete listing of the possible glyphs.

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

`\note log dots dir`

This produces a note with a stem pointing in *dir* direction, with duration *log* and *dots* augmentation dots. The duration *log* is the negative 2-logarithm of the duration denominator. For example, a quarter note has *log* 2, an eighth note 3 and a breve has *log* -1.

\hspace #amount

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.

\fontsize #size

This sets the relative font size, eg.

```
A \fontsize #2 { B C } D
```

This will enlarge the B and the C by two steps.

\translate #(cons x y)

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.

\magnify #mag

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
```

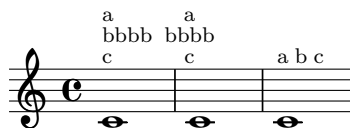
\override #(key . value)

This overrides a formatting property for its argument. The argument should be a key/value pair, e.g.

```
m \override #'(font-family . math) m m
```

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` aligns texts by their center lines:

```
c1^\markup { \column < a bbbb c > }
c1^\markup { \center < a bbbb c > }
c1^\markup { \line < a b c > }
```



The markup mechanism is extensible. Refer to ‘`scm/new-markup.scm`’ for more information.

See also

Markup-functions, and ‘`scm/new-markup.scm`’.

Bugs

Text layout is ultimately done by T_EX, 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 L^AT_EX, so L^AT_EX commands should be used for formatting.

3.18 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 combinations are tried, and the one with the best results—a layout that has uniform density and requires as little stretching or cramping as possible—is chosen. When the score is processed by \TeX , each page is filled with systems, and page breaks are chosen whenever the page gets full.

3.18.1 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

```
\property 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 `\translator` block as follows:

```
\translator {
  \PianoStaffContext
  VerticalAlignment \override #'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

Vertical alignment of staves is handled by the `VerticalAlignment` object.

3.18.2 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, `shortest-duration-space` 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 (2 times `shortest-duration-space`) 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/barline 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` statements are interpreted.

```
\paper { \translator {
  \ScoreContext
  SpacingSpanner \override #'spacing-increment = #3.0
} }
```

See also

`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.

3.18.3 Font size

The Feta font provides musical symbols at seven different sizes. These fonts are 11 point, 13 point, 16 point, 20 point, 23 point, and 26 point. The point size of a font is the height of the corresponding staff (excluding line thicknesses).

Definitions for these sizes are the files ‘`paperSZ.ly`’, where `SZ` is one of 11, 13, 16, 20, 23 and 26. If you include any of these files, the variables `paperEleven`, `paperThirteen`, `paperSixteen`, `paperTwenty`, `paperTwentythree`, and `paperTwentysix` are defined respectively. The default `\paper` block is also set. These files should be imported at toplevel, i.e.

```
\include "paper26.ly"
\score { ... }
```

The default font size settings for each staff heights are generated from the 20pt style sheet. For more details, see the file ‘`scm/font.scm`’.

3.18.4 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 linebreaks at regular intervals use `\break` separated by skips and repeated with `\repeat`:

```
<< \repeat unfold 7 { s1 * 4 \break }
    the real music
>>
```

This makes the following 28 measures (assuming 4/4 time) be broken every 4 measures.

See also

`BreakEvent`.

3.18.5 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 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 variable `textheight` sets the total height of the music on each page. 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 7 [lilypond-book manual], page 133 for more information.

Page breaks are normally computed by `TEX`, so they are not under direct control of LilyPond. However, you can insert a 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, you must first set the `papersize` paper variable variable as in the example below. Set it to the strings `a4`, `letter`, or `legal`. After this specification, you must set the font as described above. If you want the default font, then use the 20 point font.

```
\paper{ papersize = "a4" }
\include "paper16.ly"
```

The file `paper16.ly` will now include a file named ‘`a4.ly`’, which will set the paper variables `hsize` and `vsize` (used by `lilypond-book` and `lilypond`).

Predefined commands

`\newpage`.

See also

Section 6.1 [Invoking lilypond], page 126, ‘`input/regression/between-systems.ly`’, and `NonMusicalPaperColumn`.

Bugs

LilyPond has no concept of page layout, which makes it difficult to reliably choose page breaks in longer pieces.

3.19 Sound

Entered music can also be converted to MIDI output. The performance is good enough 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.

3.19.1 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.

Assignments in the `\midi` block are not allowed.

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`’.

3.19.2 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 152.

Bugs

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

4 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 Bib_T_E_X bibliography of over 100 entries in ‘Documentation/bibliography/’. It is also available online from the website.

Banter 1987

Harald Banter, Akkord Lexikon. Schott’s Söhne 1987. Mainz, Germany ISBN 3-7957-2095-8.

This book is a comprehensive overview of commonly used chords and suggests a unification for all different kinds of chord names.

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.

Hader 1948

Karl Hader, Aus der Werkstatt eines Notenstechers. Waldheim–Eberle Verlag, Vienna 1948.

Hader was the chief-engraver of an Austrian engraving firm. This beautiful booklet was intended as an introduction for laymen on the art of engraving. It contains a step by step, in-depth explanation of how to cut and stamp music into zinc plates. It also contains a few compactly formulated rules on musical orthography. This book is out of print.

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.

Wanske 1988

Helene Wanske, *Musiknotation — Von der Syntax des Notenstichs zum EDV-gesteuerten Notensatz*. Schott-Verlag, Mainz 1988. ISBN 3-7957-2886-x.

A book in two parts: 1. A very thorough overview of engraving practices of various craftsmen. It includes detailed specs of characters, dimensions etc. 2. a thorough overview of a anonymous automated system, which must be antiquated by now. EDV means E(lektronischen) D(aten)v(erarbeitung), electronic data processing.

5 Technical manual

When LilyPond is run, it reads an input file which is parsed. During parsing, Music objects are created. This music is interpreted, which is done by contexts, that produce graphical objects. This section discusses details of these three concepts, and how they are glued together with the embedded Scheme interpreter.

5.1 Interpretation context

Interpretation contexts are objects that only exist during program run. During the interpretation phase (when `interpreting music` is printed on the standard output), the music expression in a `\score` block is interpreted in time order, the same order in which we hear and play the music. During this phase, the interpretation context holds the state for the current point within the music, for example:

- What notes are playing at this point?
- What symbols will be printed at this point?
- What is the current key signature, time signature, point within the measure, etc.?

Contexts are grouped hierarchically: A **Voice** context is contained in a **Staff** context (because a staff can contain multiple voices at any point), a **Staff** context is contained in **Score**, **StaffGroup**, or **ChoirStaff** context.

Contexts associated with sheet music output are called *notation contexts*, those for sound output are called *performance contexts*. The default definitions of the standard notation and performance contexts can be found in `'ly/engraver-init.ly'` and `'ly/performer-init.ly'`, respectively.

5.1.1 Creating contexts

Contexts for a music expression can be selected manually, using one of the following music expressions:

```
\new contexttype musicexpr
\context contexttype [= contextname] musicexpr
```

This means that *musicexpr* should be interpreted within a context of type *contexttype* (with name *contextname* if specified). If no such context exists, it will be created:

```
\score {
  \notes \relative c'' {
    c4 <<d4 \context Staff = "another" e4>> f
  }
}
```



In this example, the **c** and **d** are printed on the default staff. For the **e**, a context **Staff** called **another** is specified; since that does not exist, a new context is created. Within **another**, a (default) **Voice** context is created for the **e4**. A context is ended when all music referring it has finished, so after the third quarter, **another** is removed.

The `\new` construction creates a context with a generated, unique *contextname*. An expression with `\new` always leads to a new context. This is convenient for creating multiple staves, multiple lyric lines, etc.

When using automatic staff changes, automatic phrasing, etc., the context names have special meanings, so `\new` cannot be used.

5.1.2 Default contexts

Every top level music is interpreted by the **Score** context; in other words, you may think of `\score` working like

```
\score {
  \context Score music
}
```

Music expressions inherit their context from the enclosing music expression. Hence, it is not necessary to explicitly specify `\context` for most expressions. In the following example, only the sequential expression has an explicit context. The notes contained therein inherit the `goUp` context from the enclosing music expression.

```
\notes \context Voice = goUp { c'4 d' e' }
```



Second, contexts are created automatically to be able to interpret the music expressions. Consider the following example:

```
\score { \notes { c'4-( d' e'-) } }
```



The sequential music is interpreted by the **Score** context initially, but when a note is encountered, contexts are setup to accept that note. In this case, a **Thread**, **Voice**, and **Staff** context are created. The rest of the sequential music is also interpreted with the same **Thread**, **Voice**, and **Staff** context, putting the notes on the same staff, in the same voice.

5.1.3 Context properties

Contexts have properties. These properties are set from the `‘.ly’` file using the following expression:

```
\property contextname.propname = value
```

Sets the *propname* property of the context *contextname* to the specified Scheme expression *value*. Both *propname* and *contextname* are strings, which can often be written unquoted.

Properties that are set in one context are inherited by all of the contained contexts. This means that a property valid for the **Voice** context can be set in the **Score** context (for example) and thus take effect in all **Voice** contexts.

If you do not wish to specify the name of the context in the `\property`-expression itself, you can refer to the abstract context name, **Current**. The **Current** context is the latest used context. This will typically mean the **Thread** context, but you can force another context with the `\property`-command. Hence the expressions

```
\property contextname.propname = value
```

and

```
\context contextname
\property Current.propname = value
```

do the same thing. The main use for this is in predefined variables. This construction allows the specification of a property-setting without restriction to a specific context.

Properties can be unset using the following statement.

```
\property contextname.propname \unset
```

This removes the definition of *propname* in *contextname*. If *propname* was not defined in *contextname* (but was inherited from a higher context), then this has no effect.

Bugs

The syntax of `\unset` is asymmetric: `\property \unset` is not the inverse of `\property \set`.

5.1.4 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:get-context-property x 'currentBarNumber)))
```

5.1.5 Defining contexts

The most common way to create a new context definition is by extending an existing one. An existing context from the paper block is copied by referencing a context identifier:

```
\paper {
  \translator {
    context-identifier
  }
}
```

Every predefined context has a standard identifier. For example, the `Staff` context can be referred to as `\StaffContext`.

The context can then be modified by setting or changing properties, e.g.

```
\translator {
  \StaffContext
  Stem \set #'thickness = #2.0
  defaultBarType = #"||"
}
```

These assignments happen before interpretation starts, so a `\property` command will override any predefined settings.

Bugs

It is not possible to collect multiple property assignments in a variable, and apply to one `\translator` definition by referencing that variable.

5.1.6 Engravers and performers

Each context is composed of a number of building blocks, or plug-ins called engravers. An engraver is a specialized C++ class that is compiled into the executable. Typically, an engraver is responsible for one function: the `Slur_engraver` creates only `Slur` objects, and the `Skip_event_swallow_translator` only swallows (silently gobbles) `SkipEvents`.

An existing context definition can be changed by adding or removing an engraver. The syntax for these operations is

```
\consists engravername
\remove engravername
```

Here *engravername* is a string, the name of an engraver in the system. In the following example, the `Clef_engraver` is removed from the `Staff` context. The result is a staff without a clef, where the central C is at its default position, the center line:

```
\score {
  \notes {
    c'4 f'4
  }
  \paper {
    \translator {
      \StaffContext
      \remove Clef_engraver
    }
  }
}
```



A list of all engravers is in the internal documentation, see [Engravers](#).

5.1.7 Defining new contexts

It is also possible to define new contexts from scratch. To do this, you must define give the new context a name. In the following example, a very simple `Staff` context is created: one that will put note heads on a staff symbol.

```
\translator {
  \type "Engraver_group_engraver"
  \name "SimpleStaff"
  \alias "Staff"
  \consists "Staff_symbol_engraver"
  \consists "Note_head_engraver"
  \consistsend "Axis_group_engraver"
}
```

The argument of `\type` is the name for a special engraver that handles cooperation between simple engravers such as `Note_head_engraver` and `Staff_symbol_engraver`. This should always be `Engraver_group_engraver` (unless you are defining a `Score` context from scratch, in which case `Score_engraver` must be used).

The complete list of context modifiers is the following:

- `\alias alternate-name`: This specifies a different name. In the above example, `\property Staff.X = Y` will also work on `SimpleStaffs`.
- `\consistsend engravername`: Analogous to `\consists`, but makes sure that *engravername* is always added to the end of the list of engravers.

Engravers that group context objects into axis groups or alignments need to be at the end of the list. `\consistsend` insures that engravers stay at the end even if a user adds or removes engravers.

- `\accepts contextname`: This context can contains *contextname* contexts. The first `\accepts` is created as a default context when events (e.g. notes or rests) are encountered.
- `\denies`: The opposite of `\accepts`.
- `\name contextname`: This sets the type name of the context, e.g. `Staff`, `Voice`. If the name is not specified, the translator will not do anything.

5.2 Scheme integration

LilyPond internally uses `GUILE`, a Scheme-interpreter, to represent data throughout the whole program, and glue together different program modules. For advanced usage, it is sometimes necessary to access and program the Scheme interpreter.

Scheme is a full-blown programming language, from the LISP family. and a full discussion is outside the scope of this document. Interested readers are referred to the website <http://www.schemers.org/> for more information on Scheme.

The `GUILE` library for extension is documented at <http://www.gnu.org/software/guile>.

5.2.1 Inline Scheme

Scheme expressions can be entered in the input file by entering a hash-sign (`#`). The expression following the hash-sign is evaluated as Scheme. For example, the boolean value *true* is `#t` in Scheme, so for LilyPond *true* looks like `##t`, and can be used in property assignments:

```
\property Staff.autoBeaming = ##f
```

5.2.2 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)) }
```

5.2.3 Scheme datatypes

Scheme is used to glue together different program modules. To aid this glue function, many LilyPond specific object types can be passed as Scheme value.

The following list are all LilyPond specific types, that can exist during parsing:

Duration

Input

Moment

Music

Event In C++ terms, an **Event** is a subtype of **Music**. However, both have different functions in the syntax.

Music_output_def

Pitch

Score

Translator_def

During a run, transient objects are also created and destroyed.

Grob: short for ‘Graphical object’.

Scheme_hash_table

Music_iterator

Molecule: Device-independent page output object, including dimensions.

Syllable_group

Spring_smob

Translator: An object that produces audio objects or Grobs. It may be accessed with `\applyoutput`.

Font_metric: An object representing a font.

Many functions are defined to manipulate these data structures. They are all listed and documented in the internals manual, see **All scheme functions**.

5.2.4 Assignments

Variables allow objects to be assigned to names during the parse stage. To assign a variable, use

```
name=value
```

To refer to a variable, precede its name with a backslash: ‘`\name`’. *value* is any valid Scheme value or any of the input-types listed above. Variable assignments can appear at top level in the LilyPond file, but also in `\paper` blocks.

A variable can be created with any string for its name, but for accessing it in the LilyPond syntax, its name must consist of alphabetic characters only, and may not be a keyword of the syntax. There are no restrictions for naming and accessing variables in the Scheme interpreter,

The right hand side of a variable assignment is parsed completely before the assignment is done, so variables may be redefined in terms of its old value, e.g.

```
foo = \foo * 2.0
```

When a variable is referenced in LilyPond syntax, the information it points to is copied. For this reason, an variable reference must always be the first item in a block.

```
\paper {
  foo = 1.0
  \paperIdent % wrong and invalid
}

\paper {
  \paperIdent % correct
  foo = 1.0
}
```

5.3 Music storage format

Music in LilyPond is entered as music expressions. This section discusses different types of music expressions, and explains how information is stored internally. This internal storage is accessible through the Scheme interpreter, so music expressions may be manipulated using Scheme functions.

5.3.1 Music expressions

Notes, rests, lyric syllables are music expressions. Small music expressions may be combined to form larger ones, for example, by enclosing a list of expressions in `\sequential { }` or `<< >>`. In the following example, a compound expression is formed out of the quarter note `c` and a quarter note `d`:

```
\sequential { c4 d4 }
```

The two basic compound music expressions are simultaneous and sequential music:

```
\sequential { musicexprlist }
\simultaneous { musicexprlist }
```

For both, there is a shorthand:

```
{ musicexprlist }
```

for sequential and

```
<< musicexprlist >>
```

for simultaneous music. In principle, the way in which you nest sequential and simultaneous to produce music is not relevant. In the following example, three chords are expressed in two different ways:

```
\notes \context Voice {
  <<a c'>> <<b d'>> <<c' e'>>
  << { a b c' } { c' d' e' } >>
}
```



However, using `<<` and `>>` for entering chords leads to various peculiarities. For this reason, a special syntax for chords was introduced in version 1.7: `< >`.

Other compound music expressions include:

```
\repeat expr
\transpose from to expr
\apply func expr
\context type = id expr
\times fraction expr
```

5.3.2 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`.

5.3.3 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:get-mus-property` and `ly:set-mus-property!`.

An example is a function that reverses the order of elements in its argument:

```
 #(define (rev-music-1 m)
   (ly:set-mus-property! m 'elements (reverse
    (ly:get-mus-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 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:get-mus-property music 'elements))
        (child (ly:get-mus-property music 'element))
        (reversed (reverse elements)))

    ; set children
    (ly:set-mus-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'`.

5.4 Lexical details

Begins and ends with the `"` character. To include a `"` character in a string write `\`". Various other backslash sequences have special interpretations as in the C language. A string that contains no spaces can be written without the quotes. Strings can be concatenated with the `+` operator.

5.5 Output details

LilyPond's default output format is \TeX . Using the option '`-f`' (or '`--format`') other output formats can be selected also, but currently none of them work reliably.

At the beginning of the output file, various global parameters are defined. It also contains a large `\special` call to define PostScript routines to draw items not representable with \TeX , mainly slurs and ties. A DVI driver must be able to understand such embedded PostScript, or the output will be rendered incompletely.

Then the file '`lilyponddefs.tex`' is loaded to define the macros used in the code which follows. '`lilyponddefs.tex`' includes various other files, partially depending on the global parameters.

Now the music is output system by system (a 'system' consists of all 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.

The code produced by LilyPond should be run through \LaTeX , not plain \TeX .

Here an example how to embed a small LilyPond file `foo.ly` into running \LaTeX text without using the `lilypond-book` script (see Chapter 7 [lilypond-book manual], page 133):

```
\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 foo.ly
```

It is important to set the `indent` parameter to zero in the `\paper` block of '`foo.ly`'.

The call to `\lineskip` assures that there is enough vertical space between the LilyPond box and the surrounding text lines.

6 Invoking LilyPond

This chapter details the technicalities of running LilyPond.

6.1 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 `lilypond` 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.
If you use `lilypond-book` or your own wrapper files, do not use `\usepackage[[T1]{fontenc}` in the file header but do not forget `\usepackage[latin1]{inputenc}` if you use any other non-anglosaxian characters.
- `--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.

`--debug` Print even more information. This is useful when generating bugreports.

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

6.1.1 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/ly2dvi-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.

`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 “Lily was here, *version-number*”.

6.1.2 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 pagenumbers 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.

6.2 Invoking the lilypond binary

The LilyPond system consists of two parts: a binary executable, 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

6.3 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.

-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).

¹ The status of `GUILE` is not reset across invocations, so be careful not to change any system defaults from within Scheme.

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!**)

6.4 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
<pre>{/usr/share/lilypond/1.6.0,{!!/usr/share/texmf}}</pre> |
| GS_LIB | For processing PostScript output (obtained with <code>-f ps</code>) 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 <code>-f ps</code>) 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.

6.5 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, a bug-report should be filed.

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' }
```

6.6 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.

6.7 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://ftp.math.berkeley.edu/pub/Software/TeX/xdvi.tar.gz>.
Most T_EX 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.
There is also support for Emacs: lilypond-mode for Emacs provides keyword auto-completion, 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.
 - 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 T_EX 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`.

Bugs

When you convert the \TeX file to PostScript using `dvips`, it will complain about not finding `src:X:Y` files. These complaints are harmless, and can be ignored.

7 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 2.18 [Integrating text and music], page 29. For more information about LaTeX The not so Short Introduction to LaTeX (<http://www.ctan.org/tex-archive/info/lshort/english/>) provides a introction to using LaTeX.

7.1 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[26pt]
  c' d' e' f' g'2 g'
@end lilypond
```

produces



Then the short version:

```
@lilypond[11pt]{<c' e' g'>}
```

produces



`lilypond-book` knows the default margins and a few paper sizes. One of these commands should be in the beginning of the document:

- `@afourpaper`
- `@afourlatex`
- `@afourwide`
- `@smallbook`

`@pagesizes` are not yet supported.

When producing `texinfo`, `lilypond-book` also generates bitmaps of the music, so you can make a HTML document with embedded music.

7.2 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[26pt]{lilypond}
  c' d' e' f' g'2 g'2
\end{lilypond}
```

produces



Then the short version:

```
\lilypond[11pt]{<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 and the `multicols` environment from the `multicol` package 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.

7.3 Integrating HTML and music

Music is entered using

```
<lilypond relative1 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 relative1 verbatim>
  \key c \minor r8 c16 b c8 g as c16 b c8 d | g,4
</lilypond>
```



For inline pictures, use `<lilypond ... />` syntax, eg.

Some music in `<lilypond a b c/>` a line of text.

A special feature not (yet) available in other output formats, is the `<ly2dvifile>` tag, for example,

```
<ly2dvifile>trip.ly</ly2dvifile>
```

This runs ‘trip.ly’ through lilypond (see also Section 6.1 [Invoking lilypond], page 126), 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.

7.4 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 }
```

smallverbatim

works like **verbatim**, but in a smaller font.

intertext="text"

is used in conjunction with **verbatim** option: This puts *text* between the code and the music (without indentation).

filename="filename"

saves the LilyPond code to *filename*. By default, a hash value of the code is used.

11pt



13pt



16pt



20pt



**raggedright**

produces naturally spaced lines (i.e., `raggedright = ##t`); this works well for small music fragments.

multiline

is the opposite of `singleline`: it justifies and breaks lines.

linewidth=sizeunit

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=sizeunit

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=5cm,raggedright]{lilypond}
...
\end{lilypond}
```

noindent sets indentation of the first music system to zero. This option affects LilyPond, not the text layout.

notexidoc

prevents including `texidoc`. 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 } }
```

quote instructs lilypond-book to put LaTeX and Texinfo output into a quotation block.

printfilename

prints the file name before the music example. Useful in conjunction with `\lilypondfile`.

relative, relative N

uses relative octave mode. By default, notes are specified relative central C. The optional integer argument specifies the octave of the starting note, where the default 1 is central C.

7.5 Invoking lilypond-book

Running `lilypond-book` generates lots of small files that LilyPond will process. To avoid all that garbage in the source directory, it is advisable to change to a temporary directory first:

```
cd out && lilypond-book ../yourfile.tex
```

or to use the ‘`--outdir`’ command line option, and change to that director before running LaTeX or ‘`makeinfo`’:

```
lilypond-book --outdir=out yourfile.tex
cd out && latex yourfile.latex
```

For LaTeX input, the file to give to LaTeX has extension ‘`.latex`’. Texinfo input will be written to a file with extension ‘`.texi`’.

To add titling from the `\header` section of the files, add the following to the top of the LaTeX file:

```
\input titledefs.tex
\def\preLilyPondExample{\def\mustmakelilypondtitle{}}
```

For printing the LaTeX document, you will need to use `dvips`. For producing PS with scalable fonts, add the following options to the `dvips` command line:

```
-Ppdf -u +lilypond.map
```

`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` usually 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.

‘`--default-music-fontsize=szpt`’

Set the music font size to use if no `fontsize` is given as option.

‘`--force-music-fontsize=szpt`’

Force all music to use this `fontsize`, overriding options given to `\begin{lilypond}`.

‘`-I dir`’, ‘`--include=dir`’

Add `DIR` to the include path.

‘`-M`’, ‘`--dependencies`’

Write dependencies to ‘`filename.dep`’.

‘`--dep-prefix=pref`’

Prepend `pref` before each ‘`-M`’ dependency.

‘`-n`’, ‘`--no-lily`’

Generate the `.ly` files, but do not process them.

‘`--no-music`’

Strip all music from the input file.

‘`--no-pictures`’

Do not generate pictures when processing Texinfo.

‘`--outname=file`’

The name of LaTeX file to output. If this option is not given, the output name is derived from the input name.

‘`--outdir=dir`’

Place generated files in `dir`.

`--version`

Print version information.

`--help` Print a short help message.

7.6 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`. Using `\lilypondfile` also makes upgrading files (through `convert-ly`, see Section 8.1 [Invoking `convert-ly`], page 139) easier.

`lilypond-book` processes all music fragments in one big run. The state of the `GUILE` interpreter is not reset between fragments; this means that changes made to global `GUILE` definitions, e.g. done with `set!` or `set-cdr!`, can leak from one fragment into the next fragment.

8 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.

8.1 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.

8.2 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`
Quantise 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`
Quantise 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.

8.3 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.

8.4 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 \property Voice.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 (eg. 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.

8.5 Invoking `pmx2ly`

PMX is a MusiXTeX preprocessor written by Don Simons. More information on PMX is available from the following site:

`http://icking-music-archive.org/Misc/Music/musixtex/software/pmx/`.

`pmx2ly` converts from PMX to LilyPond input. The program is invoked as follows:

`pmx2ly [option]... pmx-file`

The following options are supported by `pmx2ly`:

```
-h,--help
    this help

-o,--output=FILE
    set output filename to FILE

-v,--version
    version information
```

Bugs

This script was updated last in September 2000, and then successfully converted the ‘barsant.pmx’ example from the PMX distribution. pmx2ly cannot parse more recent PMX files.

8.6 Invoking musedata2ly

Musedata (<http://www.musedata.org/>) is an electronic library of classical music scores, currently comprising 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 ref
```

Bugs

‘musedata2ly’ converts only a small subset of musedata.

8.7 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

,		\bold.....	107
'	33	\cadenzaOff.....	43
(\cadenzaOn.....	43
(begin * * * *)	47	\caesura.....	92
(end * * * *)	47	\clef.....	41
,		\consists.....	119
'	33	\context.....	116
.		\cr.....	59
'	35	\decr.....	59
/		\defaultAccidentals.....	49, 51
/	76	\deminutum.....	98
/+	76	\descendens.....	98
<		\divisioMaior.....	92
<<	122	\divisioMaxima.....	92
>		\divisioMinima.....	92
>>	122	\dorian.....	40
?		\dotsBoth.....	35
?	33	\dotsDown.....	35
[\dotsUp.....	35
[.....	46	\dynamic.....	107
]		\dynamicBoth.....	60
]	46	\dynamicDown.....	60
-		\dynamicUp.....	60
-	70	\f.....	59
\		\ff.....	59
\"!	59	\fff.....	59
\<	59	\ffff.....	59
\>	59	\finalis.....	92
\addlyrics.....	71	\flexa.....	98
\aeolian.....	40	\fontsize.....	108
\alternative.....	60	\forgetAccidentals.....	50, 51
\arpeggio.....	68, 69	\fp.....	59
\arpeggioBracket.....	69	\germanChords.....	78
\ascendens.....	98	\glissando.....	59
\auctum.....	98	\grace.....	57
\autoBeamOff.....	48	\header in LaTeX documents.....	137
\autoBeamOn.....	48	\hideNotes.....	33
\bar	43	\hideStaffSwitch.....	70
		\hspace.....	108
		\huge.....	106
		\inclinatum.....	98
		\ionian.....	40
		\italic.....	107
		\key.....	40
		\large.....	106
		\locrian.....	40
		\lydian.....	40
		\lyrics.....	70
		\magnify.....	108
		\major.....	40
		\mark.....	79
		\mf.....	59
		\minor.....	40
		\mixolydian.....	40
		\modernAccidentals.....	49, 51
		\modernCautionaries.....	50, 51
		\modernVoiceAccidentals.....	50, 51
		\modernVoiceCautionaries.....	50, 51
		\mp.....	59
		\musicglyph.....	107
		\newpage.....	112

Arpeggio	68
Arpeggio	69
ArpeggioEvent	69
articulation	18
articulations	54
Articulations	54
ASCII-art output	129
Assignments	121
aug	76
auto-knee-gap	47
autobeam	48
autoBeamSettings	47
AutoChangeMusic	67
automatic beam generation	48
Automatic beams	46
automatic beams, tuning	47
automatic part combining	82
Automatic staff changes	66
automatic syllable durations	71

B

balance	5
Banter	78
bar check	39
Bar check	39
Bar lines	43
bar lines at start of system	44
bar numbers	80
barCheckSynchronize	39
baritone clef	41
BarLine	44
barlines, putting symbols on	79
BarNumber	80
base-shortest-duration	110
bass clef	41
BassFigure	98
BassFigure	99
BassFigureEvent	99
Basso continuo	98
Beam	46, 62
beams and line breaks	47
beams, by hand	14
beams, kneed	47
beams, manual	46
beats per minute	53
between staves, distance	109
bibliographic information	23
bitmap	126
blackness	5
block comment	20
brace, vertical	79
bracket, vertical	79
brackets	54
BreakEvent	111
breaking lines	111
breaking pages	111
BreathingSign	53, 92
BreathingSignEvent	53, 92
broken arpeggio	68
bugreport	9
bugs	130

C

call trace	130
cautionary accidental	33
CCARH	142
ChoirStaff	116
choral score	71
choral tenor clef	41
chord entry	75
chord mode	75
chord names	22, 77
chordNameExceptions	78
ChordNames	77, 83
chordNameSeparator	78
chordNoteNamer	78
chordRootNamer	78
chords	19, 22, 77
Chords	75
Chords mode	75
chords, jazz	78
Clef	41
clefs	87
cluster	100
Cluster_spanner_engraver	100
ClusterNoteEvent	100
clusters	77
ClusterSpanner	100
ClusterSpannerBeacon	100
coda	55, 79
Coda Technology	140
command line options	128
comments	20
common-shortest-duration	110
Completion_heads_engraver	39
composer	23
concatenate	124
condensing rests	82
context definition	112, 118
context properties	117
context selection	116
Context-properties	25
Contexts	104
craftsmanship	4
crescendo	18, 59
CrescendoEvent	60
cross staff	69
cross staff stem	66
cross staff voice, manual	24
cue notes	105
Current	117
currentBarNumber	80
custodes	91
custos	91
Custos	91
Custos_engraver	91

D

decrescendo	18, 59
DecrescendoEvent	60
defaultBarType	44
dim	76
diminuendo	60
direction, of dynamics	60
distance between staves	109

divisio	92
divisiones	92
documents, adding music to	133
DotColumn	35
Dots	35
DoublePercentRepeat	63
downbow	55
drums	63
duration	34
DVI driver	12
DVI file	12
dvilj	12
dvips	12, 137
DynamicLineSpanner	60
dynamics	18
Dynamics	59
DynamicText	60

E

easy notation	37
editor	131
Emacs	131
Emacs mode	131
Engraved by LilyPond	23
engraver	118, 119
Engravers	119
engraving	3
enigma	140
entering notes	32
error	130
error messages	130
errors, message format	130
ETF	140
evaluating Scheme	120
exceptions, chord names	78
expanding repeats	61
expression	17
extender	70
extender line	21
ExtenderEvent	70
extending lilypond	9
extra-offset	26

F

fatal error	130
FDL, GNU Free Documentation License	160
fermata	55
fermata on multi-measure rest	82
fermatas	79, 100
FiguredBass	83
FiguredBass	98
FiguredBass	99
file searching	129
Finale	140
finalis	92
finding graphical objects	103
finger change	56
finger-interface	104
FingerEvent	56, 103
fingering	18, 55
Fingering	56, 103, 104
Fingering-engraver	104

flageolet	55
flags	89
FoldedRepeatedMusic	62
follow voice	69
followVoice	69
font	5
font magnification	106
font selection	106
font size	105
font size, setting	111
font size, texts	106
font style, for texts	107
font switching	106
font-interface	103
font-interface	106
font-style	106
foot marks	55
footer	127
foreign languages	9
four bar music	111
french clef	41
Frenched scores	83
Frenched staves	47

G

ghostscript	37, 130
Ghostscript	12
Glissando	59
GlissandoEvent	59
glyph size	105
grace notes	20, 57
GraceMusic	58, 123, 124
grand staff	79
GrandStaff	50, 79
graphical object	104
graphical object descriptions	103
Gregorian square neumes ligatures	94
grob	104
grob-interface	104
GS_FONTPATH	130
GS_LIB	130
GUILE	120
GVim	131

H

Hairpin	59, 60
Hal Leonard	37
header	127
Hiding staves	83
Horizontal_bracket_engraver	54
HorizontalBracket	54
html	133
HTML, music in	29
hufnagel	85
HyphenEvent	70
hyphens	70

I

idiom	9
indent	18
indent	111

index 9
 inheriting 117
 installing LilyPond 129
 instrument names 113
 InstrumentName 80
 internal documentation 9, 103
interscoreline 111
interscorelinefill 111
 invisible objects 26
 Invisible rest 34
 invoking dvips 137
 Invoking LilyPond 128
 item-interface 103

J

jargon 9
 jazz chords 78

K

KDE 131
 KDVI 131
 kerning 108
 Key signature 40
 key signature, setting 13
 KeyChangeEvent 41
keySignature 41
 KeySignature 41, 87
 kneed beams 47

L

LANG 130
 language 9
 larger 107
lastpagefill 111
 latex 133
 LaTeX, music in 29
 layers 45
 layout object 104
 lead sheet 22
 Lead sheets 22
 Ligature_bracket_engraver 93, 94
 LigatureBracket 93
 Ligatures 92
 Lily was here 23
 lilypond-book and titling 137
 lilypond-internals 9
 lilypond-mode for Emacs 131
 LILYPONDPREFIX 130
 line breaks 111
 line comment 20
 line-column-location 132
 line-location 131
linewidth 111
 LISP 120
 lowering text 107
lpr 12
 LyricCombineMusic 71
 LyricEvent 70
 lyrics 21, 48, 70
 Lyrics 21
 Lyrics 71, 83

lyrics and melodies 71
 LyricsVoice 71, 72, 83

M

m 76
 magnification 105
maj 76
 majorSevenSymbol 78
 manual staff switches 67
 marcato 55
 mark 79
 MarkEvent 79
 markup 106
 markup text 106
 Markup-functions 108
 master 4
 measure lines 43
 measure numbers 80
 measure repeats 63
 measure, partial 43
 Measure_grouping_engraver 42
 MeasureGrouping 42
 Medicaea, Editio 85
 melisma 21, 70
 mensural 85
 Mensural ligatures 93
 Mensural_ligature_engraver 93, 94
 meter 42
 metronome marking 53
 MetronomeChangeEvent 53
 mezzosoprano clef 41
 MIDI 23, 129, 139
 MIDI block 112
 modifiers, in chords 76
 mordent 55
 moving text 107
 multi measure rests 81
 MultiMeasureRest 82
 MultiMeasureRestEvent 82
 MultiMeasureRestMusicGroup 82
 MultiMeasureRestNumber 82
 MultiMeasureRestText 82
 MultiMeasureTextEvent 82
 multiple voices 24
 MUP 142
 Musedata 142
 Music classes 123
 Music entry 37
 music expression 17
 music expressions 122
 Music expressions 123
 Music properties 123
 Music Publisher 142
 Music-expressions 104
 musical symbols 5
 musicological analysis 54
 MuiXTeX 141

N

NEdit 131
 NewTieEvent 36
 Non-guitar tablatures 74

NonMusicalPaperColumn	111
NonMusicalPaperColumn	112
Note entry	32
note grouping bracket	54
note heads	86
note names, Dutch	32
Note specification	32
Note_heads_engraver	39
NoteCollision	45
NoteCollision	46
NoteColumn	46
NoteEvent	33, 123
NoteGroupingEvent	54
NoteHead	33, 103, 105
NoteSpacing	110
number of staff lines, setting	40

O

object description	101
object, layout	104
octavation	42
open	55
optical spacing	5
options, command line	128
organ pedal marks	55
ornaments	54, 57
ottava	42
OttavaSpanner	42
outline fonts	137
output format, setting	129
OverrideProperty	103

P

padding	27
padding	104
page breaks	111
page layout	111, 127
page size	112
paper file	111
paper size	112
papersize	112
parenthesized accidental	33
part combiner	82
PartCombineMusic	83
Partial	43
partial measure	20, 43
PDF	12, 126
PDFTeX output	129
Pedals	67
percent repeats	63
PercentRepeat	63
PercentRepeatedMusic	63
percussion	63
Petrucchi	85
phrasing brackets	54
phrasing marks	52
phrasing slurs	19, 52
phrasing, in lyrics	71
PhrasingSlur	53
PhrasingSlurEvent	53
PianoPedalBracket	68
PianoStaff	50, 66, 68, 69, 109

pickup	20
picture	126
Pitch names	32
pitch	32
pixmap	126
plug-in	119
PMX	141
point and click	131
polyphony	24, 45
portato	55
PostScript	12, 130
PostScript output	129
prall	55
prall, down	55
prall, up	55
prallmordent	55
prallprall	55
preview	126
preview image	135
printing chord names	77
Printing output	12
printing postscript	130
Programming error	130
properties	9
properties, context	117
properties, unsetting	118
'property-init.ly'	48
PropertySet	103
punctuation	70

Q

quarter tones	32
quotes, in lyrics	70

R

r	35
R	81
raising text	107
regular line breaks	111
regular rhythms	5
regular spacing	5
Rehearsal marks	79
RehearsalMark	79
Relative	37
relative octave specification	37
reminder accidental	33
removals, in chords	75
RemoveEmptyVerticalGroup	84
removing objects	26
repeat bars	43
repeatCommands	44
repeatCommands	61
RepeatedMusic	62, 123
repeats	60
RepeatSlash	63
reporting bugs	130
Rest	34, 87
RestCollision	46
RestEvent	34
rests	10, 87
Rests	34
Rests, multi measure	81

reverseturn 55
 RevertProperty 103
 root of chord 75

S

s 35
 SATB 71
 Scalable fonts 126
 Scheme 9, 120
 Scheme dump 129
 Scheme error 130
 Scheme, in-line code 120
 Score 37, 42, 51, 104, 116, 117
 screenshot 126
 Script 55
 script on multi-measure rest 82
 ScriptEvent 55
 scripts 54, 56
 search in manual 9
 search path 129
 segno 55, 79
 self-alignment-interface 103
 semi-flats, semi-sharps 32
 SeparatingGroupSpanner 110
 SeparationItem 110
 sequential music 122
 SequentialMusic 122
 SequentialMusic 123
 setting object properties 26
 shorten measures 43
 side-position-interface 103
 signature line 23
 Simons, Don 141
 Simultaneous music 122
 SimultaneousMusic 123
 size 105
 Sketch output 129
 Skip 34
 SkipEvent 34
 skipTypesetting 39
 slur 18
 Slur 52
 SlurEvent 52
 Slurs 51
 slurs versus ties 19
 smaller 107
 snippets 9
 Songs 21
 soprano clef 41
 sound 23
 Sound 112
 space between staves 109
 Space note 34
 spaces, in lyrics 70
spacing 110
 SpacingSpanner 109, 110
 SpacingSpanner, overriding properties 110
 SpanBar 44
 Square neumes ligatures 94
 staccatissimo 55
 staccato 18, 55
 Staff 40, 44, 50, 51, 54, 73, 80, 84, 91, 104, 110, 116, 139
 staff distance 109

staff group 79
 staff lines, setting number of 40
 staff lines, setting thickness of 40
 Staff notation 40
 staff order, with `\addlyrics` 71
 staff size, setting 111
 staff switch, manual 24, 67
 staff switching 69
 Staff, multiple 79
Staff.midiInstrument 113
 StaffGroup 44, 79, 80, 116
 staves per page 109
 StaffSpacing 110
 StaffSymbol 40
 StaffSymbol, using `\property` 40
 start of system 44
 Stem 35, 89, 101, 105
 stem, cross staff 66
stem-spacing-correction 110
stemLeftBeamCount 46
stemRightBeamCount 47
 StemTremolo 62
 stopped 55
 string 124
 StringNumberEvent 74
 subbass clef 41
 subdivideBeams 47
 subscript 56
 superscript 56
sus 76
 SustainPedal 67
 switches 128
 symbol size 105
 syntax coloring 131
 SystemStartBar 44
 SystemStartBrace 44
 SystemStartBracket 44

T

Tablatures basic 73
 TabStaff 73, 74
 TabVoice 73, 74
 tag line 23
 Tempo 53
 tenor clef 41
 tenuto 55
 terminology 9
texi 133
 texinfo 133
 Texinfo, music in 29
 TEXMF 130
 text markup 106
 text on multi-measure rest 82
 Text scripts 56
 Text spanners 53
 text-interface 103
 text-script-interface 103
textheight 111
 TextScript 56
 TextScriptEvent 56
 TextSpanEvent 54
 TextSpanner 54
 thickness of staff lines, setting 40

Thread 117
 Thread_devnull_engraver 83
 thumb marking 55
 thumbnail 126, 135
 tie 14
 Tie 35, 36
 TieEvent 36
 ties 35
 Time signature 42
 time signatures 90
 TimeScaledMusic 37
 TimeSignature 43, 90
 Timing_engraver 43
 titles 23, 127
 titling and lilypond-book 137
 titling in THML 135
 translating text 107
 translator definition 118
 transparent objects 26
 Transpose 81
 TransposedMusic 81
transposing 85
 transposition of pitches 81
 treble clef 41
 tremolo beams 62
 tremolo marks 62
 TremoloEvent 62
tremoloFlags 62
 trill 55
 triplets 20, 36
 tuning automatic beaming 47
 tuplet formatting 37
 TupletBracket 37
tupletNumberFormatFunction 37
 triplets 20, 36
 turn 55
 tweaking 103
 type1 fonts 137
 typeset text 106
 typography 3

U

UnfoldedRepeatedMusic 62
 UntransposableMusic 81

upbow 55
 upstep 20
 URL 9
 using the manual 9

V

varbaritone clef 41
 varcoda 55
 variables 9
 Vaticana, Editio 85
 VaticanaStaffContext 99
 VaticanaVoiceContext 99
 vertical spacing 109, 111
 VerticalAlignment 109
 Viewing music 12
 Vim 131
 violin clef 41
 Voice 45, 52
 Voice 59, 71, 72, 73, 93, 95, 100, 103, 110
 Voice 112
 Voice 116, 117, 139
Voice.autoBeaming 48
 Voice_devnull_engraver 83
Voice_engraver 83
 VoiceFollower 70
 voices, more – on a staff 24
 VoltaBracket 62
 VoltaRepeatedMusic 62

W

warning 130
 website 9
 whichBar 44
 White mensural ligatures 93
 whole rests for a full measure 82
 Writing parts 79

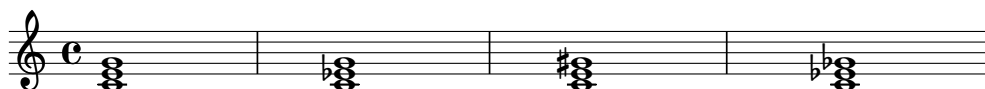
X

xdvi 12
 Xdvi 37, 131
XEDITOR 131
 XEmacs 131

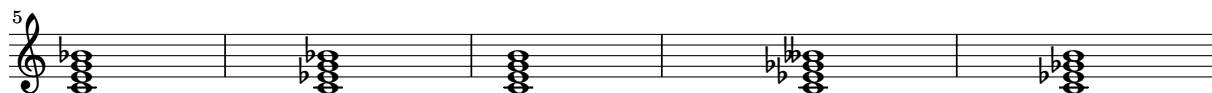
Appendix A Reference manual details

A.1 Chord name chart

Ignatzek (default)	C	Cm	C+	C°
Alternative	C	C ^{b3}	C ^{#5}	C ^{b3 b5}



Def	C ⁷	Cm ⁷	C ^Δ	C ^{o7}	Cm ^{Δ/b5}
Alt	C ⁷	C ^{7 b3}	C ^{#7}	C ^{b3 b5 b7}	C ^{b3 b5 #7}



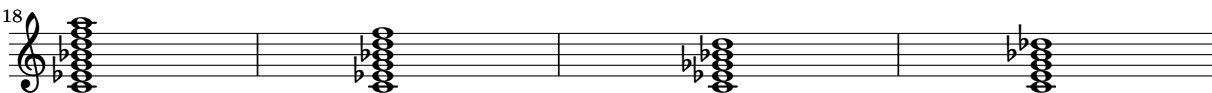
Def	C ^{7/#5}	Cm ^Δ	C ^{Δ/#5}	C ^φ
Alt	C ^{7 #5}	C ^{b3 #7}	C ^{#5 #7}	C ^{7 b3 b5}



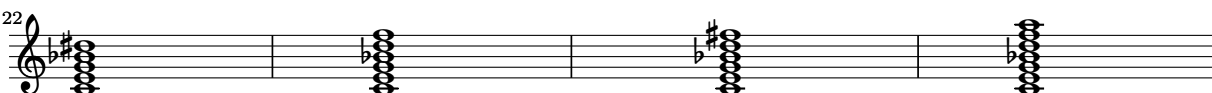
Def	C ⁶	Cm ⁶	C ⁹	Cm ⁹
Alt	C ⁶	C ^{b3 6}	C ⁹	C ^{9 b3}



Def	Cm ¹³	Cm ¹¹	Cm ^{7/b5/9}	C ^{7/b9}
Alt	C ^{13 b3}	C ^{11 b3}	C ^{9 b3 b5}	C ^{7 b9}



Def	C ^{7/#9}	C ¹¹	C ^{7/#11}	C ¹³
Alt	C ^{7 #9}	C ¹¹	C ^{9 #11}	C ¹³



Def	C ^{7/#11/b13}	C ^{7/#5/#9}	C ^{7/#9/#11}	C ^{7/b13}
Alt	C ^{9 #11 b13}	C ^{7 #5 #9}	C ^{7 #9 #11}	C ^{11 b13}



Def	$C^{7/b9/b13}$	$C^{7/\#11}$	$C^{\Delta/9}$	$C^{7/b13}$
Alt	$C^{11\ b9\ b13}$	$C^{9\ \#11}$	$C^{9\ \#7}$	$C^{11\ b13}$

Def	$C^{7/b9/b13}$	$C^{7/b9/13}$	$C^{\Delta/9}$	$C^{\Delta/13}$
Alt	$C^{11\ b9\ b13}$	$C^{13\ b9}$	$C^{9\ \#7}$	$C^{13\ \#7}$

Def	$C^{\Delta/\#11}$	$C^{7/b9/13}$	C^{sus4}	$C^{7/sus4}$
Alt	$C^{9\ \#7\ \#11}$	$C^{13\ b9}$	$C^{add4\ 5}$	$C^{add4\ 5\ 7}$

Def	$C^{9/sus4}$	C^{add9}	C^{madd11}
Alt	$C^{add4\ 5\ 7\ 9}$	C^{add9}	$C^{b3\ add11}$

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 section"	"sitar"
"church organ"	"synthbrass 1"	"banjo"
"reed organ"	"synthbrass 2"	"shamisen"
"accordion"	"soprano sax"	"koto"
"harmonica"	"alto sax"	"kalimba"
"concertina"	"tenor sax"	"bagpipe"
"acoustic guitar (nylon)"	"baritone sax"	"fiddle"

"acoustic guitar (steel)"	"oboe"	"shantai"
"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)"	"skakuhachi"	"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 3.17.5 [Text markup], page 106.





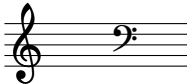





rests-0	rests-1	rests-0o
rests-1o	rests--3	rests--2
rests--1	rests-2	rests-2classical
rests-3	rests-4	rests-5
rests-6	rests-7	accidentals-2
accidentals-1	accidentals-3	accidentals-0
accidentals--2	accidentals--1	accidentals--4
accidentals--3	accidentals-4	accidentals-rightparen
accidentals-leftparen	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	noteheads-ledgerending	scripts-ufermata
scripts-dfermata	scripts-ushortfermata	scripts-dshortfermata
scripts-ulongfermata	scripts-dlongfermata	scripts-uverylongfermata
scripts-dverylongfermata	scripts-thumb	scripts-sforzato

scripts-staccato	scripts-ustaccatissimo	scripts-dstaccatissimo
scripts-tenuto	scripts-uportato	scripts-dportato
scripts-umarcato	scripts-dmarcato	scripts-open
scripts-stopped	scripts-upbow	scripts-downbow
scripts-reverseturn	scripts-turn	scripts-trill
scripts-upedalheel	scripts-dpedalheel	scripts-upedaltoe
scripts-dpedaltoe	scripts-flageolet	scripts-segno
scripts-coda	scripts-varcoda	scripts-rcomma
scripts-lcomma	scripts-rvarcomma	scripts-lvarcomma
scripts-arpeggio	scripts-trill-element	scripts-arpeggio-arrow--1
scripts-arpeggio-arrow-1	scripts-trilelement	scripts-prall
scripts-mordent	scripts-prallprall	scripts-prallmordent
scripts-upprall	scripts-downprall	scripts-upmordent
scripts-downmordent	scripts-lineprall	scripts-pralldown
scripts-prallup	scripts-caesura	flags-u3
flags-u4	flags-u5	flags-u6
flags-d3	flags-ugrace	flags-dgrace
flags-d4	flags-d5	flags-d6
flags-stem	flags-dstem	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-accSB
accordion-accBB	accordion-accOldEE	accordion-accOldEES
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-1neo_mensural	≡ noteheads--3neo_mensural
≡ noteheads--2neo_mensural	≡ noteheads--1neo_mensural	◇ noteheads-0neo_mensural
◇ noteheads-1neo_mensural	◆ noteheads-2neo_mensural	≡ noteheads-1mensural
≡ noteheads--3mensural	≡ noteheads--2mensural	≡ noteheads--1mensural
◇ noteheads-0mensural	◇ noteheads-1mensural	◆ noteheads-2mensural
■ noteheads-vaticana_punctum	◻ noteheads-vaticana_punctum_cavum	■ noteheads-vaticana_linea_punctum
◻ noteheads-vaticana_linea_punctum_cavum	◆ noteheads-vaticana_inclinatum	■ noteheads-vaticana_lpes
■ noteheads-vaticana_vlpes	■ noteheads-vaticana_upes	■ noteheads-vaticana_vupes
└ noteheads-vaticana_plica	└ noteheads-vaticana_epiphonus	└ noteheads-vaticana_vepiphonus
└ noteheads-vaticana_reverse_plica	└ noteheads-vaticana_inner_cephalicus	└ noteheads-vaticana_cephalicus
■ noteheads-vaticana_quilisma	└ noteheads-solesmes_incl_parvum	└ noteheads-solesmes_auct_asc
■ noteheads-solesmes_auct_desc	└ noteheads-solesmes_incl_auctum	└ noteheads-solesmes_stropha
└ noteheads-solesmes_stropha_aucta	■ noteheads-solesmes_oriscus	◆ noteheads-medicaea_inclinatum
■ noteheads-medicaea_punctum	┌ noteheads-medicaea_rvirga	┌ noteheads-medicaea_virga
◆ noteheads-hufnagel_punctum	└ noteheads-hufnagel_virga	■ 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-petrucchi_c1	└ clefs-petrucchi_c1_change
└ clefs-petrucchi_c2	└ clefs-petrucchi_c2_change	└ clefs-petrucchi_c3
└ clefs-petrucchi_c3_change	└ clefs-petrucchi_c4	└ clefs-petrucchi_c4_change
└ clefs-petrucchi_c5	└ clefs-petrucchi_c5_change	≡ clefs-mensural_c
≡ clefs-mensural_c_change	└ clefs-petrucchi_f	└ clefs-petrucchi_f_change

 clefs-mensural_f	 clefs-mensural_f_change	 clefs-mensural_g
 clefs-mensural_g_change	 clefs-petrucci_g	 clefs-petrucci_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		

Appendix B Cheat sheet

Syntax	Description	Example
<code>1 2 8 16</code>	durations	
<code>c4. c4..</code>	augmentation dots	
<code>c d e f g a b</code>	scale	
<code>fis bes</code>	alteration	
<code>\clef treble \clef bass</code>	clefs	
<code>\time 3/4 \time 4/4</code>	time signature	
<code>r4 r8</code>	rest	
<code>d ~ d</code>	tie	
<code>\key es \major</code>	key signature	
<code>note'</code>	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\s fz`

dynamics

`a\< b\!`

crescendo

`a\> b\!`

decrescendo

`< >`

chord



`\partialial 8`

upstep

`\times 2/3 {f g a}`

triplets

`\grace`

grace notes


`\lyrics { ... }`
`\context Lyrics`

entering lyrics
printing lyrics
`twin -- kle`

lyric hyphen



twin-kle

`\chords { c:dim f:maj7 }`

chords

`\context ChordNames`

printing chord names

C^o F^Δ`<<{e f} \{\c d}>>`

polyphony

`s4 s8 s16`

spacer rests

Appendix C GNU Free Documentation License

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file

format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to

the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgments”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

C.0.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being *list*”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.