LilyPond

The music typesetter

Learning Manual

The LilyPond development team

For LilyPond version 2.12.3

Table of Contents

Preface 1

1 Introduction 2

1.1 Background 2
Engraving 2
Automated engraving 3
What symbols to engrave? 4
Music representation 6
Example applications 7
1.2 About the documentation 8
About the Learning Manual 8
About the Music Glossary 8
About the Notation Reference 9
About the Application Usage 9
About the Snippet List 9
About the Internals Reference 10
Other documentation 10

2 Tutorial 11

2.1 First steps 11
2.1.1 Compiling a file 11
2.1.2 Simple notation 12
2.1.3 Working on input files 16
2.1.4 How to read the manual 17
2.2 Single staff notation 18
2.2.1 Accidentals and key signatures 18
2.2.2 Ties and slurs 19
2.2.3 Articulation and dynamics 20
2.2.4 Adding text 22
2.2.5 Automatic and manual beams 22
2.2.6 Advanced rhythmic commands 23
2.3 Multiple notes at once 24
2.3.1 Music expressions explained 24
2.3.2 Multiple staves 25
2.3.3 Staff groups 26
2.3.4 Combining notes into chords 27
2.3.5 Single staff polyphony 28
2.4 Songs 28
2.4.1 Setting simple songs 28
2.4.2 Aligning lyrics to a melody 29
2.4.3 Lyrics to multiple staves 32
2.5 Final touches 33
2.5.1 Organizing pieces with variables 33
2.5.2 Version number 34
2.5.3 Adding titles 34
2.5.4 Absolute note names 35
2.5.5 After the tutorial 36

3 Fundamental concepts ... 37

3.1 How LilyPond input files work ... 37
3.1.1 Introduction to the LilyPond file structure ... 37
3.1.2 Score is a (single) compound musical expression ... 39
3.1.3 Nesting music expressions ... 41
3.1.4 On the un-nestedness of brackets and ties ... 43
3.2 Voices contain music ... 44
3.2.1 I'm hearing Voices ... 44
3.2.2 Explicitly instantiating voices ... 48
3.2.3 Voices and vocals ... 52
3.3 Contexts and engravers ... 58
3.3.1 Contexts explained ... 58
3.3.2 Creating contexts ... 59
3.3.3 Engravers explained ... 61
3.3.4 Modifying context properties ... 62
3.3.5 Adding and removing engravers ... 66
3.4 Extending the templates ... 68
3.4.1 Soprano and cello ... 69
3.4.2 Four-part SATB vocal score ... 72
3.4.3 Building a score from scratch ... 76

4 Tweaking output ... 80

4.1 Tweaking basics ... 80
4.1.1 Introduction to tweaks ... 80
4.1.2 Objects and interfaces ... 80
4.1.3 Naming conventions of objects and properties ... 81
4.1.4 Tweaking methods ... 81
4.2 The Internals Reference manual ... 84
4.2.1 Properties of layout objects ... 84
4.2.2 Properties found in interfaces ... 88
4.2.3 Types of properties ... 90
4.3 Appearance of objects ... 91
4.3.1 Visibility and color of objects ... 91
4.3.2 Size of objects ... 95
4.3.3 Length and thickness of objects ... 98
4.4 Placement of objects ... 99
4.4.1 Automatic behavior ... 99
4.4.2 Within-staff objects ... 100
4.4.3 Outside-staff objects ... 103
4.5 Collisions of objects ... 109
4.5.1 Moving objects ... 109
4.5.2 Fixing overlapping notation ... 111
4.5.3 Real music example ... 116
4.6 Further tweaking ... 124
4.6.1 Other uses for tweaks ... 124
4.6.2 Using variables for tweaks ... 126
4.6.3 Other sources of information ... 127
4.6.4 Avoiding tweaks with slower processing ... 128
4.6.5 Advanced tweaks with Scheme ... 129

5 Working on LilyPond projects ... 131

5.1 Suggestions for writing LilyPond input files ... 131
5.1.1 General suggestions ... 131
5.1.2 Typesetting existing music ... 132
5.1.3 Large projects ... 132
5.1.4 Saving typing with variables and functions ... 133
5.1.5 Style sheets ... 134
5.2 When things don't work ... 138
5.2.1 Updating old input files ... 138
5.2.2 Troubleshooting (taking it all apart) ... 138
5.2.3 Minimal examples ... 139
5.3 Scores and parts ... 139

Appendix A Templates ... 141

A.1 Single staff ... 141
A.1.1 Notes only ... 141
A.1.2 Notes and lyrics ... 141
A.1.3 Notes and chords ... 142
A.1.4 Notes, lyrics, and chords. ... 143
A.2 Piano templates ... 143
A.2.1 Solo piano ... 143
A.2.2 Piano and melody with lyrics ... 144
A.2.3 Piano centered lyrics ... 145
A.2.4 Piano centered dynamics ... 146
A.3 String quartet ... 148
A.3.1 String quartet ... 148
A.3.2 String quartet parts ... 149
A.4 Vocal ensembles ... 152
A.4.1 SATB vocal score ... 152
A.4.2 SATB vocal score and automatic piano reduction ... 154
A.4.3 SATB with aligned contexts ... 156
A.5 Ancient notation templates ... 157
A.5.1 Transcription of mensural music ... 158
A.5.2 Gregorian transcription template ... 163
A.6 Jazz combo ... 164
A.7 lilypond-book templates ... 170
A.7.1 LaTeX ... 170
A.7.2 Texinfo ... 171
A.7.3 xelatex ... 171

Appendix B Scheme tutorial ... 173

B.1 Tweaking with Scheme ... 174

Appendix C GNU Free Documentation License ... 176

Appendix D LilyPond index ... 182

Preface

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

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

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

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

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

Han-Wen and Jan

Utrecht/Eindhoven, The Netherlands, July 2002.

1 Introduction

This chapter introduces readers to LilyPond and the documentation.

1.1 Background

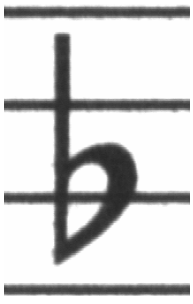
This section covers the overall goals and architecture of LilyPond.

Engraving

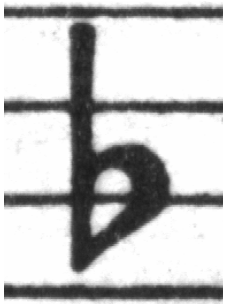
The art of music typography is called *(plate) engraving*. The term derives from the traditional process of music printing. Just a few decades ago, sheet music was made by cutting and stamping the music into a zinc or pewter plate in mirror image. The plate would be inked, the depressions caused by the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

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

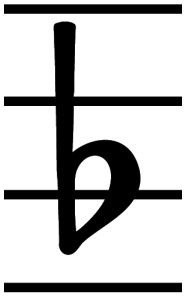
The images below illustrate the difference between traditional engraving and typical computer output, and the third picture shows how LilyPond mimics the traditional look. The left picture shows a scan of a flat symbol from an edition published in 2000. The center depicts a symbol from a hand-engraved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



Henle (2000)

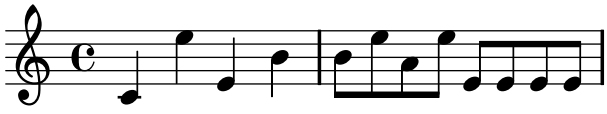


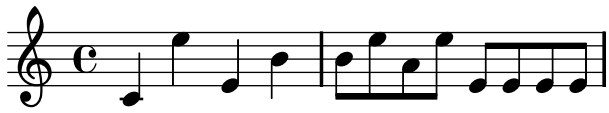
Bärenreiter (1950)



LilyPond Feta font (2003)

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





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

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

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

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

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

Automated engraving

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

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

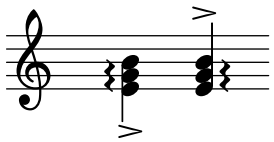
When we started, we wrote the LilyPond program entirely in the C++ programming language; the program's functionality was set in stone by the developers. That proved to be unsatisfactory for a number of reasons:

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

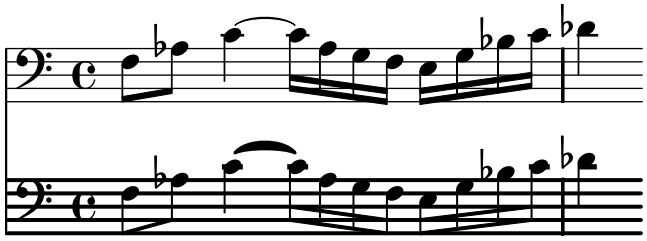
- Finally, we continually refine the formatting algorithms, so we need a flexible approach to rules. The C++ language forces a certain method of grouping rules that do not match well with how music notation works.

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

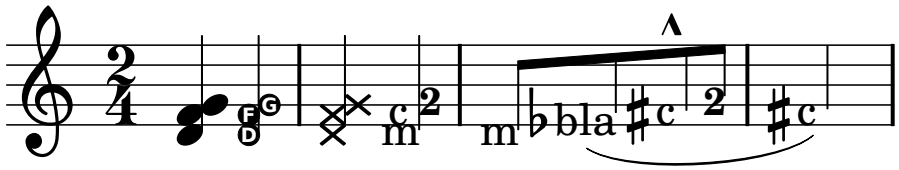
Scheme variables control layout decisions. For example, many graphical objects have a direction variable that encodes the choice between up and down (or left and right). Here you see two chords, with accents and arpeggios. In the first chord, the graphical objects have all directions down (or left).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



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



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



What symbols to engrave?

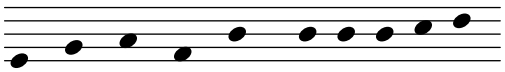
The formatting process decides where to place symbols. However, this can only be done once it is decided *what* symbols should be printed, in other words what notation to use.

Common music notation is a system of recording music that has evolved over the past 1000 years. The form that is now in common use dates from the early renaissance. Although the basic form (i.e., note heads on a 5-line staff) has not changed, the details still evolve to express the innovations of contemporary notation. Hence, it encompasses some 500 years of music. Its applications range from monophonic melodies to monstrous counterpoints for large orchestras.

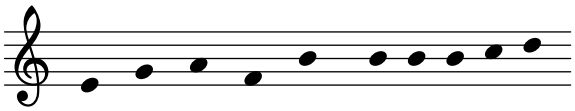
How can we get a grip on such a many-headed beast, and force it into the confines of a computer program? Our solution is to break up the problem of notation (as opposed to engraving, i.e., typography) into digestible and programmable chunks: every type of symbol is handled by a separate module, a so-called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

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

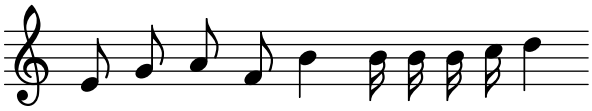
Then a `Staff_symbol_engraver` adds the staff



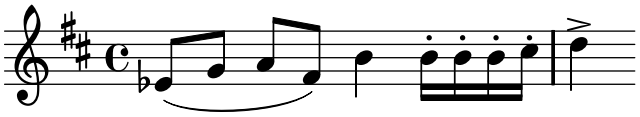
the `Clef_engraver` defines a reference point for the staff



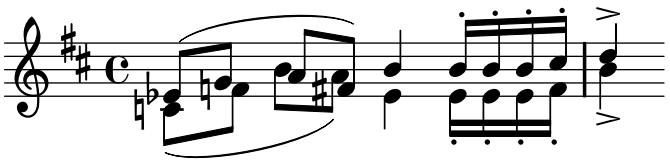
and the `Stem_engraver` adds stems.



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



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

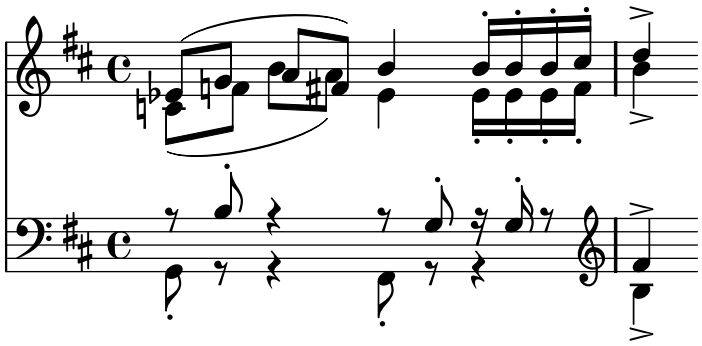


In this situation, the accidentals and staff are shared, but the stems, slurs, beams, etc., are private to each voice. Hence, engravers should be grouped. The engravers for note heads, stems,

slurs, etc., go into a group called ‘Voice context,’ while the engravers for key, accidental, bar, etc., go into a group called ‘Staff context.’ In the case of polyphony, a single Staff context contains more than one Voice context. Similarly, multiple Staff contexts can be put into a single Score context. The Score context is the top level notation context.

See also

Internals Reference: Section “Contexts” in *Internals Reference*.



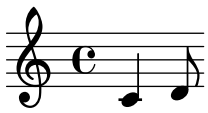
Music representation

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

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

```
{
  c'4 d'8
}
```

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



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

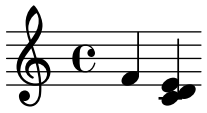
```
f4
```

Simultaneous notes can be constructed by enclosing them with `<<` and `>>`:

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

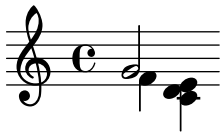
This expression is put in sequence by enclosing it in curly braces `{ ... }`:

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



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

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

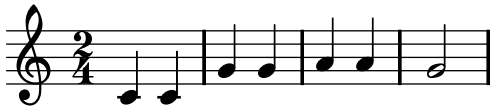


Such recursive structures can be specified neatly and formally in a context-free grammar. The parsing code is also generated from this grammar. In other words, the syntax of LilyPond is clearly and unambiguously defined.

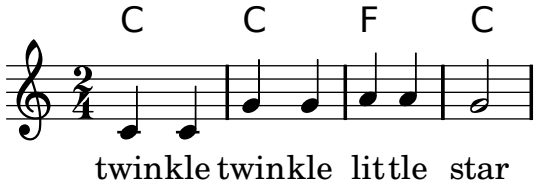
User-interfaces and syntax are what people see and deal with most. They are partly a matter of taste, and also subject of much discussion. Although discussions on taste do have their merit, they are not very productive. In the larger picture of LilyPond, the importance of input syntax is small: inventing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

Example applications

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



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

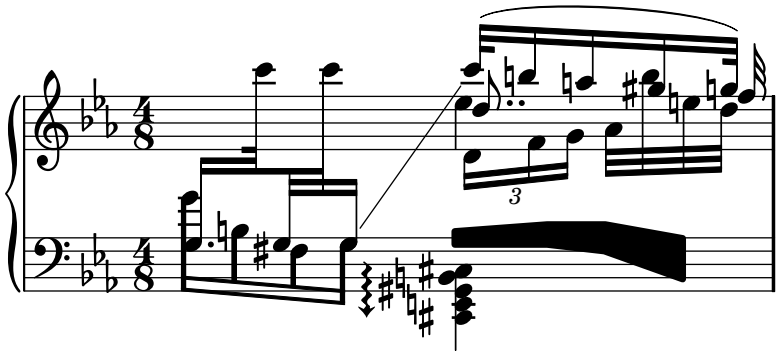


Polyphonic notation and piano music can also be printed. The following example combines some more exotic constructs.

Screech and boink

Random complex notation

Han-Wen Nienhuys



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

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

1.2 About the documentation

This section explains the different portions of the documentation.

About the Learning Manual

This book explains how to begin learning LilyPond, as well as explaining some key concepts in easy terms. You should read these chapters in a linear fashion.

There is a paragraph **See also** at the end of each section, which contains cross-references to other sections: you should not follow these cross-references at first reading; when you have read all of the Learning Manual, you may want to read some sections again and follow cross-references for further reading.

- Chapter 1 [Introduction], page 2: explains the background and overall goal of LilyPond.
- Chapter 2 [Tutorial], page 11: gives a gentle introduction to typesetting music. First time users should start here.
- Chapter 3 [Fundamental concepts], page 37: explains some general concepts about the LilyPond file format. If you are not certain where to place a command, read this chapter!
- Chapter 4 [Tweaking output], page 80: shows how to change the default engraving that LilyPond produces.
- Chapter 5 [Working on LilyPond projects], page 131: discusses practical uses of LilyPond and how to avoid some common problems. Read this before undertaking large projects!

The Learning Manual also contains appendices which are not part of the recommended linear reading. They may be useful for later viewing:

- Appendix A [Templates], page 141: shows ready-made templates of LilyPond pieces. Just cut and paste a template into a file, add notes, and you're done!
- Appendix B [Scheme tutorial], page 173: presents a short introduction to Scheme, the programming language that music functions use. This is material for advanced tweaks; many users never touch Scheme at all.

About the Music Glossary

Section "Music glossary" in *Music Glossary* this explains musical terms, and includes translations to various languages. If you are not familiar with music notation or music terminology (especially if you are a non-native English speaker), it is highly advisable to consult the glossary.

About the Notation Reference

This book explains all the LilyPond commands which produce notation. It assumes that readers are familiar with the concepts in the Learning Manual.

- Section "Musical notation" in *Notation Reference*: discusses topics grouped by notation construct. This section gives details about basic notation that will be useful in almost any notation project.
- Section "Specialist notation" in *Notation Reference*: discusses topics grouped by notation construct. This section gives details about special notation that will only be useful for particular instrument (or vocal) groups.
- Section "General input and output" in *Notation Reference*: discusses general information about LilyPond input files and controlling output.
- Section "Spacing issues" in *Notation Reference*: discusses issues which affect the global output, such as selecting paper size or specifying page breaks.
- Section "Changing defaults" in *Notation Reference*: explains how to tweak LilyPond to produce exactly the notation you want.
- Section "Interfaces for programmers" in *Notation Reference*: explains how to create music functions with scheme.

The Notation Reference also contains appendices with useful reference charts.

- Section "Literature list" in *Notation Reference*: contains a set of useful reference books for those who wish to know more on notation and engraving.
- Section "Notation manual tables" in *Notation Reference*: are a set of tables showing the chord names, MIDI instruments, a list of color names, and the Feta font.
- Section "Cheat sheet" in *Notation Reference*: is a handy reference of the most common LilyPond commands.
- Section "LilyPond command index" in *Notation Reference*: an index of all LilyPond `\commands`.
- Section "LilyPond index" in *Notation Reference*: a complete index.

About the Application Usage

This book explains how to execute the programs and how to integrate LilyPond notation with other programs.

- Section "Install" in *Application Usage*: explains how to install LilyPond, including compilation if desired.
- Section "Setup" in *Application Usage*: describes how to configure your computer for optimum LilyPond usage, such as using special environments for certain text editors.
- Section "Running LilyPond" in *Application Usage*: shows how to run LilyPond and its helper programs. In addition, this section explains how to upgrade input files from previous versions of LilyPond.
- Section "LilyPond-book" in *Application Usage*: explains the details behind creating documents with in-line music examples, like this manual.
- Section "Converting from other formats" in *Application Usage*: 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.

About the Snippet List

Section "LilyPond Snippet List" in *Snippets*: this shows a selected set of LilyPond snippets from the LilyPond Snippet Repository (LSR). All the snippets are in the public domain.

Please note that this document is not an exact subset of LSR. LSR is running a stable LilyPond version, so any snippet which demonstrates new features of a development version must be added separately. These are stored in '`input/new/`' in the LilyPond source tree.

The list of snippets for each subsection of the Notation Reference are also linked from the **See also** portion.

About the Internals Reference

Section "Internals Reference" in *Internals Reference*: this is a set of heavily cross linked HTML pages which document the nitty-gritty details of each and every LilyPond class, object, and function. It is produced directly from the formatting definitions in the source code.

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

Other documentation

There are a number of other sources of information which may be very valuable.

- News: this is a summary of important changes and new features in LilyPond since the previous version.
- The lilypond-user mailist archives: this is a collection of previous emails sent to the user list. Many questions have been asked multiple times; there is a very good chance that if you have a question, the answer might be found in these archives.
- The lilypond-devel mailist archives: this is a collection of previous emails sent to the developer's list. The discussion here is more technical; if you have an advanced question about lilypond internals, the answer might be in these archives.
- Embedded music fragments: in all HTML documents that have music fragments embedded, the exact LilyPond input that was used to produce that image can be viewed by clicking the image.
- Init files: 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/lsr/dirname/bla.ly`' may refer to the file '`lilypond2.x.y/input/lsr/dirname/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/`'. For more details, see Section 4.6.3 [Other sources of information], page 127.

2 Tutorial

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

2.1 First steps

This section gives a basic introduction to working with LilyPond.

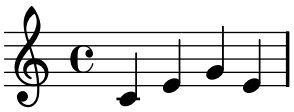
2.1.1 Compiling a file

"Compiling" is the term used for processing an input file in LilyPond format to produce a file which can be printed and (optionally) a MIDI file which can be played. LilyPond input files are simple text files. The first example shows what a simple input file looks like.

To create sheet music, we write an input file that specifies the notation. For example, if we write:

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

the result looks like this:



> **Note:** Notes and lyrics in LilyPond input must always be surrounded by `{` **curly braces** `}`. The braces should also be surrounded by a space unless they are at the beginning or end of a line to avoid ambiguities. The braces may be omitted in some examples in this manual, but don't forget them in your own music! For more information about the display of examples in the manual, see Section 2.1.4 [How to read the manual], page 17.

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

Entering music and viewing output

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

Note that there are several other text editors available with better support for LilyPond. For more information, see Section "Text editor support" in *Application Usage*.

> **Note:** The first time you ever run LilyPond, it may take a minute or two because all of the system fonts have to be analyzed first. After this, LilyPond will be much faster!

MacOS X

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

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

Windows

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

To edit an existing ‘`.ly`’ file, right-click on it and select “Edit source”. To get an empty file to start from, run the editor as described above and use “New” in the “File” menu, or right-click on the desktop and select “New..Text Document”, change its name to a name of your choice and change the file extension to `.ly`. Double-click the icon to type in your LilyPond source code as before.

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

UNIX

Create a text file called ‘`test.ly`’ and enter:

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

To process ‘`test.ly`’, proceed as follows:

```
lilypond test.ly
```

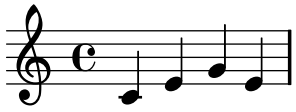
You will see something resembling:

```
lilypond test.ly
GNU LilyPond 2.12.3

Processing `test.ly'
Parsing...
Interpreting music...
Preprocessing graphical objects...
Finding the ideal number of pages...
Fitting music on 1 page...
Drawing systems...
Layout output to `test.ps'...
Converting to `test.pdf'...
```

2.1.2 Simple notation

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

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



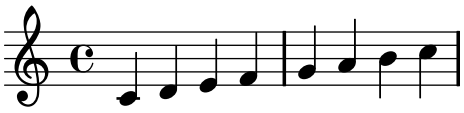
This behavior may be altered, but in most cases these automatic values are useful.

Pitches

Music Glossary: Section “pitch” in *Music Glossary*, Section “interval” in *Music Glossary*, Section “scale” in *Music Glossary*, Section “middle C” in *Music Glossary*, Section “octave” in *Music Glossary*, Section “accidental” in *Music Glossary*.

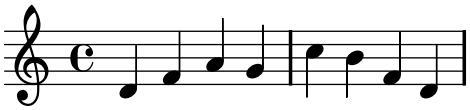
The easiest way to enter notes is by using `\relative` mode. In this mode, the octave is chosen automatically by assuming the following note is always to be placed closest to the previous note, i.e., it is to be placed in the octave which is within three staff spaces of the previous note. We begin by entering the most elementary piece of music, a *scale*, in which every note is within just one staff space of the previous note.

```
% set the starting point to middle C
\relative c' {
  c d e f
  g a b c
}
```



The initial note is *middle C*. Each successive note is placed closest to the previous note – in other words, the first `c` is the closest C to middle C. This is followed by the closest D to the previous note. We can create melodies which have larger intervals, still using only `\relative` mode:

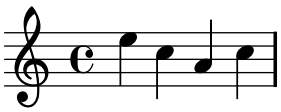
```
\relative c' {
  d f a g
  c b f d
}
```



It is not necessary for the first note of the melody to start on the note which specifies the starting pitch. In the previous example, the first note – the `d` – is the closest D to middle C.

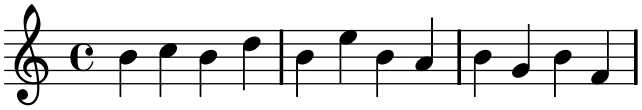
By adding (or removing) quotes `'` or commas `,` from the `\relative c' {` command, we can change the starting octave:

```
% one octave above middle C
\relative c'' {
  e c a c
}
```



Relative mode can be confusing initially, but is the easiest way to enter most melodies. Let us see how this relative calculation works in practice. Starting from a B, which is on the middle line in a treble clef, you can reach a C, D and E within 3 staff spaces going up, and an A, G and F within 3 staff spaces going down. So if the note following a B is a C, D or E it will be assumed to be above the B, and an A, G or F will be assumed to be below.

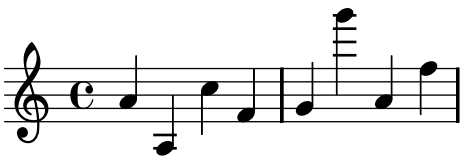
```
\relative c'' {
  b c  % c is 1 staff space up, so is the c above
  b d  % d is 2 up or 5 down, so is the d above
  b e  % e is 3 up or 4 down, so is the e above
  b a  % a is 6 up or 1 down, so is the a below
  b g  % g is 5 up or 2 down, so is the g below
  b f  % f is 4 up or 3 down, so is the f below
}
```



Exactly the same happens even when any of these notes are sharpened or flattened. *Accidentals* are **totally ignored** in the calculation of relative position. Precisely the same staff space counting is done from a note at any other position on the staff.

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

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



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

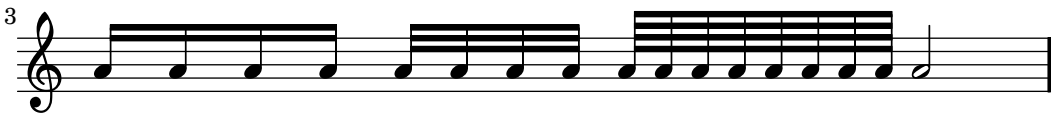
Durations (rhythms)

Music Glossary: Section “beam” in *Music Glossary*, Section “duration” in *Music Glossary*, Section “whole note” in *Music Glossary*, Section “half note” in *Music Glossary*, Section “quarter note” in *Music Glossary*, Section “dotted note” in *Music Glossary*.

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

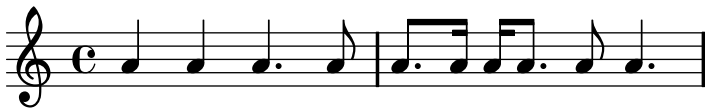
If you do not specify a duration, the previous duration is used for the next note. The duration of the first note defaults to a quarter.

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



To create *dotted notes*, add a dot `.` to the duration number. The duration of a dotted note must be stated explicitly (i.e., with a number).

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

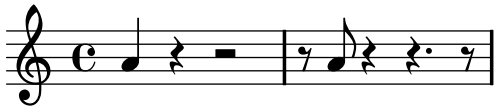


Rests

Music Glossary: Section "rest" in *Music Glossary*.

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

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

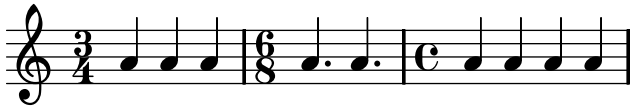


Time signature

Music Glossary: Section "time signature" in *Music Glossary*.

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

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



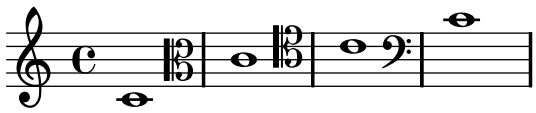
Clef

Music Glossary: Section "clef" in *Music Glossary*.

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

```
\relative c' {
  \clef treble
  c1
  \clef alto
  c1
```

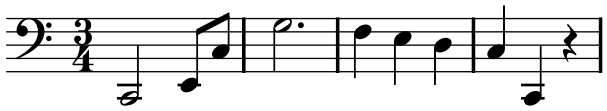
```
  \clef tenor
  c1
  \clef bass
  c1
}
```



All together

Here is a small example showing all these elements together:

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



See also

Notation Reference: Section "Writing pitches" in *Notation Reference*, Section "Writing rhythms" in *Notation Reference*, Section "Writing rests" in *Notation Reference*, Section "Time signature" in *Notation Reference*, Section "Clef" in *Notation Reference*.

2.1.3 Working on input files

LilyPond input files are similar to source files in many common programming languages. They are case sensitive, and white-space is generally ignored. Expressions are formed with curly braces `{ }`, and comments are denoted with `%` or `%{ ... %}`.

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

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

  ```
  { c                    d
                  e   }
  ```

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

  ```
  {
    c d e
  }
  ```

- **Expressions**: every piece of LilyPond input needs to have `{` **curly braces** `}` placed around the input. These braces tell LilyPond that the input is a single music expression, just like

parentheses `()` in mathematics. The braces should be surrounded by a space unless they are at the beginning or end of a line to avoid ambiguities.

A LilyPond command followed by a simple expression in braces (such as `\relative { }`) also counts as a single music expression.

- **Comments**: a comment is a remark for the human reader of the music input; it is ignored while parsing, so it has no effect on the printed output. There are two types of comments. The percent symbol `%` introduces a line comment; anything after `%` on that line is ignored. By convention, a line comment is placed *above* the code it refers to.

```
a4 a a a
% this comment refers to the Bs
b2 b
```

A block comment marks a whole section of music input as a comment. Anything that is enclosed in `%{` and `%}` is ignored. However, block comments do not 'nest'. This means that you cannot place a block comment inside another block comment. If you try, the first `%}` will terminate *both* block comments. The following fragment shows possible uses for comments:

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

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

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

2.1.4 How to read the manual

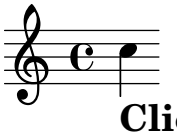
LilyPond input must be surrounded by `{ }` marks or a `\relative c'' { ... }`, as we saw in Section 2.1.3 [Working on input files], page 16. For the rest of this manual, most examples will omit this. To replicate the examples, you may copy and paste the displayed input, but you **must** add the `\relative c'' { }` like this:

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

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

Clickable examples

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. Try it on this image:



Click here.

By cutting and pasting everything in the "ly snippet" section, you have a starting template for experiments. To see exactly the same output (line-width and all), copy everything from "Start cut-&-pastable section" to the bottom of the file.

See also

There are more tips for constructing input files in Section 5.1 [Suggestions for writing LilyPond input files], page 131. But it might be best to read through the rest of the tutorial first.

2.2 Single staff notation

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

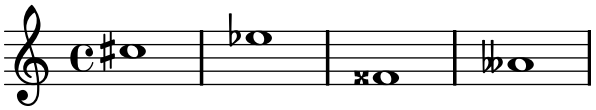
2.2.1 Accidentals and key signatures

Accidentals

Music Glossary: Section "sharp" in *Music Glossary*, Section "flat" in *Music Glossary*, Section "double sharp" in *Music Glossary*, Section "double flat" in *Music Glossary*, Section "accidental" in *Music Glossary*.

A *sharp* pitch is made by adding `is` to the name, and a *flat* pitch by adding `es`. As you might expect, a *double sharp* or *double flat* is made by adding `isis` or `eses`. This syntax is derived from note naming conventions in Nordic and Germanic languages, like German and Dutch. To use other names for *accidentals*, see Section "Note names in other languages" in *Notation Reference*.

```
cis1 ees fisis, aeses
```

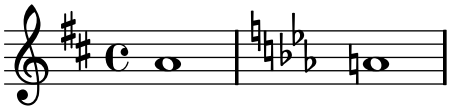


Key signatures

Music Glossary: Section "key signature" in *Music Glossary*, Section "major" in *Music Glossary*, Section "minor" in *Music Glossary*.

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

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



Warning: key signatures and pitches

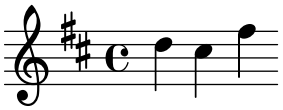
Music Glossary: Section "accidental" in *Music Glossary*, Section "key signature" in *Music Glossary*, Section "pitch" in *Music Glossary*, Section "flat" in *Music Glossary*, Section "natural" in *Music Glossary*, Section "sharp" in *Music Glossary*, Section "transposition" in *Music Glossary*.

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

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

In this example:

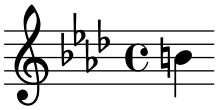
```
\key d \major
d cis fis
```



No note has a printed accidental, but you must still add `is` and type `cis` and `fis` in the input file.

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

```
\key aes \major
b
```



Adding all alterations explicitly might require a little more effort when typing, but the advantage is that *transposing* is easier, and accidentals can be printed according to different conventions. For some examples how accidentals can be printed according to different rules, see Section "Automatic accidentals" in *Notation Reference*.

See also

Notation Reference: Section "Note names in other languages" in *Notation Reference*, Section "Accidentals" in *Notation Reference*, Section "Automatic accidentals" in *Notation Reference*, Section "Key signature" in *Notation Reference*.

Music Glossary: Section "Pitch names" in *Music Glossary*.

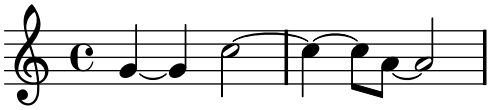
2.2.2 Ties and slurs

Ties

Music Glossary: Section "tie" in *Music Glossary*.

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

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

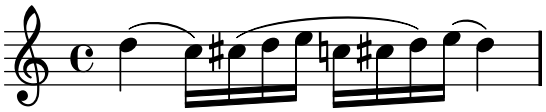


Slurs

Music Glossary: Section "slur" in *Music Glossary*.

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

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

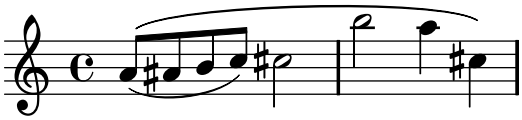


Phrasing slurs

Music Glossary: Section "slur" in *Music Glossary*, Section "phrasing" in *Music Glossary*.

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

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

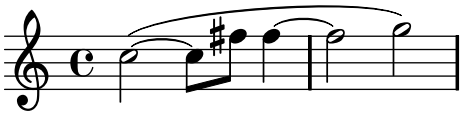


Warnings: slurs vs. ties

Music Glossary: Section "articulation" in *Music Glossary*, Section "slur" in *Music Glossary*, Section "tie" in *Music Glossary*.

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

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



See also

Notation Reference: Section "Ties" in *Notation Reference*, Section "Slurs" in *Notation Reference*, Section "Phrasing slurs" in *Notation Reference*.

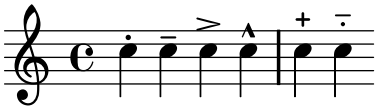
2.2.3 Articulation and dynamics

Articulations

Music Glossary: Section "articulation" in *Music Glossary*.

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

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

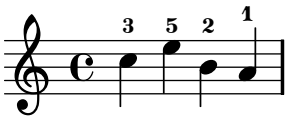


Fingerings

Music Glossary: Section “fingering” in *Music Glossary*.

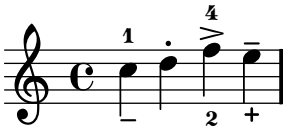
Similarly, *fingering* indications can be added to a note using a dash (`-`) and the digit to be printed:

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



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

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

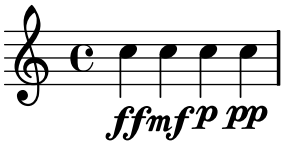


Dynamics

Music Glossary: Section “dynamics” in *Music Glossary*, Section “crescendo” in *Music Glossary*, Section “decrescendo” in *Music Glossary*.

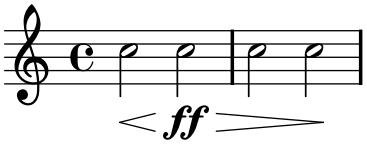
Dynamic signs are made by adding the markings (with a backslash) to the note:

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



Crescendi and *decrescendi* are started with the commands `\<` and `\>`. The next dynamics sign, for example `\f`, will end the (de)crescendo, or the command `\!` can be used:

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



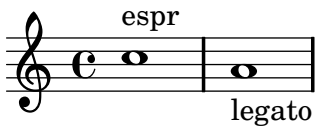
See also

Notation Reference: Section “Articulations and ornamentations” in *Notation Reference*, Section “Fingering instructions” in *Notation Reference*, Section “Dynamics” in *Notation Reference*.

2.2.4 Adding text

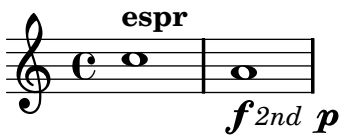
Text may be added to your scores:

```
c1^"espr" a_"legato"
```



Extra formatting may be added with the `\markup` command:

```
c1^\markup{ \bold espr}
a1_\markup{
  \dynamic f \italic \small { 2nd } \hspace #0.1 \dynamic p
}
```



See also

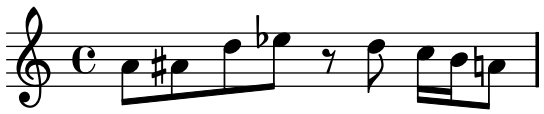
Notation Reference: Section "Writing text" in *Notation Reference*.

2.2.5 Automatic and manual beams

Music Glossary: Section "beam" in *Music Glossary*.

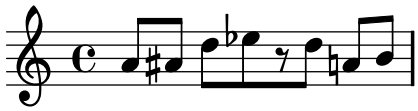
All *beams* are drawn automatically:

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



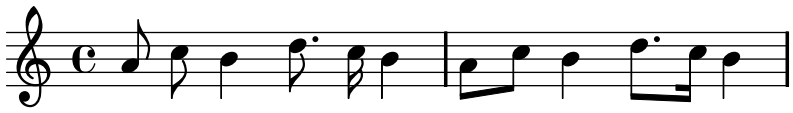
If you do not like the automatic beams, they may be overridden manually. To correct just an occasional beam mark the first note to be beamed with `[` and the last one with `]`.

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



If you want to turn off automatic beaming entirely or for an extended section of music, use the command `\autoBeamOff` to turn off automatic beaming and `\autoBeamOn` to turn it on again.

```
\autoBeamOff
a8 c b4 d8. c16 b4
\autoBeamOn
a8 c b4 d8. c16 b4
```



See also

Notation Reference: Section "Automatic beams" in *Notation Reference*, Section "Manual beams" in *Notation Reference*.

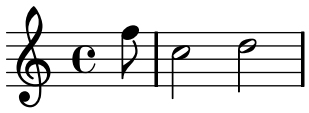
2.2.6 Advanced rhythmic commands

Partial measure

Music Glossary: Section "anacrusis" in *Music Glossary*.

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

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

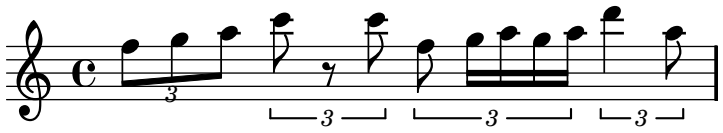


Tuplets

Music Glossary: Section "note value" in *Music Glossary*, Section "triplet" in *Music Glossary*.

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

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

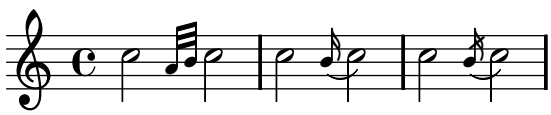


Grace notes

Music Glossary: Section "grace notes" in *Music Glossary*, Section "acciaccatura" in *Music Glossary*, Section "appoggiatura" in *Music Glossary*.

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

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



See also

Notation Reference: Section "Grace notes" in *Notation Reference*, Section "Tuplets" in *Notation Reference*, Section "Upbeats" in *Notation Reference*.

2.3 Multiple notes at once

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

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

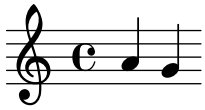
2.3.1 Music expressions explained

In LilyPond input files, music is represented by *music expressions*. A single note is a music expression:

```
a4
```

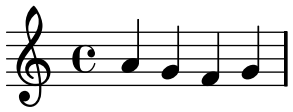
Enclosing a note in braces creates a *compound music expression*. Here we have created a compound music expression with two notes:

```
{ a4 g4 }
```



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

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



Analogy: mathematical expressions

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

```
1

1 + 2

(1 + 2) * 3

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

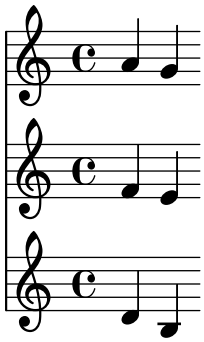
This is a sequence of expressions, where each expression is contained in the next (larger) one. The simplest expressions are numbers, and larger ones are made by combining expressions with operators (like `+`, `*` and `/`) and parentheses. Like mathematical expressions, music expressions can be nested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

Simultaneous music expressions: multiple staves

Music Glossary: Section "polyphony" in *Music Glossary*.

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

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



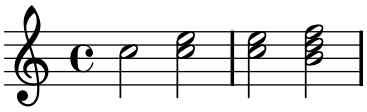
Note that we have indented each level of the input with a different amount of space. LilyPond does not care how much (or little) space there is at the beginning of a line, but indenting LilyPond code like this makes it much easier for humans to read.

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

Simultaneous music expressions: single staff

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

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

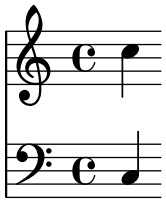


2.3.2 Multiple staves

LilyPond input files are constructed out of music expressions, as we saw in Section 2.3.1 [Music expressions explained], page 24. If the score begins with simultaneous music expressions, LilyPond creates multiples staves. However, it is easier to see what happens if we create each staff explicitly.

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

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



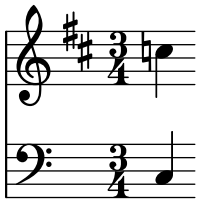
The command `\new` introduces a 'notation context.' A notation context is an environment in which musical events (like notes or `\clef` commands) are interpreted. For simple pieces, such notation contexts are created automatically. For more complex pieces, it is best to mark contexts explicitly.

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

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

Time signatures entered in one staff affects all other staves by default. On the other hand, the key signature of one staff does *not* affect other staves. This different default behavior is because scores with transposing instruments are more common than polyrhythmic scores.

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



2.3.3 Staff groups

Music Glossary: Section "brace" in *Music Glossary*.

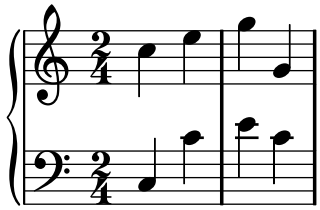
Piano music is typeset in two staves connected by a *brace*. Printing such a staff is similar to the polyphonic example in Section 2.3.2 [Multiple staves], page 25. However, now this entire expression is inserted inside a `PianoStaff`:

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

```
    \new Staff ...
  >>
```

Here is a small example:

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



Other staff groupings are introduced with `\new GrandStaff`, suitable for orchestral scores, and `\new ChoirStaff`, suitable for vocal scores. These staff groups each form another type of context, one that generates the brace at the left end of every system and also controls the extent of bar lines.

See also

Notation Reference: Section "Keyboard and other multi-staff instruments" in *Notation Reference*, Section "Displaying staves" in *Notation Reference*.

2.3.4 Combining notes into chords

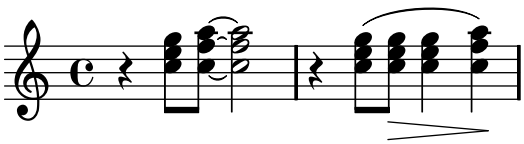
Music Glossary: Section "chord" in *Music Glossary*.

We saw earlier how notes can be combined into *chords* by indicating they are simultaneous by enclosing them in double angle brackets. However, the normal way of indicating a chord is to surround the pitches with *single* angle brackets. Note that all the notes in a chord must have the same duration, and that the duration is placed after the closing bracket.

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

Think of chords as almost equivalent to single notes: almost everything you can attach to a single note can be attached to a chord, and everything must go *outside* the angle brackets. For example, you can combine markings like beams and ties with chords. They must be placed outside the angle brackets.

```
r4 <c e g>8[ <c f a>]~ <c f a>2
r4 <c e g>8( <c e g>\> <c e g>4 <c f a>\!)
```



2.3.5 Single staff polyphony

Polyphonic music in lilypond, while not difficult, uses concepts that we haven't discussed yet, so we're not going to introduce them here. Instead, the following sections introduce these concepts and explain them thoroughly.

See also

Learning Manual: Section 3.2 [Voices contain music], page 44.

Notation Reference: Section "Simultaneous notes" in *Notation Reference*.

2.4 Songs

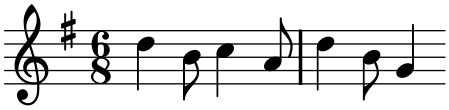
This section introduces vocal music and simple song sheets.

2.4.1 Setting simple songs

Music Glossary: Section "lyrics" in *Music Glossary*.

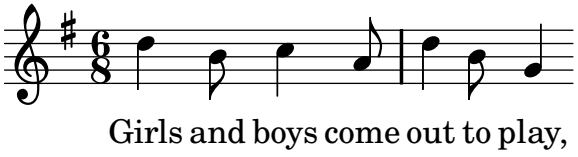
Here is the start of the melody to a nursery rhyme, *Girls and boys come out to play*:

```
\relative c'' {
  \key g \major
  \time 6/8
  d4 b8 c4 a8 d4 b8 g4
}
```



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

```
<<
  \relative c'' {
    \key g \major
    \time 6/8
    d4 b8 c4 a8 d4 b8 g4
  }
  \addlyrics {
    Girls and boys come out to play,
  }
>>
```



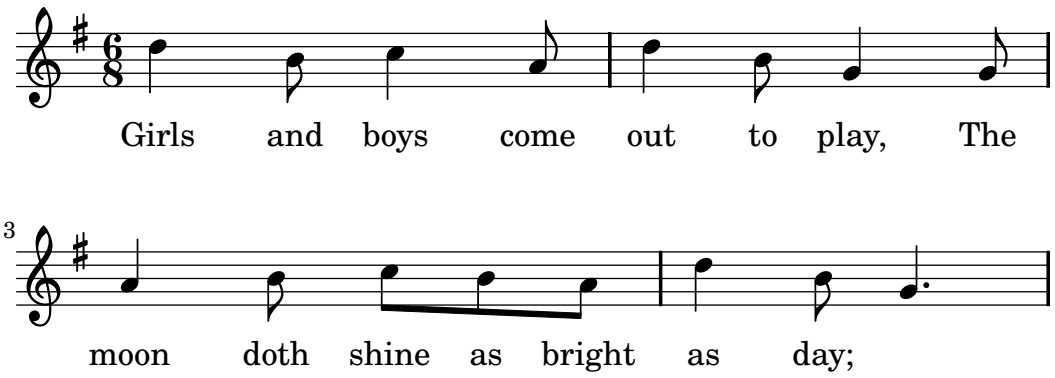
Note the curly brackets delimiting both the music and the lyrics, and the double angle brackets `<< ... >>` around the whole piece to show that the music and lyrics are to occur at the same time.

2.4.2 Aligning lyrics to a melody

Music Glossary: Section “melisma” in *Music Glossary*, Section “extender line” in *Music Glossary*.

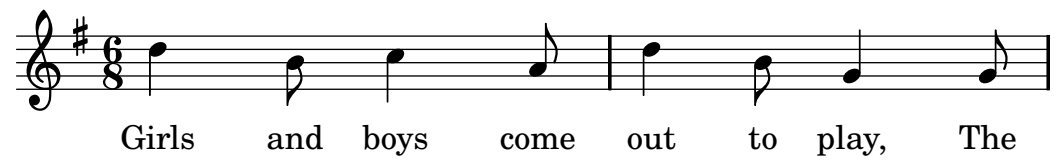
The next line in the nursery rhyme is *The moon doth shine as bright as day*. Let’s extend it:

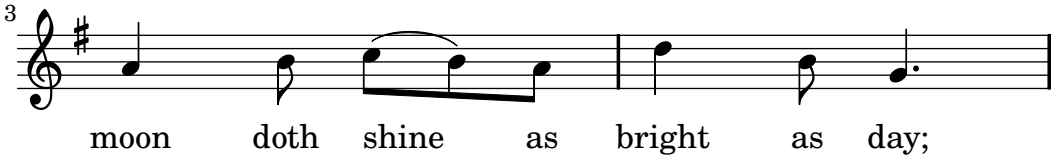
```
<<
  \relative c'' {
    \key g \major
    \time 6/8
    d4 b8 c4 a8 d4 b8 g4
    g8 a4 b8 c b a d4 b8 g4.
  }
  \addlyrics {
    Girls and boys come out to play,
    The moon doth shine as bright as day;
  }
>>
```



We see the extra lyrics do not align properly with the notes. The word *shine* should be sung on two notes, not one. This is called a *melisma*, a single syllable sung to more than one note. There are several ways to spread a syllable over multiple notes, the simplest being to add a slur across them, for details, see Section 2.2.2 [Ties and slurs], page 19:

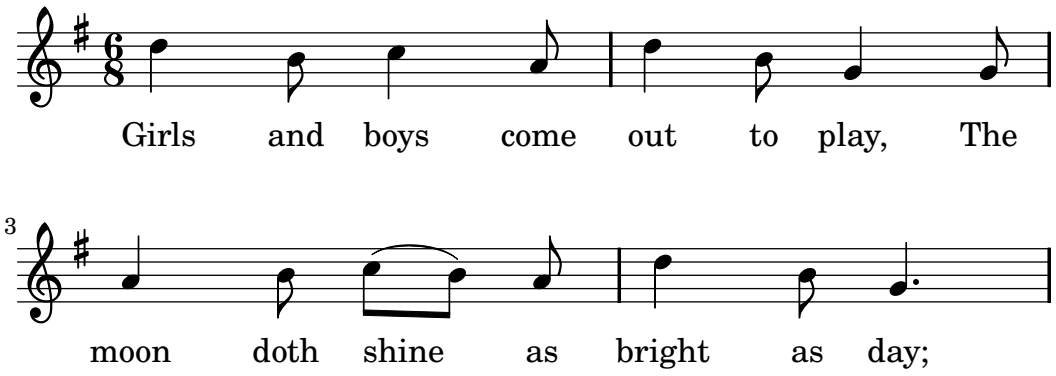
```
<<
  \relative c'' {
    \key g \major
    \time 6/8
    d4 b8 c4 a8 d4 b8 g4
    g8 a4 b8 c( b) a d4 b8 g4.
  }
  \addlyrics {
    Girls and boys come out to play,
    The moon doth shine as bright as day;
  }
>>
```





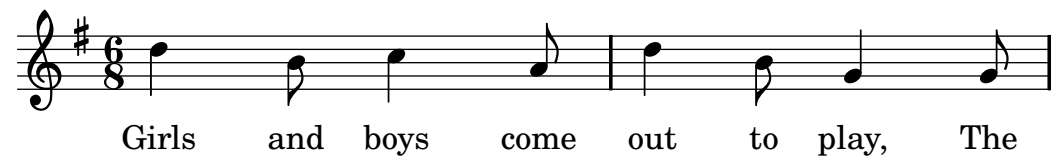
The words now line up correctly with the notes, but the automatic beaming for the notes above *shine as* does not look right. We can correct this by inserting manual beaming commands to override the automatic beaming here, for details, see Section 2.2.5 [Automatic and manual beams], page 22.

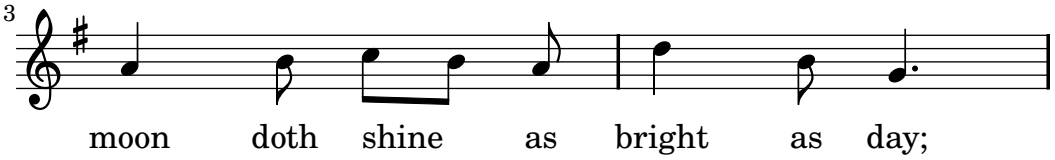
```
<<
  \relative c'' {
    \key g \major
    \time 6/8
    d4 b8 c4 a8 d4 b8 g4
    g8 a4 b8 c([ b]) a d4 b8 g4.
  }
  \addlyrics {
    Girls and boys come out to play,
    The moon doth shine as bright as day;
  }
>>
```



As an alternative to using slurs, the melismata may be indicated in just the lyrics by using an underscore _ for each note that should be included in the melisma:

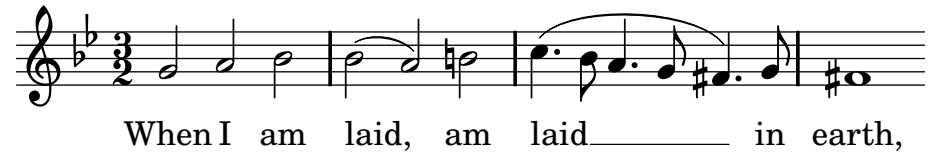
```
<<
  \relative c'' {
    \key g \major
    \time 6/8
    d4 b8 c4 a8 d4 b8 g4
    g8 a4 b8 c[ b] a d4 b8 g4.
  }
  \addlyrics {
    Girls and boys come out to play,
    The moon doth shine _ as bright as day;
  }
>>
```





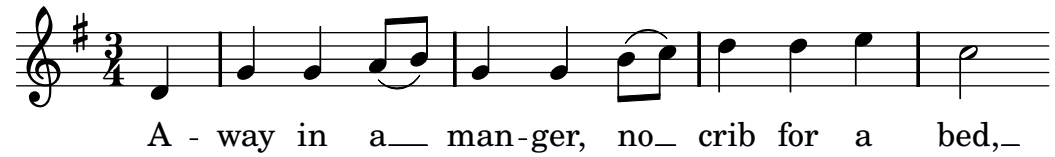
If a syllable extends over several notes or a single very long note an *extender line* is usually drawn from the syllable extending under all the notes for that syllable. It is entered as two underscores __. Here is an example from the first three bars of *Dido's Lament*, from Purcell's *Dido and Æneas*:

```
<<
  \relative c'' {
    \key g \minor
    \time 3/2
    g2 a bes bes( a)
    b c4.( bes8 a4. g8 fis4.) g8 fis1
  }
  \addlyrics {
    When I am laid,
    am laid __ in earth,
  }
>>
```



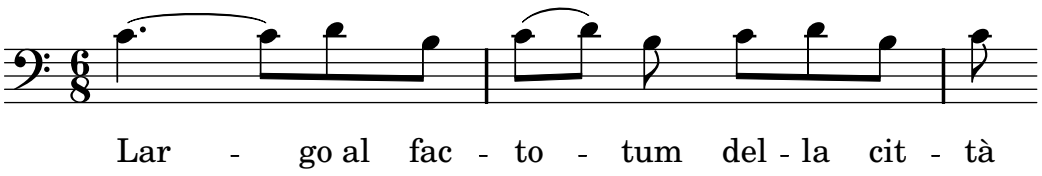
None of the examples so far have involved words containing more than one syllable. Such words are usually split one syllable to a note, with hyphens between syllables. Such hyphens are entered as two dashes, resulting in a centered hyphen between the syllables. Here is an example showing this and everything we have learned so far about aligning lyrics to notes.

```
<<
  \relative c' {
    \key g \major
    \time 3/4
    \partial 4
    d4 g4 g a8( b) g4 g4
    b8( c) d4 d e4 c2
  }
  \addlyrics {
    A -- way in a __ man -- ger,
    no __ crib for a bed, __
  }
>>
```



Some lyrics, especially those in Italian, require the opposite: setting more than one syllable to a single note. This is achieved by linking the syllables together with a single underscore _ (with no spaces), or enclosing them in quotes. Here's an example from Rossini's *Figaro*, where *al* has to be sung on the same note as the *go* of *Largo* in Figaro's aria *Largo al factotum*:

```
<<
  \relative c' {
    \clef bass
    \key c \major
    \time 6/8
    c4.~ c8 d b c([ d]) b c d b c
  }
  \addlyrics {
    Lar -- go_al fac -- to -- tum del -- la cit -- tà
  }
>>
```



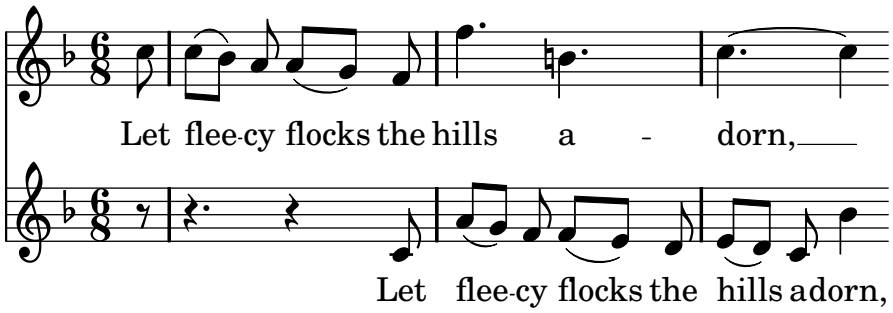
See also

Notation Reference: Section "Vocal music" in *Notation Reference*.

2.4.3 Lyrics to multiple staves

The simple approach using `\addlyrics` can be used for placing lyrics under more than one staff. Here is an example from Handel's *Judas Maccabæus*:

```
<<
  \relative c'' {
    \key f \major
    \time 6/8
    \partial 8
    c8 c([ bes]) a a([ g]) f f'4. b, c4.~ c4
  }
  \addlyrics {
    Let flee -- cy flocks the hills a -- dorn, __
  }
  \relative c' {
    \key f \major
    \time 6/8
    \partial 8
    r8 r4. r4 c8 a'([ g]) f f([ e]) d e([ d]) c bes'4
  }
  \addlyrics {
    Let flee -- cy flocks the hills a -- dorn,
  }
>>
```



Scores any more complex than this simple example are better produced by separating out the score structure from the notes and lyrics with variables. These are discussed in Section 2.5.1 [Organizing pieces with variables], page 33.

See also

Notation Reference: Section "Vocal music" in *Notation Reference*.

2.5 Final touches

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

2.5.1 Organizing pieces with variables

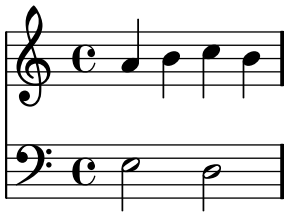
When all of the elements discussed earlier are combined to produce larger files, the music expressions get a lot bigger. In polyphonic music with many staves, the input files can become very confusing. We can reduce this confusion by using *variables*.

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

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

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

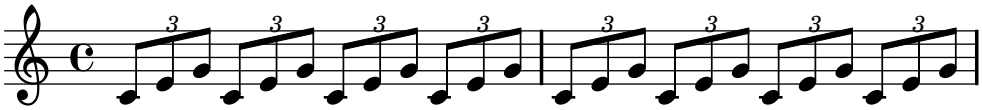
```
violin = \new Staff {
  \relative c'' {
    a4 b c b
  }
}
cello = \new Staff {
  \relative c {
    \clef bass
    e2 d
  }
}
{
  <<
    \violin
    \cello
  >>
}
```



The name of a variable must have alphabetic characters only, no numbers, underscores, or dashes.

Variables must be defined *before* the main music expression, but may be used as many times as required anywhere after they have been defined. They may even be used in a later definition of another variable, giving a way of shortening the input if a section of music is repeated many times.

```
tripletA = \times 2/3 { c,8 e g }
barA = { \tripletA \tripletA \tripletA \tripletA }

\relative c'' {
 \barA \barA
}
```



Variables may be used 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 variable can be used in different places. The following example uses the above variables:

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

2.5.2 Version number

The `\version` statement records the version of LilyPond that was used to write the file:

```
\version "2.12.3"
```

By convention, this is placed at the top of your LilyPond file.

These annotations make future upgrades of LilyPond go more smoothly. Changes in the syntax are handled with a special program, `convert-ly`, and it uses `\version` to determine what rules to apply. For details, see Section "Updating files with convert-ly" in *Application Usage*.

2.5.3 Adding titles

The title, composer, opus number, and similar information are entered in the `\header` block. This exists outside of the main music expression; the `\header` block is usually placed underneath the version number.

```
\version "2.12.3"
\header {
  title = "Symphony"
  composer = "Me"
  opus = "Op. 9"
}

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

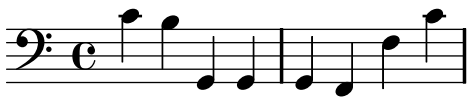
When the file is processed, the title and composer are printed above the music. More information on titling can be found in Section "Creating titles" in *Notation Reference*.

2.5.4 Absolute note names

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

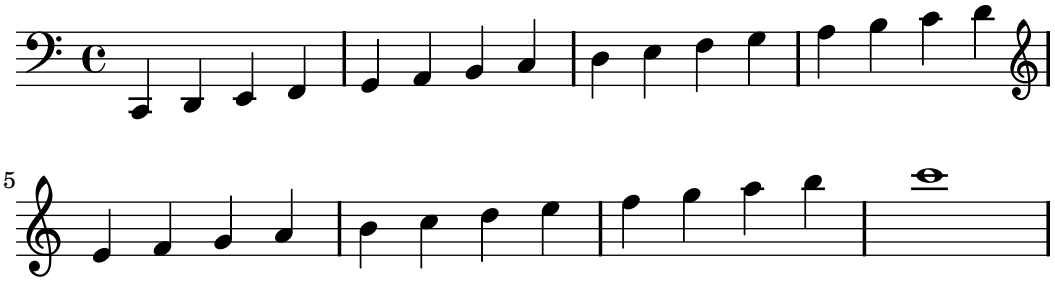
If you omit the `\relative`, LilyPond treats all pitches as absolute values. A `c'` will always mean middle C, a `b` will always mean the note one step below middle C, and a `g,` will always mean the note on the bottom staff of the bass clef.

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



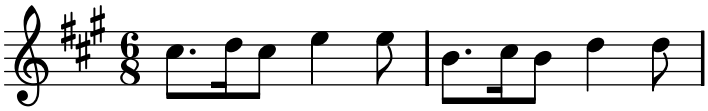
Here is a four-octave scale:

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



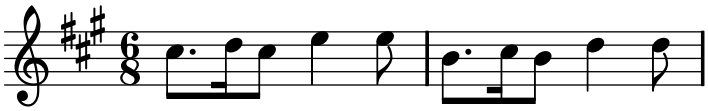
As you can see, writing a melody in the treble clef involves a lot of quote ' marks. Consider this fragment from Mozart:

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



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

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



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

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

2.5.5 After the tutorial

After finishing the tutorial, you should probably try writing a piece or two. Start by adding notes to one of the Appendix A [Templates], page 141. If you need any notation that was not covered in the tutorial, look at the Notation Reference, starting with Section "Musical notation" in *Notation Reference*. If you want to write for an instrument ensemble that is not covered in the templates, take a look at Section 3.4 [Extending the templates], page 68.

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

In this tutorial and in the rest of the Learning Manual, there is a paragraph **See also** at the end of each section, which contains cross-references to other sections: you should not follow these cross-references at first reading; when you have read all of the Learning Manual, you may want to read some sections again and follow cross-references for further reading.

If you have not done so already, *please* read Section 1.2 [About the documentation], page 8. There is a lot of information about LilyPond, so newcomers often do not know where they should look for help. If you spend five minutes reading that section carefully, you might save yourself hours of frustration looking in the wrong places!

3 Fundamental concepts

You've seen in the Tutorial how to produce beautifully printed music from a simple text file. This section introduces the concepts and techniques required to produce equally beautiful but more complex scores.

3.1 How LilyPond input files work

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

3.1.1 Introduction to the LilyPond file structure

A basic example of a LilyPond input file is

```
\version "2.12.3"
\header { }
\score {
  ...compound music expression...  % all the music goes here!
  \layout { }
  \midi { }
}
```

There are many variations of this basic pattern, but this example serves as a useful starting place.

Up to this point none of the examples you have seen has used a `\score{}` command. This is because LilyPond automatically adds the extra commands which are needed when you give it simple input. LilyPond treats input like this:

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

as shorthand for this:

```
\book {
  \score {
    \new Staff {
      \new Voice {
        \relative c'' {
          c4 a b c
        }
      }
    }
    \layout { }
  }
}
```

In other words, if the input contains a single music expression, LilyPond will interpret the file as though the music expression was wrapped up inside the commands shown above.

A word of warning! Many of the examples in the LilyPond documentation will omit the `\new Staff` and `\new Voice` commands, leaving them to be created implicitly. For simple examples this works well, but for more complex examples, especially when additional commands are used, the implicit creation of contexts can give surprising results, maybe creating extra unwanted

staves. The way to create contexts explicitly is explained in Section 3.3 [Contexts and engravers], page 58.

> **Note:** When entering more than a few lines of music it is advisable to always create staves and voices explicitly.

For now, though, let us return to the first example and examine the `\score` command, leaving the others to default.

A `\score` block must always contain just one music expression, and this must appear immediately after the `\score` command. Remember that a music expression could be anything from a single note to a huge compound expression like

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

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

As we saw previously, the `\score` block can contain other things, such as

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

Note that these three commands – `\header`, `\layout` and `\midi` – are special: unlike many other commands which begin with a backward slash (\) they are *not* music expressions and are not part of any music expression. So they may be placed inside a `\score` block or outside it. In fact, these commands are commonly placed outside the `\score` block – for example, `\header` is often placed above the `\score` command, as the example at the beginning of this section shows.

Two more commands you have not previously seen are `\layout { }` and `\midi {}`. If these appear as shown they will cause LilyPond to produce a printed output and a MIDI output respectively. They are described fully in the Notation Reference – Section "Score layout" in *Notation Reference*, and Section "Creating MIDI files" in *Notation Reference*.

You may code multiple `\score` blocks. Each will be treated as a separate score, but they will be all combined into a single output file. A `\book` command is not necessary – one will be implicitly created. However, if you would like separate output files from one `.ly` file then the `\book` command should be used to separate the different sections: each `\book` block will produce a separate output file.

In summary:

Every `\book` block creates a separate output file (e.g., a PDF file). If you haven't explicitly added one, LilyPond wraps your entire input code in a `\book` block implicitly.

Every `\score` block is a separate chunk of music within a `\book` block.

Every `\layout` block affects the `\score` or `\book` block in which it appears – i.e., a `\layout` block inside a `\score` block affects only that `\score` block, but a `\layout` block outside of a `\score` block (and thus in a `\book` block, either explicitly or implicitly) will affect every `\score` in that `\book`.

For details see Section "Multiple scores in a book" in *Notation Reference*.

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

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

\score {
  \melody
}
```

When LilyPond looks at this file, it takes the value of `melody` (everything after the equals sign) and inserts it whenever it sees `\melody`. There's nothing special about the names – it could be `melody`, `global`, `TimeKey`, `pianorighthand`, or `foofoobarbaz`. For more details, see Section 5.1.4 [Saving typing with variables and functions], page 133. Remember that you can use almost any name you like as long as it contains just alphabetic characters and is distinct from LilyPond command names. The exact limitations on variable names are detailed in Section "File structure" in *Notation Reference*.

See also

For a complete definition of the input format, see Section "File structure" in *Notation Reference*.

3.1.2 Score is a (single) compound musical expression

We saw the general organization of LilyPond input files in the previous section, Section 3.1.1 [Introduction to the LilyPond file structure], page 37. But we seemed to skip over the most important part: how do we figure out what to write after `\score`?

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

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

To understand what is meant by a music expression and a compound music expression, you may find it useful to review the tutorial, Section 2.3.1 [Music expressions explained], page 24. In that section, we saw how to build big music expressions from small pieces – we started from notes, then chords, etc. Now we're going to start from a big music expression and work our way down.

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

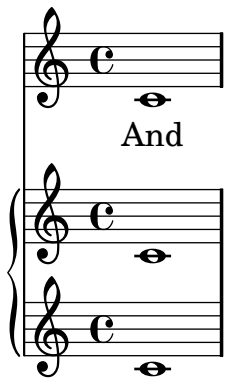
A whole Wagner opera would easily double the length of this manual, so let's just add a singer and piano. We don't need a `StaffGroup` for this ensemble, which simply groups a number of staves together with a bracket at the left, so we shall remove it. We *do* need a singer and a piano, though.

```
\score {
  <<
    \new Staff = "singer" <<
    >>
    \new PianoStaff = "piano" <<
    >>
  >>
  \layout { }
```

```
}
```

Remember that we use `<< ... >>` instead of `{ ... }` to show simultaneous music. And we definitely want to show the vocal part and piano part at the same time, not one after the other! Note that the `<< ... >>` construct is not really necessary for the Singer staff, as it contains only one sequential music expression; however, using `<< ... >>` instead of braces is still necessary if the music in the Staff is made of two simultaneous expressions, e.g. two simultaneous Voices, or a Voice with lyrics. We'll add some real music later; for now let's just put in some dummy notes and lyrics.

```
\score {
  <<
    \new Staff = "singer" <<
      \new Voice = "vocal" { c'1 }
      \addlyrics { And }
    >>
    \new PianoStaff = "piano" <<
      \new Staff = "upper" { c'1 }
      \new Staff = "lower" { c'1 }
    >>
  >>
  \layout { }
}
```



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

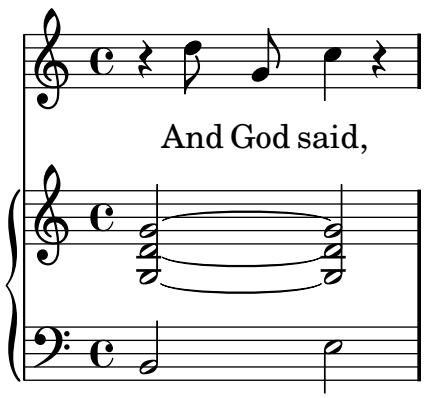
At this stage, we could start filling in notes. Inside the curly braces next to `\new Voice = "vocal"`, we could start writing

```
\relative c'' {
  r4 d8\noBeam g, c4 r
}
```

But if we did that, the `\score` section would get pretty long, and it would be harder to understand what was happening. So let's use variables instead. These were introduced at the end of the previous section, remember? So, adding a few notes, we now have a piece of real music:

```
melody = \relative c'' { r4 d8\noBeam g, c4 r }
text   = \lyricmode { And God said, }
upper  = \relative c'' { <g d g,>2~ <g d g,> }
lower  = \relative c { b2 e2 }
```

```
\score {
  <<
    \new Staff = "singer" <<
      \new Voice = "vocal" { \melody }
      \addlyrics { \text }
    >>
    \new PianoStaff = "piano" <<
      \new Staff = "upper" { \upper }
      \new Staff = "lower" {
        \clef "bass"
        \lower
      }
    >>
  >>
  \layout { }
}
```



Be careful about the difference between notes, which are introduced with `\relative` or which are directly included in a music expression, and lyrics, which are introduced with `\lyricmode`. These are essential to tell LilyPond to interpret the following content as music and text respectively.

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

See also

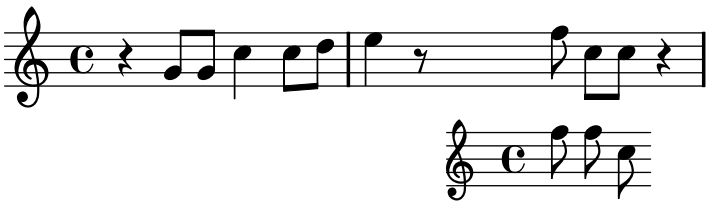
Notation Reference: Section "Structure of a score" in *Notation Reference*.

3.1.3 Nesting music expressions

It is not essential to declare all staves at the beginning; they may be introduced temporarily at any point. This is particularly useful for creating ossia sections – see Section "ossia" in *Music Glossary*. Here is a simple example showing how to introduce a new staff temporarily for the duration of three notes:

```
\new Staff {
  \relative g' {
    r4 g8 g c4 c8 d |
    e4 r8
```

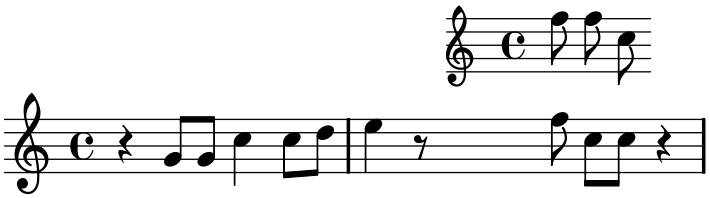
```
    <<
      { f c c }
      \new Staff {
        f8 f c
      }
    >>
    r4 |
  }
}
```



Note that the size of the clef is the same as a clef printed following a clef change – slightly smaller than the clef at the beginning of the line. This is usual for clefs printed in the middle of a line.

The ossia section may be placed above the staff as follows:

```
\new Staff = "main" {
  \relative g' {
    r4 g8 g c4 c8 d |
    e4 r8
    <<
      { f c c }
      \new Staff \with {
        alignAboveContext = #"main" }
      { f8 f c }
    >>
    r4 |
  }
}
```



This example uses `\with`, which will be explained more fully later. It is a means of modifying the default behavior of a single Staff. Here it says that the new staff should be placed above the staff called “main” instead of the default position which is below.

See also

Ossia are often written without clef and without time signature and are usually in a smaller font. These require further commands which have not yet been introduced. See Section 4.3.2 [Size of objects], page 95, and Section “Ossia staves” in *Notation Reference*.

3.1.4 On the un-nestedness of brackets and ties

You have already met a number of different types of bracket in writing the input file to LilyPond. These obey different rules which can be confusing at first. Before we explain the rules let's first review the different types of bracket.

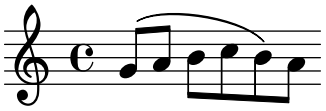
Bracket Type	**Function**
`{ .. }`	Encloses a sequential segment of music
`< .. >`	Encloses the notes of a chord
`<< .. >>`	Encloses simultaneous music expressions
`( .. )`	Marks the start and end of a slur
`\( .. \)`	Marks the start and end of a phrasing slur
`[ .. ]`	Marks the start and end of a manual beam

To these we should add other constructs which generate lines between or across notes: ties (marked by a tilde, `~`), tuplets written as `\times x/y {..}`, and grace notes written as `\grace{..}`.

Outside LilyPond, the conventional use of brackets requires the different types to be properly nested, like this, `<< [ { ( .. ) } ] >>`, with the closing brackets being encountered in exactly the opposite order to the opening brackets. This **is** a requirement for the three types of bracket described by the word 'Encloses' in the table above – they must nest properly. However, the remaining brackets, described with the word 'Marks' in the table above together with ties and tuplets, do **not** have to nest properly with any of the brackets. In fact, these are not brackets in the sense that they enclose something – they are simply markers to indicate where something starts and ends.

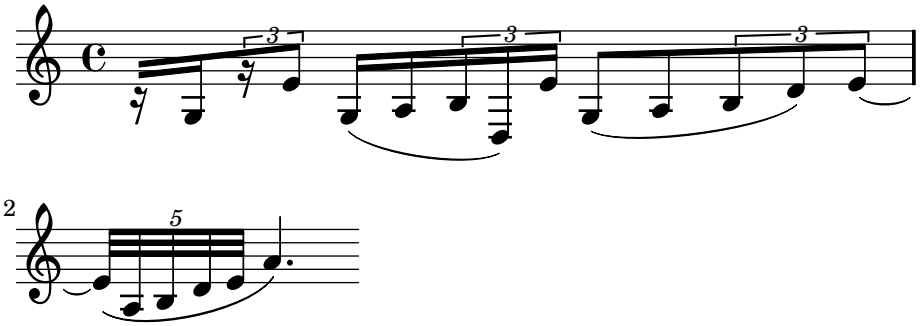
So, for example, a phrasing slur can start before a manually inserted beam and end before the end of the beam – not very musical, perhaps, but possible:

```
{ g8\( a b[ c b\) a] }
```



In general, different kinds of brackets, and those implied by tuplets, ties and grace notes, may be mixed freely. This example shows a beam extending into a tuplet (line 1), a slur extending into a tuplet (line 2), a beam and a slur extending into a tuplet, a tie crossing two tuplets, and a phrasing slur extending out of a tuplet (lines 3 and 4).

```
{
  r16[ g16 \times 2/3 {r16 e'8] }
  g16( a \times 2/3 {b d) e' }
  g8[( a \times 2/3 {b d') e'~]}
  \times 4/5 {e'32\( a b d' e'} a'4.\)
}
```



3.2 Voices contain music

Singers need voices to sing, and so does LilyPond. The actual music for all instruments in a score is contained in Voices – the most fundamental of all LilyPond's concepts.

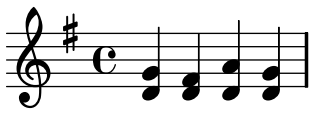
3.2.1 I'm hearing Voices

The lowest, most fundamental or innermost layers in a LilyPond score are called 'Voice contexts' or just 'Voices' for short. Voices are sometimes called 'layers' in other notation packages.

In fact, a Voice layer or context is the only one which can contain music. If a Voice context is not explicitly declared one is created automatically, as we saw at the beginning of this chapter. Some instruments such as an Oboe can play only one note at a time. Music written for such instruments is monophonic and requires just a single voice. Instruments which can play more than one note at a time like the piano will often require multiple voices to encode the different concurrent notes and rhythms they are capable of playing.

A single voice can contain many notes in a chord, of course, so when exactly are multiple voices needed? Look first at this example of four chords:

```
\key g \major
<d g>4 <d fis> <d a'> <d g>
```



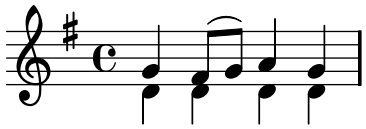
This can be expressed using just the single angle bracket chord symbols, `< ... >`, and for this just a single voice is needed. But suppose the F-sharp were actually an eighth-note followed by an eighth-note G, a passing note on the way to the A? Now we have two notes which start at the same time but have different durations: the quarter-note D and the eighth-note F-sharp. How are these to be coded? They cannot be written as a chord because all the notes in a chord must have the same duration. And they cannot be written as two sequential notes as they need to start at the same time. This is when two voices are required.

Let us see how this is done in LilyPond input syntax.

The easiest way to enter fragments with more than one voice on a staff is to enter each voice as a sequence (with `{...}`), and combine them simultaneously with angle brackets, `<<...>>`. The fragments must also be separated with double backward slashes, `\\`, to place them in separate voices. Without these, the notes would be entered into a single voice, which would usually cause errors. This technique is particularly suited to pieces of music which are largely monophonic with occasional short sections of polyphony.

Here's how we split the chords above into two voices and add both the passing note and a slur:

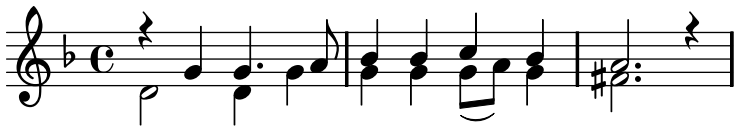
```
\key g \major
%    Voice "1"                  Voice "2"
<< { g4 fis8( g) a4 g }    \\ { d4 d d d }  >> |
```



Notice how the stems of the second voice now point down.

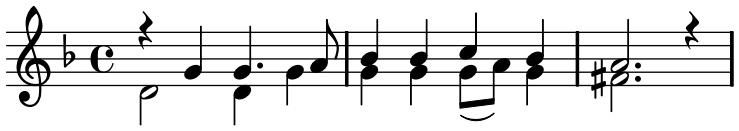
Here's another simple example:

```
\key d \minor
%    Voice "1"              Voice "2"
<< { r4 g g4. a8 }    \\ { d,2 d4 g }       >> |
<< { bes4 bes c bes } \\ { g4 g g8( a) g4 } >> |
<< { a2. r4 }         \\ { fis2. s4 }       >> |
```



It is not necessary to use a separate << \\ >> construct for each bar. For music with few notes in each bar this layout can help the legibility of the code, but if there are many notes in each bar it may be better to split out each voice separately, like this:

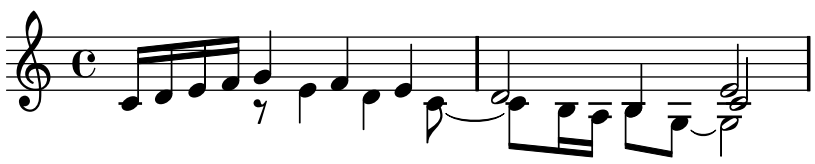
```
\key d \minor
<< {
  % Voice "1"
  r4 g g4. a8 |
  bes4 bes c bes |
  a2. r4 |
} \\ {
  % Voice "2"
  d,2 d4 g |
  g4 g g8( a) g4 |
  fis2. s4 |
} >>
```



This example has just two voices, but the same construct may be used to encode three or more voices by adding more back-slash separators.

The Voice contexts bear the names "1", "2", etc. In each of these contexts, the vertical direction of slurs, stems, ties, dynamics etc., is set appropriately.

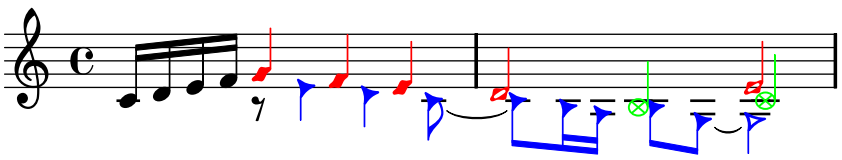
```
\new Staff \relative c' {
  % Main voice
  c16 d e f
  %    Voice "1"     Voice "2"                  Voice "3"
  << { g4 f e } \\ { r8 e4 d c8 ~ } >> |
  << { d2 e2 }  \\ { c8 b16 a b8 g ~ g2 } \\ { s4 b4 c2 } >> |
}
```



These voices are all separate from the main voice that contains the notes just outside the << .. >> construct. Let's call this the *simultaneous construct*. Slurs and ties may only connect notes within the same voice, so slurs and ties cannot go into or out of a simultaneous construct.

Conversely, parallel voices from separate simultaneous constructs on the same staff are the same voice. Other voice-related properties also carry across simultaneous constructs. Here is the same example, with different colors and note heads for each voice. Note that changes in one voice do not affect other voices, but they do persist in the same voice later. Note also that tied notes may be split across the same voices in two constructs, shown here in the blue triangle voice.

```
\new Staff \relative c' {
  % Main voice
  c16 d e f
  << % Bar 1
    {
      \voiceOneStyle
      g4 f e
    }
  \\
    {
      \voiceTwoStyle
      r8 e4 d c8 ~
    }
  >>
  << % Bar 2
     % Voice 1 continues
    { d2 e2 }
  \\
     % Voice 2 continues
    { c8 b16 a b8 g ~ g2 }
  \\
    {
      \voiceThreeStyle
      s4 b4 c2
    }
  >>
}
```



The commands `\voiceXXXStyle` are mainly intended for use in educational documents such as this one. They modify the color of the note head, the stem and the beams, and the style of the note head, so that the voices may be easily distinguished. Voice one is set to red diamonds, voice two to blue triangles, voice three to green crossed circles, and voice four (not used here) to magenta crosses; `\voiceNeutralStyle` (also not used here) reverts the style back to the default. We shall see later how commands like these may be created by the user. See Section 4.3.1 [Visibility and color of objects], page 91 and Section 4.6.2 [Using variables for tweaks], page 126.

Polyphony does not change the relationship of notes within a `\relative { }` block. Each note is still calculated relative to the note immediately preceding it, or to the first note of the preceding chord. So in

```
\relative c' { noteA << < noteB noteC > \\ noteD >> noteE }
```

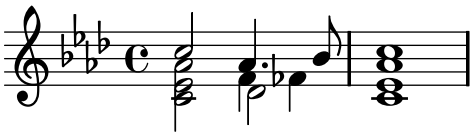
`noteB` is relative to `noteA`
`noteC` is relative to `noteB`, not `noteA`;

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

An alternative way, which may be clearer if the notes in the voices are widely separated, is to place a `\relative` command at the start of each voice:

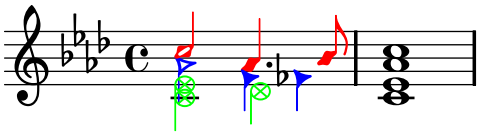
```
\relative c' { noteA ... }
<<
  \relative c'' { < noteB noteC > ... }
\\
  \relative g' { noteD ... }
>>
\relative c' { noteE ... }
```

Let us finally analyze the voices in a more complex piece of music. Here are the notes from the first two bars of the second of Chopin's Deux Nocturnes, Op 32. This example will be used at later stages in this and the next chapter to illustrate several techniques for producing notation, so please ignore for now anything in the underlying code which looks mysterious and concentrate just on the music and the voices – the complications will all be explained in later sections.



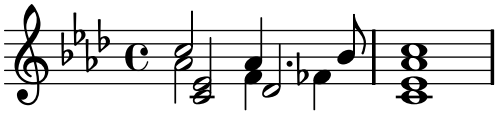
The direction of the stems is often used to indicate the continuity of two simultaneous melodic lines. Here the stems of the highest notes are all pointing up and the stems of the lower notes are all pointing down. This is the first indication that more than one voice is required.

But the real need for multiple voices arises when notes which start at the same time have different durations. Look at the notes which start at beat three in the first bar. The A-flat is a dotted quarter note, the F is a quarter note and the D-flat is a half note. These cannot be written as a chord as all the notes in a chord must have the same duration. Neither can they be written as sequential notes, as they must start at the same time. This section of the bar requires three voices, and the normal practice would be to write the whole bar as three voices, as shown below, where we have used different note heads and colors for the three voices. Again, the code behind this example will be explained later, so ignore anything you do not understand.



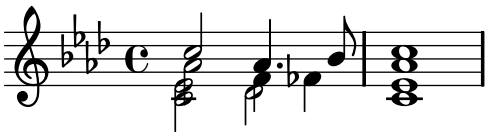
Let us try to encode this music from scratch. As we shall see, this encounters some difficulties. We begin as we have learnt, using the `<< \\ >>` construct to enter the music of the first bar in three voices:

```
\new Staff \relative c'' {
  \key aes \major
  <<
    { c2 aes4. bes8 } \\ { aes2 f4 fes } \\ { <ees c>2 des2 }
  >>
  <c ees aes c>1
}
```



The stem directions are automatically assigned with the odd-numbered voices taking upward stems and the even-numbered voices downward ones. The stems for voices 1 and 2 are right, but the stems in voice 3 should go down in this particular piece of music. We can correct this simply by missing out voice three and placing the music in voice four:

```
\new Staff \relative c'' {
  \key aes \major
  << % Voice one
    { c2 aes4. bes8 }
  \\ % Voice two
    { aes2 f4 fes   }
  \\ % Omit Voice three
  \\ % Voice four
    { <ees c>2 des2 }
  >> |
  <c ees aes c>1 |
}
```



We see that this fixes the stem direction, but exposes a problem sometimes encountered with multiple voices – the stems of the notes in one voice can collide with the note heads in other voices. In laying out the notes, LilyPond allows the notes or chords from two voices to occupy the same vertical note column provided the stems are in opposite directions, but the notes from the third and fourth voices are displaced, if necessary, to avoid the note heads colliding. This usually works well, but in this example the notes of the lowest voice are clearly not well placed by default. LilyPond provides several ways to adjust the horizontal placing of notes. We are not quite ready yet to see how to correct this, so we shall leave this problem until a later section — see the `force-hshift` property in Section 4.5.2 [Fixing overlapping notation], page 111.

See also

Notation Reference: Section “Multiple voices” in *Notation Reference*.

3.2.2 Explicitly instantiating voices

Voice contexts can also be created manually inside a `<< >>` block to create polyphonic music, using `\voiceOne` ... `\voiceFour` to indicate the required directions of stems, slurs, etc. In longer scores this method is clearer, as it permits the voices to be separated and to be given more descriptive names.

Specifically, the construct `<< \\ >>` which we used in the previous section:

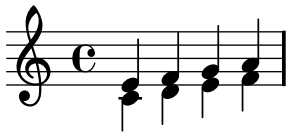
```
\new Staff {
  \relative c' {
    << { e4 f g a } \\ { c,4 d e f } >>
  }
}
```

is equivalent to

```
\new Staff <<
  \new Voice = "1" { \voiceOne \relative c' { e4 f g a } }
```

```
  \new Voice = "2" { \voiceTwo \relative c' { c4 d e f } }
>>
```

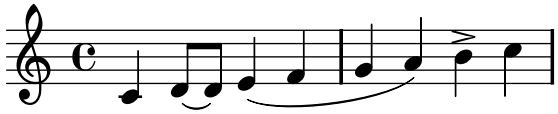
Both of the above would produce



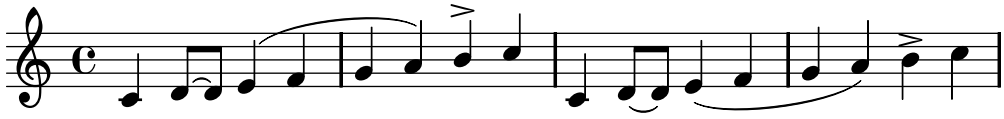
The `\voiceXXX` commands set the direction of stems, slurs, ties, articulations, text annotations, augmentation dots of dotted notes, and fingerings. `\voiceOne` and `\voiceThree` make these objects point upwards, while `\voiceTwo` and `\voiceFour` make them point downwards. These commands also generate a horizontal shift for each voice when this is required to avoid clashes of note heads. The command `\oneVoice` reverts the settings back to the normal values for a single voice.

Let us see in some simple examples exactly what effect `\oneVoice`, `\voiceOne` and `voiceTwo` have on markup, ties, slurs, and dynamics:

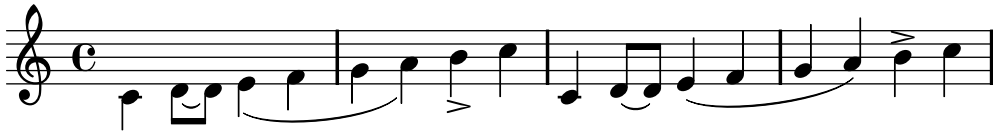
```
\relative c'{
  % Default behavior or behavior after \oneVoice
  c d8 ~ d e4 ( f g a ) b-> c
}
```



```
\relative c'{
  \voiceOne
  c d8 ~ d e4 ( f g a ) b-> c
  \oneVoice
  c, d8 ~ d e4 ( f g a ) b-> c
}
```



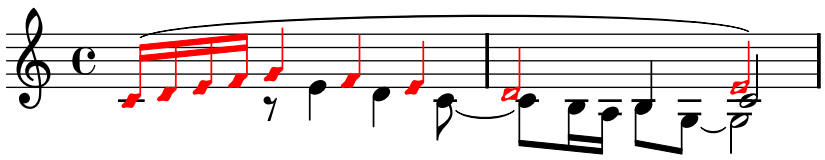
```
\relative c'{
  \voiceTwo
  c d8 ~ d e4 ( f g a ) b-> c
  \oneVoice
  c, d8 ~ d e4 ( f g a ) b-> c
}
```



Now let's look at three different ways to notate the same passage of polyphonic music, each of which is advantageous in different circumstances, using the example from the previous section.

An expression that appears directly inside a << >> belongs to the main voice (but, note, **not** in a << \\ >> construct). This is useful when extra voices appear while the main voice is playing. Here is a more correct rendition of our example. The red diamond-shaped notes demonstrate that the main melody is now in a single voice context, permitting a phrasing slur to be drawn over them.

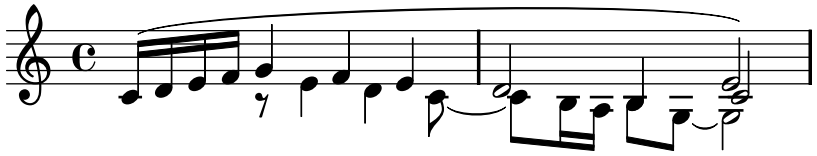
```
\new Staff \relative c' {
  \voiceOneStyle
  % The following notes are monophonic
  c16^( d e f
  % Start simultaneous section of three voices
  <<
    % Continue the main voice in parallel
    { g4 f e | d2 e2) }
    % Initiate second voice
    \new Voice {
      % Set stems, etc, down
      \voiceTwo
      r8 e4 d c8 ~ | c8 b16 a b8 g ~ g2
    }
    % Initiate third voice
    \new Voice {
      % Set stems, etc, up
      \voiceThree
      s2. | s4 b4 c2
    }
  >>
}
```



More deeply nested polyphony constructs are possible, and if a voice appears only briefly this might be a more natural way to typeset the music:

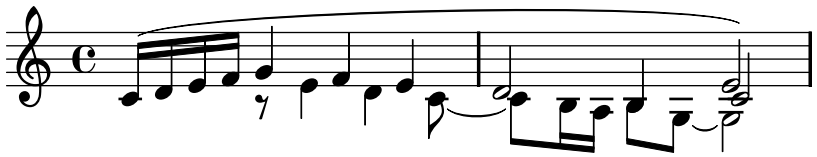
```
\new Staff \relative c' {
  c16^( d e f
  <<
    { g4 f e | d2 e2) }
    \new Voice {
      \voiceTwo
      r8 e4 d c8 ~ |
      <<
        {c8 b16 a b8 g ~ g2}
        \new Voice {
          \voiceThree
          s4 b4 c2
        }
      >>
    }
  >>
```

```
}
```



This method of nesting new voices briefly is useful when only small sections of the music are polyphonic, but when the whole staff is largely polyphonic it can be clearer to use multiple voices throughout, using spacing notes to step over sections where the voice is silent, as here:

```
\new Staff \relative c' <<
  % Initiate first voice
  \new Voice {
    \voiceOne
    c16^( d e f g4 f e | d2 e2) |
  }
  % Initiate second voice
  \new Voice {
    % Set stems, etc, down
    \voiceTwo
    s4 r8 e4 d c8 ~ | c8 b16 a b8 g ~ g2 |
  }
  % Initiate third voice
  \new Voice {
    % Set stems, etc, up
    \voiceThree
    s1 | s4 b4 c2 |
  }
>>
```



Note columns

Closely spaced notes in a chord, or notes occurring at the same time in different voices, are arranged in two, occasionally more, columns to prevent the note heads overlapping. These are called note columns. There are separate columns for each voice, and the currently specified voice-dependent shift is applied to the note column if there would otherwise be a collision. This can be seen in the example above. In bar 2 the C in voice two is shifted to the right relative to the D in voice one, and in the final chord the C in voice three is also shifted to the right relative to the other notes.

The `\shiftOn`, `\shiftOnn`, `\shiftOnnn`, and `\shiftOff` commands specify the degree to which notes and chords of the voice should be shifted if a collision would otherwise occur. By default, the outer voices (normally voices one and two) have `\shiftOff` specified, while the inner voices (three and four) have `\shiftOn` specified. When a shift is applied, voices one and three are shifted to the right and voices two and four to the left.

`\shiftOnn` and `\shiftOnnn` define further shift levels which may be specified temporarily to resolve collisions in complex situations – see Section 4.5.3 [Real music example], page 116.

A note column can contain just one note (or chord) from a voice with stems up and one note (or chord) from a voice with stems down. If notes from two voices which have their stems in the same direction are placed at the same position and both voices have no shift or the same shift specified, the error message "Too many clashing note columns" will be produced.

See also

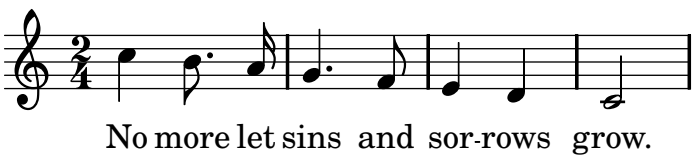
Notation Reference: Section "Multiple voices" in *Notation Reference*.

3.2.3 Voices and vocals

Vocal music presents a special difficulty: we need to combine two expressions – notes and lyrics.

You have already seen the `\addlyrics{}` command, which handles simple scores well. However, this technique is quite limited. For more complex music, you must introduce the lyrics in a `Lyrics` context using `\new Lyrics` and explicitly link the lyrics to the notes with `\lyricsto{}`, using the name assigned to the Voice.

```
<<
  \new Voice = "one" \relative c'' {
    \autoBeamOff
    \time 2/4
    c4 b8. a16 g4. f8 e4 d c2
  }
  \new Lyrics \lyricsto "one" {
    No more let sins and sor -- rows grow.
  }
>>
```



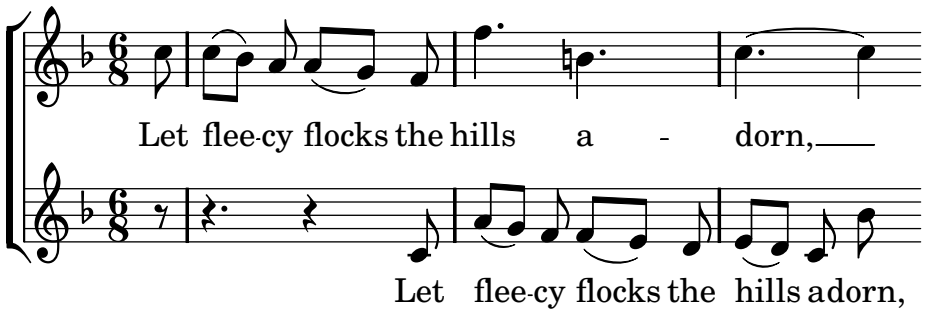
Note that the lyrics must be linked to a `Voice` context, *not* a `Staff` context. This is a case where it is necessary to create `Staff` and `Voice` contexts explicitly.

The automatic beaming which LilyPond uses by default works well for instrumental music, but not so well for music with lyrics, where beaming is either not required at all or is used to indicate melismata in the lyrics. In the example above we use the command `\autoBeamOff` to turn off the automatic beaming.

Let us reuse the earlier example from Judas Maccabæus to illustrate this more flexible technique. We first recast it to use variables so the music and lyrics can be separated from the staff structure. We also introduce a ChoirStaff bracket. The lyrics themselves must be introduced with `\lyricmode` to ensure they are interpreted as lyrics rather than music.

```
global = { \time 6/8 \partial 8 \key f \major}
SopOneMusic = \relative c'' {
  c8 | c([ bes)] a a([ g)] f | f'4. b, | c4.~ c4 }
SopTwoMusic = \relative c' {
  r8 | r4. r4 c8 | a'([ g)] f f([ e)] d | e([ d)] c bes' }
SopOneLyrics = \lyricmode {
  Let | flee -- cy flocks the | hills a -- dorn, __ }
SopTwoLyrics = \lyricmode {
  Let | flee -- cy flocks the | hills a -- dorn, }
```

```
\score {
  \new ChoirStaff <<
    \new Staff <<
      \new Voice = "SopOne" {
        \global
        \SopOneMusic
      }
      \new Lyrics \lyricsto "SopOne" {
        \SopOneLyrics
      }
    >>
    \new Staff <<
      \new Voice = "SopTwo" {
        \global
        \SopTwoMusic
      }
      \new Lyrics \lyricsto "SopTwo" {
        \SopTwoLyrics
      }
    >>
  >>
}
```

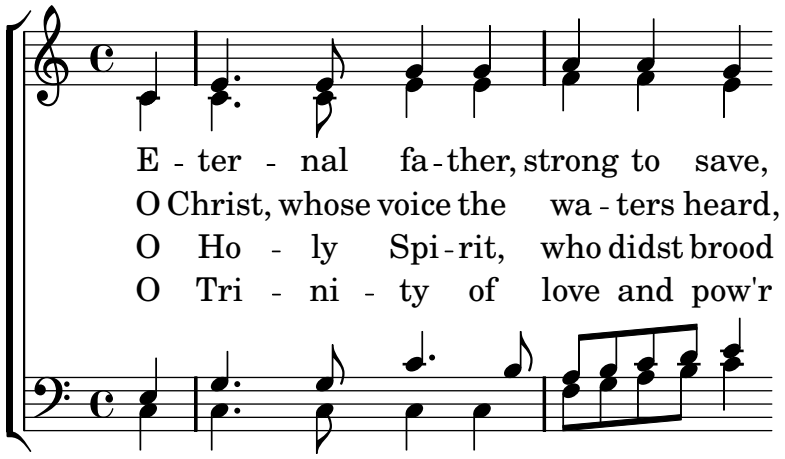


This is the basic structure of all vocal scores. More staves may be added as required, more voices may be added to the staves, more verses may be added to the lyrics, and the variables containing the music can easily be placed in separate files should they become too long.

Here is an example of the first line of a hymn with four verses, set for SATB. In this case the words for all four parts are the same. Note how we use variables to separate the music notation and words from the staff structure. See too how a variable, which we have chosen to call 'TimeKey', is used to hold several commands for use within the two staves. In other examples this is often called 'global'.

```
TimeKey = { \time 4/4 \partial 4 \key c \major}
SopMusic   = \relative c' { c4 | e4. e8 g4  g  | a a g }
AltoMusic  = \relative c' { c4 | c4. c8 e4  e  | f f e }
TenorMusic = \relative c  { e4 | g4. g8 c4. b8 | a8 b c d e4 }
BassMusic  = \relative c  { c4 | c4. c8 c4  c  | f8 g a b c4 }
VerseOne   = \lyricmode {
  E -- | ter -- nal fa -- ther, | strong to save, }
VerseTwo   = \lyricmode {
  O | Christ, whose voice the | wa -- ters heard, }
VerseThree = \lyricmode {
  O | Ho -- ly Spi -- rit, | who didst brood }
```

```
VerseFour  = \lyricmode {
  O | Tri -- ni -- ty of | love and pow'r }

\score {
  \new ChoirStaff <<
    \new Staff <<
      \clef "treble"
      \new Voice = "Sop"  { \voiceOne \TimeKey \SopMusic }
      \new Voice = "Alto" { \voiceTwo \AltoMusic }
      \new Lyrics \lyricsto "Sop" { \VerseOne   }
      \new Lyrics \lyricsto "Sop" { \VerseTwo   }
      \new Lyrics \lyricsto "Sop" { \VerseThree }
      \new Lyrics \lyricsto "Sop" { \VerseFour  }
    >>
    \new Staff <<
      \clef "bass"
      \new Voice = "Tenor" { \voiceOne \TimeKey \TenorMusic }
      \new Voice = "Bass"  { \voiceTwo \BassMusic  }
    >>
  >>
}
```

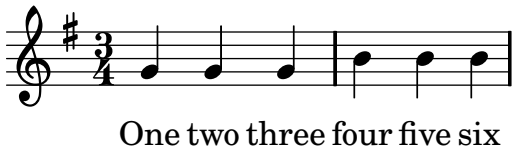


We end with an example to show how we might code a solo verse which continues into a two-part refrain in two staves. The positioning of the sequential and simultaneous sections to achieve this within a single score is quite tricky, so follow the explanation carefully!

Let's start with a score block containing a `ChoirStaff`, as we would like the brace to appear at the start of the chorus. Normally you would need angle brackets after `\new ChoirStaff` to bring in all the staves in parallel, but here we want to defer the parallelism during the solo so we use braces, although angle brackets here wouldn't hurt. Inside the `ChoirStaff` we want first the staff which will contain the verse. This must contain notes and lyrics in parallel, so here we need angle brackets around the `\new Voice` and `\new Lyrics` to start them at the same time:

```
versenotes = \relative c'' {
  \clef "treble"
  \key g \major
  \time 3/4 g g g b b b
}
versewords = \lyricmode {
  One two three four five six
```

```
}
\score {
  \new ChoirStaff {
    \new Staff <<
      \new Voice = "verse" {
        \versenotes \break
      }
      \new Lyrics \lyricsto verse {
        \versewords
      }
    >>
  }
}
```



That gives the verse line.

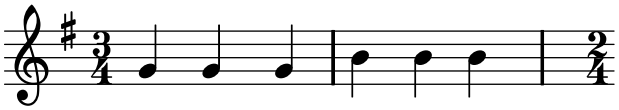
Now we want to continue with refrainA on the same staff while a second staff is introduced in parallel with it for refrainB, so this is a parallel section which must be positioned immediately following the `\break` in the verse Voice. Yes, *within* the verse Voice! Here's that parallel section. More staves could be introduced here in the same way.

```
<<
  \refrainnotesA
  \new Lyrics \lyricsto verse {
    \refrainwordsA
  }
  \new Staff <<
    \new Voice = "refrainB" {
      \refrainnotesB
    }
    \new Lyrics \lyricsto "refrainB" {
      \refrainwordsB
    }
  >>
>>
```

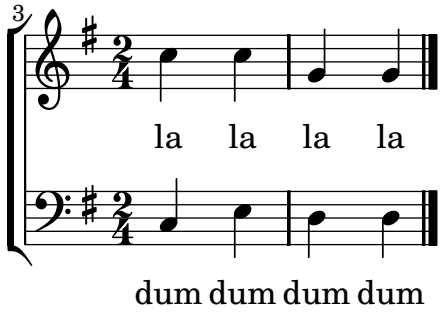
Here's the final result with two staves in the chorus showing how the parallel section is positioned within the verse Voice:

```
versenotes = \relative c'' {
  \clef "treble"
  \key g \major
  \time 3/4 g g g b b b
}
refrainnotesA = \relative c'' {
  \time 2/4
  c c g g \bar "|."
}
refrainnotesB = \relative c {
  \clef "bass"
```

```
  \key g \major
  c e d d
}
versewords = \lyricmode {
  One two three four five six
}
refrainwordsA = \lyricmode {
  la la la la
}
refrainwordsB = \lyricmode {
  dum dum dum dum
}
\score {
  \new ChoirStaff {
    \new Staff <<
      \new Voice = "verse" {
        \versenotes \break
        <<
          \refrainnotesA
          \new Lyrics \lyricsto "verse" {
            \refrainwordsA
          }
          \new Staff <<
            \new Voice = "refrainB" {
              \refrainnotesB
            }
            \new Lyrics \lyricsto "refrainB" {
              \refrainwordsB
            }
          >>
        >>
      }
      \new Lyrics \lyricsto "verse" {
        \versewords
      }
    >>
  }
}
```



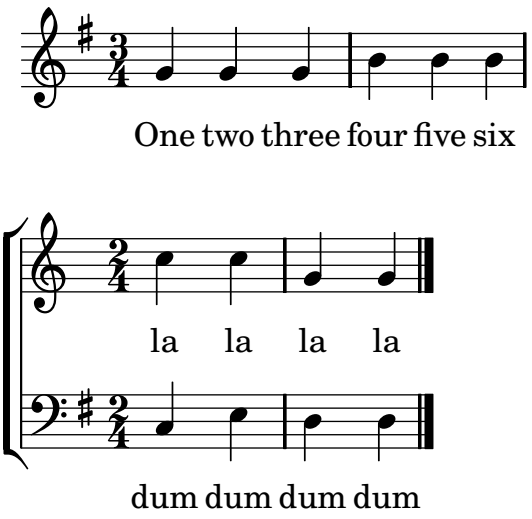
One two three four five six



However, although this is an interesting and useful exercise to help you to understand how sequential and simultaneous blocks work, in practice one would perhaps choose to code this as two `\score` blocks within an implicit `\book` block, as follows:

```
versenotes = \relative c'' {
  \clef "treble"
  \key g \major
  \time 3/4 g g g b b b
}
refrainnotesA = \relative c'' {
  \time 2/4
  c c g g \bar "|."
}
refrainnotesB = \relative c {
  \clef "bass"
  \key g \major
  c e d d
}
versewords = \lyricmode {
  One two three four five six
}
refrainwordsA = \lyricmode {
  la la la la
}
refrainwordsB = \lyricmode {
  dum dum dum dum
}
\score {
  \new Staff <<
    \new Voice = "verse" {
      \versenotes
    }
    \new Lyrics \lyricsto "verse" {
      \versewords
    }
  >>
}

\score {
  \new ChoirStaff <<
    \new Staff <<
      \new Voice = "refrainA" {
        \refrainnotesA
      }
      \new Lyrics \lyricsto "refrainA" {
```

```
        \refrainwordsA
      }
    >>
    \new Staff <<
      \new Voice = "refrainB" {
        \refrainnotesB
      }
      \new Lyrics \lyricsto "refrainB" {
        \refrainwordsB
      }
    >>
  >>
}
```



See also

Notation Reference: Section "Vocal music" in *Notation Reference*.

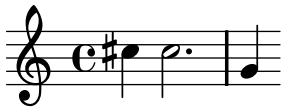
3.3 Contexts and engravers

Contexts and engravers have been mentioned informally in earlier sections; we now must look at these concepts in more detail, as they are important in the fine-tuning of LilyPond output.

3.3.1 Contexts explained

When music is printed, many notational elements which do not appear explicitly in the input file must be added to the output. For example, compare the input and output of the following example:

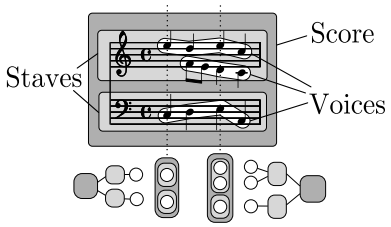
```
cis4 cis2. g4
```



The input is rather sparse, but in the output, bar lines, accidentals, clef, and time signature have been added. When LilyPond *interprets* the input the musical information is inspected in time order, similar to reading a score from left to right. While reading the input, the program

remembers where measure boundaries are, and which pitches require explicit accidentals. This information must be held on several levels. For example, the effect of an accidental is limited to a single staff, while a bar line must be synchronized across the entire score.

Within LilyPond, these rules and bits of information are grouped in *Contexts*. We have already met the `Voice` context. Others are the `Staff` and `Score` contexts. Contexts are hierarchical to reflect the hierarchical nature of a musical score. For example: a `Staff` context can contain many `Voice` contexts, and a `Score` context can contain many `Staff` contexts.



Each context has the responsibility for enforcing some notation rules, creating some notation objects and maintaining the associated properties. For example, the `Voice` context may introduce an accidental and then the `Staff` context maintains the rule to show or suppress the accidental for the remainder of the measure.

As another example, the synchronization of bar lines is, by default, handled in the `Score` context. However, in some music we may not want the bar lines to be synchronized – consider a polymetric score in 4/4 and 3/4 time. In such cases, we must modify the default settings of the `Score` and `Staff` contexts.

For very simple scores, contexts are created implicitly, and you need not be aware of them. For larger pieces, such as anything with more than one staff, they must be created explicitly to make sure that you get as many staves as you need, and that they are in the correct order. For typesetting pieces with specialized notation, it is usual to modify existing, or even to define totally new, contexts.

In addition to the `Score`, `Staff` and `Voice` contexts there are contexts which fit between the score and staff levels to control staff groups, such as the `PianoStaff` and `ChoirStaff` contexts. There are also alternative staff and voice contexts, and contexts for lyrics, percussion, fret boards, figured bass, etc.

The names of all context types are formed from one or more words, each word being capitalized and joined immediately to the preceding word with no hyphen or underscore, e.g., `GregorianTranscriptionStaff`.

See also

Notation Reference: Section "Contexts explained" in *Notation Reference*.

3.3.2 Creating contexts

There can be only one top level context: the `Score` context. This is created with the `\score` command, or, in simple scores, it is created automatically.

For scores with only one voice and one staff, the `Voice` and `Staff` contexts may be left to be created automatically, but for more complex scores it is necessary to create them by hand. The simplest command that does this is `\new`. It is prepended to a music expression, for example

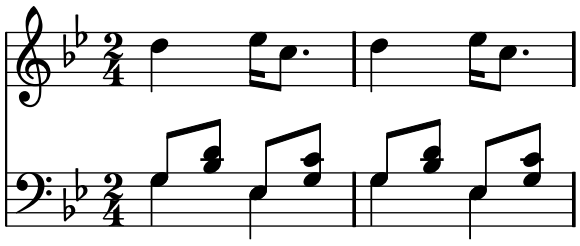
```
\new type music-expression
```

where *type* is a context name (like `Staff` or `Voice`). This command creates a new context, and starts interpreting the *music-expression* within that context.

Note that there is no `\new Score` command; the single top-level `Score` context is introduced with `\score`.

You have seen many practical examples which created new `Staff` and `Voice` contexts in earlier sections, but to remind you how these commands are used in practice, here's an annotated real-music example:

```
\score {  % start of single compound music expression
  <<  % start of simultaneous staves section
    \time 2/4
    \new Staff {  % create RH staff
      \key g \minor
      \clef "treble"
      \new Voice {  % create voice for RH notes
        \relative c'' {  % start of RH notes
          d4 ees16 c8. |
          d4 ees16 c8. |
        }  % end of RH notes
      }  % end of RH voice
    }  % end of RH staff
    \new Staff <<  % create LH staff; needs two simultaneous voices
      \key g \minor
      \clef "bass"
      \new Voice {  % create LH voice one
        \voiceOne
        \relative g {  % start of LH voice one notes
          g8 <bes d> ees, <g c> |
          g8 <bes d> ees, <g c> |
        }  % end of LH voice one notes
      }  % end of LH voice one
      \new Voice {  % create LH voice two
        \voiceTwo
        \relative g {  % start of LH voice two notes
          g4 ees |
          g4 ees |
        }  % end of LH voice two notes
      }  % end of LH voice two
    >>  % end of LH staff
  >>  % end of simultaneous staves section
}  % end of single compound music expression
```



(Note how all the statements which open a block with either a curly bracket, `{`, or double angle brackets, `<<`, are indented by two further spaces, and the corresponding closing bracket is indented by exactly the same amount. While this is not required, following this practice will greatly reduce the number of 'unmatched bracket' errors, and is strongly recommended. It enables the structure of the music to be seen at a glance, and any unmatched brackets will be obvious. Note too how the LH staff is created using double angle brackets because it requires two

voices for its music, whereas the RH staff is created with a single music expression surrounded by curly brackets because it requires only one voice.)

The `\new` command may also give a identifying name to the context to distinguish it from other contexts of the same type,

```
\new type = id music-expression
```

Note the distinction between the name of the context type, `Staff`, `Voice`, etc, and the identifying name of a particular instance of that type, which can be any sequence of letters invented by the user. Digits and spaces can also be used in the identifying name, but then it has to be placed in quotes, i.e. `\new Staff = "MyStaff 1" music-expression`. The identifying name is used to refer back to that particular instance of a context. We saw this in use in the section on lyrics, see Section 3.2.3 [Voices and vocals], page 52.

See also

Notation Reference: Section "Creating contexts" in *Notation Reference*.

3.3.3 Engravers explained

Every mark on the printed output of a score produced by LilyPond is produced by an `Engraver`. Thus there is an engraver to print staves, one to print note heads, one for stems, one for beams, etc, etc. In total there are over 120 such engravers! Fortunately, for most scores it is not necessary to know about more than a few, and for simple scores you do not need to know about any.

Engravers live and operate in Contexts. Engravers such as the `Metronome_mark_engraver`, whose action and output apply to the score as a whole, operate in the highest level context – the `Score` context.

The `Clef_engraver` and `Key_engraver` are to be found in every `Staff` Context, as different staves may require different clefs and keys.

The `Note_heads_engraver` and `Stem_engraver` live in every `Voice` context, the lowest level context of all.

Each engraver processes the particular objects associated with its function, and maintains the properties that relate to that function. These properties, like the properties associated with contexts, may be modified to change the operation of the engraver or the appearance of those elements in the printed score.

Engravers all have compound names formed from words which describe their function. Just the first word is capitalized, and the remainder are joined to it with underscores. Thus the `Staff_symbol_engraver` is responsible for creating the lines of the staff, the `Clef_engraver` determines and sets the pitch reference point on the staff by drawing a clef symbol.

Here are some of the most common engravers together with their function. You will see it is usually easy to guess the function from the name, or vice versa.

Engraver	**Function**
Accidental_engraver	Makes accidentals, cautionary and suggested accidentals
Beam_engraver	Engraves beams
Clef_engraver	Engraves clefs
Completion_heads_engraver	Splits notes which cross bar lines
New_dynamic_engraver	Creates hairpins and dynamic texts
Forbid_line_break_engraver	Prevents line breaks if a musical element is still active
Key_engraver	Creates the key signature
Metronome_mark_engraver	Engraves metronome marking
Note_heads_engraver	Engraves note heads

Rest_engraver	Engraves rests
Staff_symbol_engraver	Engraves the five (by default) lines of the staff
Stem_engraver	Creates stems and single-stem tremolos
Time_signature_engraver	Creates time signatures

We shall see later how the output of LilyPond can be changed by modifying the action of Engravers.

See also

Internals reference: Section "Engravers and Performers" in *Internals Reference*.

3.3.4 Modifying context properties

Contexts are responsible for holding the values of a number of context *properties*. Many of them can be changed to influence the interpretation of the input and so change the appearance of the output. They are changed by the `\set` command. This takes the form

```
\set ContextName.propertyName = #value
```

Where the *ContextName* is usually `Score`, `Staff` or `Voice`. It may be omitted, in which case `Voice` is assumed.

The names of context properties consist of words joined together with no hyphens or underscores, all except the first having a capital letter. Here are a few examples of some commonly used ones. There are many more.

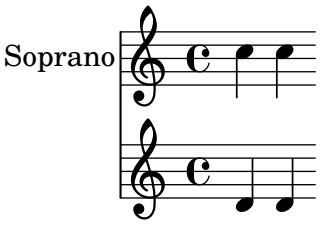
propertyName	**Type**	**Function**	**Example Value**
extraNatural	Boolean	If true, set extra natural signs before accidentals	`#t`, `#f`
currentBarNumber	Integer	Set the current bar number	`50`
doubleSlurs	Boolean	If true, print slurs both above and below notes	`#t`, `#f`
instrumentName	Text	Set the name to be placed at the start of the staff	`"Cello I"`
fontSize	Real	Increase or decrease the font size	`2.4`
stanza	Text	Set the text to print before the start of a verse	`"2"`

where a Boolean is either True (`#t`) or False (`#f`), an Integer is a positive whole number, a Real is a positive or negative decimal number, and text is enclosed in double apostrophes. Note the occurrence of hash signs, (`#`), in two different places – as part of the Boolean value before the `t` or `f`, and before *value* in the `\set` statement. So when a Boolean is being entered you need to code two hash signs, e.g., `##t`.

Before we can set any of these properties we need to know in which context they operate. Sometimes this is obvious, but occasionally it can be tricky. If the wrong context is specified, no error message is produced, but the expected action will not take place. For example, the `instrumentName` clearly lives in the `Staff` context, since it is the staff that is to be named. In this example the first staff is labelled, but not the second, because we omitted the context name.

```
<<
  \new Staff \relative c'' {
    \set Staff.instrumentName = #"Soprano"
    c4 c
 }
```

```
  \new Staff \relative c' {
  \set instrumentName = #"Alto"  % Wrong!
  d4 d
 }
>>
```



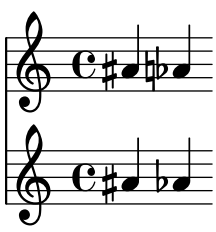
Remember the default context name is `Voice`, so the second `\set` command set the property `instrumentName` in the `Voice` context to "Alto", but as LilyPond does not look for any such property in the `Voice` context, no further action took place. This is not an error, and no error message is logged in the log file.

Similarly, if the property name is mis-spelt no error message is produced, and clearly the expected action cannot be performed. In fact, you can set any (fictitious) 'property' using any name you like in any context that exists by using the `\set` command. But if the name is not known to LilyPond it will not cause any action to be taken. Some text editors with special support for LilyPond input files document property names with bullets when you hover them with the mouse, like JEdit with LilyPondTool, or highlight unknown property names differently, like ConTEXT. If you do not use an editor with such features, it is recommended to check the property name in the Internals Reference: see Section "Tunable context properties" in *Internals Reference*, or Section "Contexts" in *Internals Reference*.

The `instrumentName` property will take effect only if it is set in the `Staff` context, but some properties can be set in more than one context. For example, the property `extraNatural` is by default set to ##t (true) for all staves. If it is set to ##f (false) in one particular `Staff` context it applies just to the accidentals on that staff. If it is set to false in the `Score` context it applies to all staves.

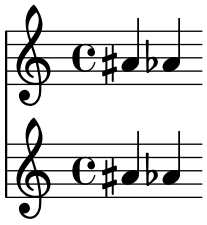
So this turns off extra naturals in one staff:

```
<<
  \new Staff \relative c'' {
    ais4 aes
 }
  \new Staff \relative c'' {
    \set Staff.extraNatural = ##f
    ais4 aes
 }
>>
```



and this turns them off in all staves:

```
<<
  \new Staff \relative c'' {
    ais4 aes
 }
  \new Staff \relative c'' {
    \set Score.extraNatural = ##f
    ais4 aes
 }
>>
```

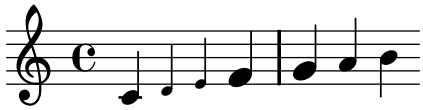


As another example, if `clefOctavation` is set in the `Score` context this immediately changes the value of the octavation in all current staves and sets a new default value which will be applied to all staves.

The opposite command, `\unset`, effectively removes the property from the context, which causes most properties to revert to their default value. Usually `\unset` is not required as a new `\set` command will achieve what is wanted.

The `\set` and `\unset` commands can appear anywhere in the input file and will take effect from the time they are encountered until the end of the score or until the property is `\set` or `\unset` again. Let's try changing the font size, which affects the size of the note heads (among other things) several times. The change is from the default value, not the most recently set value.

```
c4
% make note heads smaller
\set fontSize = #-4
d e
% make note heads larger
\set fontSize = #2.5
f g
% return to default size
\unset fontSize
a b
```



We have now seen how to set the values of several different types of property. Note that integers and numbers are always preceded by a hash sign, `#`, while a true or false value is specified by `##t` and `##f`, with two hash signs. A text property should be enclosed in double quotation signs, as above, although we shall see later that text can actually be specified in a much more general way by using the very powerful `markup` command.

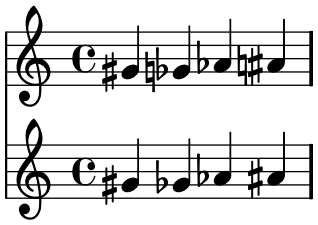
Setting context properties with \with

Context properties may also be set at the time the context is created. Sometimes this is a clearer way of specifying a property value if it is to remain fixed for the duration of the context. When a context is created with a `\new` command it may be followed immediately by a `\with { .. }` block in which the property values are set. For example, if we wish to suppress the printing of extra naturals for the duration of a staff we would write:

```
\new Staff \with { extraNatural = ##f }
```

like this:

```
<<
  \new Staff
  \relative c'' {
    gis ges aes ais
  }
  \new Staff \with { extraNatural = ##f }
  \relative c'' {
    gis ges aes ais
  }
>>
```



Properties set in this way may still be changed dynamically using `\set` and returned to their default value with `\unset`.

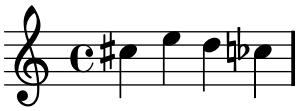
The `fontSize` property is treated differently. If this is set in a `\with` clause it effectively resets the default value of the font size. If it is later changed with `\set`, this new default value may be restored with the `\unset fontSize` command.

Setting context properties with \context

The values of context properties may be set in *all* contexts of a particular type, such as all `Staff` contexts, with a single command. The context type is identified by using its type name, like `Staff`, prefixed by a back-slash: `\Staff`. The statement which sets the property value is the same as that in a `\with` block, introduced above. It is placed in a `\context` block within a `\layout` block. Each `\context` block will affect all contexts of the type specified throughout the `\score` or `\book` block in which the `\layout` block appears. Here is a example to show the format:

```
\score {
  \new Staff {
    \relative c'' {
      cis4 e d ces
    }
  }
  \layout {
    \context {
      \Staff
      extraNatural = ##t
```

```
    }
  }
}
```



Context properties set in this way may be overridden for particular instances of contexts by statements in a `\with` block, and by `\set` commands embedded in music statements.

See also

Notation Reference: Section "Changing context default settings" in *Notation Reference*.

Internals Reference: Section "Contexts" in *Internals Reference*, Section "Tunable context properties" in *Internals Reference*.

3.3.5 Adding and removing engravers

We have seen that contexts each contain several engravers, each of which is responsible for producing a particular part of the output, like bar lines, staves, note heads, stems, etc. If an engraver is removed from a context, it can no longer produce its output. This is a crude way of modifying the output, but it can sometimes be useful.

Changing a single context

To remove an engraver from a single context we use the `\with` command placed immediately after the context creation command, as in the previous section.

As an illustration, let's repeat an example from the previous section with the staff lines removed. Remember that the staff lines are produced by the `Staff_symbol_engraver`.

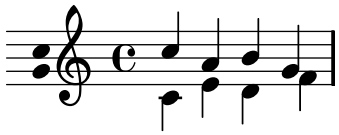
```
\new Staff \with {
  \remove Staff_symbol_engraver
}
\relative c' {
  c4
  \set fontSize = #-4  % make note heads smaller
  d e
  \set fontSize = #2.5  % make note heads larger
  f g
  \unset fontSize  % return to default size
  a b
}
```

Engravers can also be added to individual contexts. The command to do this is

`\consists` *`Engraver_name`*,

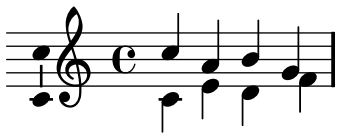
placed inside a `\with` block. Some vocal scores have an ambitus placed at the beginning of a staff to indicate the range of notes in that staff – see Section "ambitus" in *Music Glossary*. The ambitus is produced by the `Ambitus_engraver`, which is not normally included in any context. If we add it to the `Voice` context, it calculates the range from that voice only:

```
\new Staff <<
  \new Voice \with {
    \consists Ambitus_engraver
  }
  \relative c'' {
    \voiceOne
    c a b g
  }
  \new Voice
  \relative c' {
    \voiceTwo
    c e d f
  }
>>
```



but if we add the ambitus engraver to the `Staff` context, it calculates the range from all the notes in all the voices on that staff:

```
\new Staff \with {
    \consists Ambitus_engraver
  }
  <<
  \new Voice
  \relative c'' {
    \voiceOne
    c a b g
  }
  \new Voice
  \relative c' {
    \voiceTwo
    c e d f
  }
>>
```

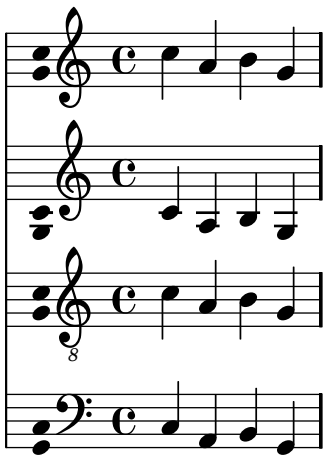


Changing all contexts of the same type

The examples above show how to remove or add engravers to individual contexts. It is also possible to remove or add engravers to every context of a specific type by placing the commands in the appropriate context in a `\layout` block. For example, if we wanted to show an ambitus for every staff in a four-staff score, we could write

```
\score {
  <<
    \new Staff <<
      \relative c'' { c a b g }
```

```
    >>
    \new Staff <<
      \relative c' { c a b g }
    >>
    \new Staff <<
      \clef "G_8"
      \relative c' { c a b g }
    >>
    \new Staff <<
      \clef "bass"
      \relative c { c a b g }
    >>
  >>
  \layout {
    \context {
      \Staff
      \consists Ambitus_engraver
    }
  }
}
```



The values of context properties may also be set for all contexts of a particular type by including the `\set` command in a `\context` block in the same way.

See also

Notation Reference: Section "Modifying context plug-ins" in *Notation Reference*, Section "Changing context default settings" in *Notation Reference*.

3.4 Extending the templates

You've read the tutorial, you know how to write music, you understand the fundamental concepts. But how can you get the staves that you want? Well, you can find lots of templates (see Appendix A [Templates], page 141) which may give you a start. But what if you want something that isn't covered there? Read on.

3.4.1 Soprano and cello

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

```
\version "2.12.3"
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4
  a4 b c d
}

text = \lyricmode {
  Aaa Bee Cee Dee
}

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

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

```
\version "2.12.3"
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4
  a4 b c d
}

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

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

If we simply cut and paste the `melody` section, we would end up with two `melody` definitions. This would not generate an error, but the second one would be used for both melodies. So let's rename them to make them distinct. We'll call the section for the soprano `sopranoMusic` and the section for the cello `celloMusic`. While we're doing this, let's rename `text` to be `sopranoLyrics`. Remember to rename both instances of all these names – both the initial definition (the `melody = \relative c' {` part) and the name's use (in the `\score` section).

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

```
\version "2.12.3"
sopranoMusic = \relative c' {
  \clef treble
  \key c \major
  \time 4/4
  a4 b c d
}

sopranoLyrics = \lyricmode {
  Aaa Bee Cee Dee
}

celloMusic = \relative c {
  \clef bass
  \key c \major
  \time 4/4
  d4 g fis8 e d4
}

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

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

```
\new Staff \celloMusic
```

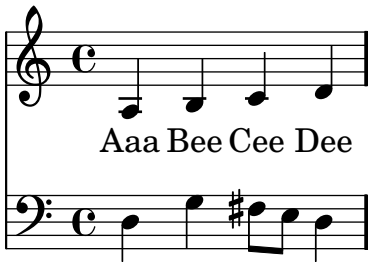
underneath the soprano stuff. We also need to add `<<` and `>>` around the music – that tells LilyPond that there's more than one thing (in this case, two `Staves`) happening at once. The `\score` looks like this now:

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

```
  \midi { }
}
```

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

```
\version "2.12.3"

sopranoMusic = \relative c' {
  \clef treble
  \key c \major
  \time 4/4
  a4 b c d
}

sopranoLyrics = \lyricmode {
  Aaa Bee Cee Dee
}

celloMusic = \relative c {
  \clef bass
  \key c \major
  \time 4/4
  d4 g fis8 e d4
}

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

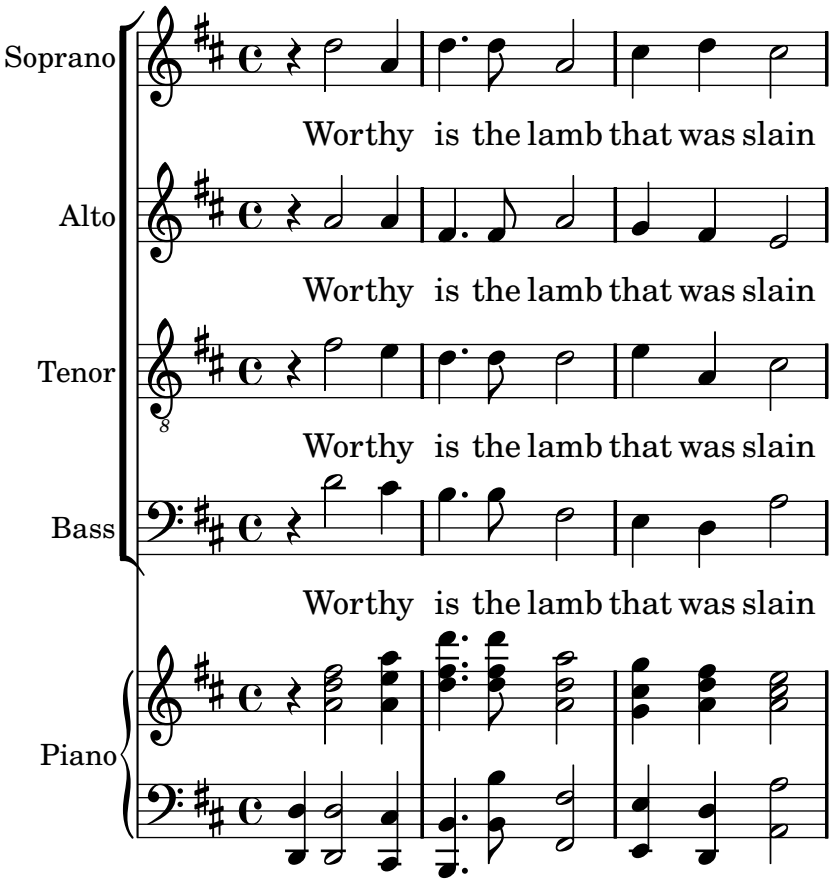


See also

The starting templates can be found in the 'Templates' appendix, see Section A.1 [Single staff], page 141.

3.4.2 Four-part SATB vocal score

Most vocal scores of music written for four-part mixed choir with orchestral accompaniment such as Mendelssohn's Elijah or Handel's Messiah have the choral music and words on four staves, one for each of SATB, with a piano reduction of the orchestral accompaniment underneath. Here's an example from Handel's Messiah:



None of the templates provides this layout exactly. The nearest is 'SATB vocal score and automatic piano reduction' – see Section A.4 [Vocal ensembles], page 152 – but we need to change the layout and add a piano accompaniment which is not derived automatically from the vocal parts. The variables holding the music and words for the vocal parts are fine, but we shall need to add variables for the piano reduction.

The order in which the contexts appear in the ChoirStaff of the template do not correspond with the order in the vocal score shown above. We need to rearrange them so there are four staves with the words written directly underneath the notes for each part. All the voices should be `\voiceOne`, which is the default, so the `\voiceXXX` commands should be removed. We also need to specify the tenor clef for the tenors. The way in which lyrics are specified in the template has not yet been encountered so we need to use the method with which we are familiar. We should also add the names of each staff.

Doing this gives for our ChoirStaff:

```
\new ChoirStaff <<
  \new Staff = "sopranos" <<
    \set Staff.instrumentName = #"Soprano"
    \new Voice = "sopranos" { \global \sopranoMusic }
  >>
  \new Lyrics \lyricsto "sopranos" { \sopranoWords }
```

```
        \new Staff = "altos" <<
          \set Staff.instrumentName = #"Alto"
          \new Voice = "altos" { \global \altoMusic }
        >>
        \new Lyrics \lyricsto "altos" { \altoWords }
        \new Staff = "tenors" <<
          \set Staff.instrumentName = #"Tenor"
          \new Voice = "tenors" { \global \tenorMusic }
        >>
        \new Lyrics \lyricsto "tenors" { \tenorWords }
        \new Staff = "basses" <<
          \set Staff.instrumentName = #"Bass"
          \new Voice = "basses" { \global \bassMusic }
        >>
        \new Lyrics \lyricsto "basses" { \bassWords }
      >>  % end ChoirStaff
```

Next we must work out the piano part. This is easy - we just pull out the piano part from the 'Solo piano' template:

```
\new PianoStaff <<
  \set PianoStaff.instrumentName = #"Piano  "
  \new Staff = "upper" \upper
  \new Staff = "lower" \lower
>>
```

and add the variable definitions for `upper` and `lower`.

The ChoirStaff and PianoStaff must be combined using angle brackets as we want them to be stacked one above the other:

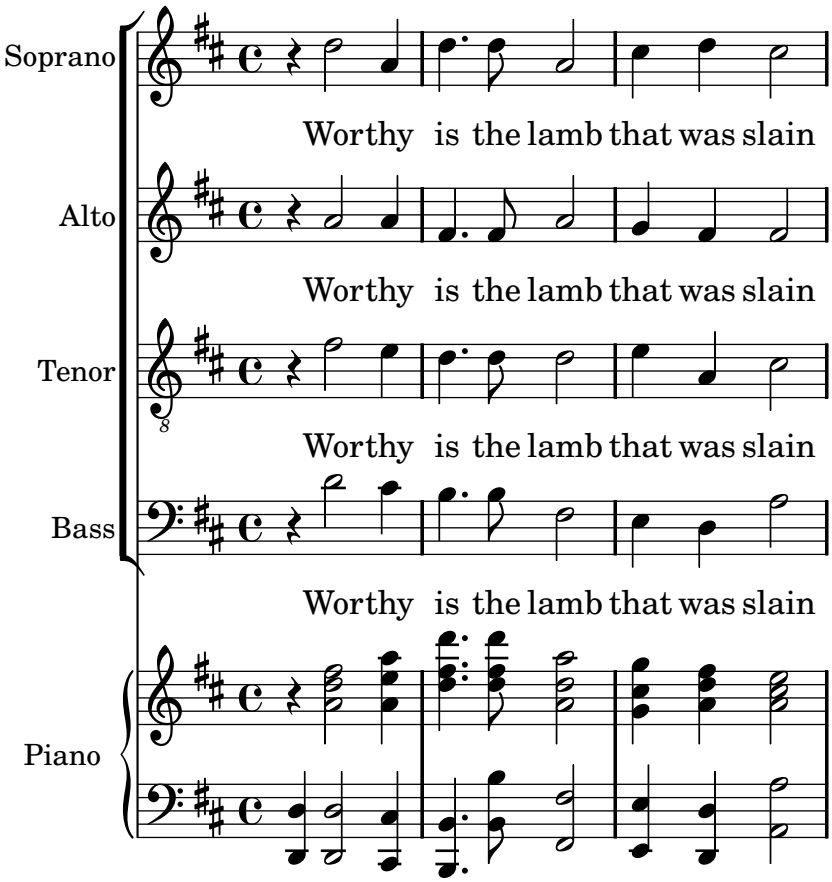
```
<<  % combine ChoirStaff and PianoStaff one above the other
  \new ChoirStaff <<
    \new Staff = "sopranos" <<
      \new Voice = "sopranos" { \global \sopranoMusic }
    >>
    \new Lyrics \lyricsto "sopranos" { \sopranoWords }
    \new Staff = "altos" <<
      \new Voice = "altos" { \global \altoMusic }
    >>
    \new Lyrics \lyricsto "altos" { \altoWords }
    \new Staff = "tenors" <<
      \clef "G_8"  % tenor clef
      \new Voice = "tenors" { \global \tenorMusic }
    >>
    \new Lyrics \lyricsto "tenors" { \tenorWords }
    \new Staff = "basses" <<
      \clef "bass"
      \new Voice = "basses" { \global \bassMusic }
    >>
    \new Lyrics \lyricsto "basses" { \bassWords }
  >>  % end ChoirStaff

  \new PianoStaff <<
    \set PianoStaff.instrumentName = #"Piano"
    \new Staff = "upper" \upper
```

```
    \new Staff = "lower" \lower
  >>
>>
```

Combining all these together and adding the music for the three bars of the example above gives:

```
\version "2.12.3"

global = { \key d \major \time 4/4 }
sopranoMusic = \relative c'' {
  \clef "treble"
  r4 d2 a4 | d4. d8 a2 | cis4 d cis2 |
}
sopranoWords = \lyricmode {
  Wor -- thy is the lamb that was slain
}
altoMusic = \relative a' {
  \clef "treble"
  r4 a2 a4 | fis4. fis8 a2 | g4 fis fis2 |
}
altoWords = \sopranoWords
tenorMusic = \relative c' {
  \clef "G_8"
  r4 fis2 e4 | d4. d8 d2 | e4 a, cis2 |
}
tenorWords = \sopranoWords
bassMusic = \relative c' {
  \clef "bass"
  r4 d2 cis4 | b4. b8 fis2 | e4 d a'2 |
}
bassWords = \sopranoWords
upper = \relative a' {
  \clef "treble"
  \global
  r4 <a d fis>2 <a e' a>4 |
  <d fis d'>4. <d fis d'>8 <a d a'>2 |
  <g cis g'>4 <a d fis> <a cis e>2 |
}
lower = \relative c, {
  \clef "bass"
  \global
  <d d'>4 <d d'>2 <cis cis'>4 |
  <b b'>4. <b' b'>8 <fis fis'>2 |
  <e e'>4 <d d'> <a' a'>2 |
}

\score {
  <<  % combine ChoirStaff and PianoStaff in parallel
    \new ChoirStaff <<
      \new Staff = "sopranos" <<
        \set Staff.instrumentName = #"Soprano"
        \new Voice = "sopranos" { \global \sopranoMusic }
      >>
```

```
      \new Lyrics \lyricsto "sopranos" { \sopranoWords }
      \new Staff = "altos" <<
        \set Staff.instrumentName = #"Alto"
        \new Voice = "altos" { \global \altoMusic }
      >>
      \new Lyrics \lyricsto "altos" { \altoWords }
      \new Staff = "tenors" <<
        \set Staff.instrumentName = #"Tenor"
        \new Voice = "tenors" { \global \tenorMusic }
      >>
      \new Lyrics \lyricsto "tenors" { \tenorWords }
      \new Staff = "basses" <<
        \set Staff.instrumentName = #"Bass"
        \new Voice = "basses" { \global \bassMusic }
      >>
      \new Lyrics \lyricsto "basses" { \bassWords }
    >>  % end ChoirStaff

    \new PianoStaff <<
      \set PianoStaff.instrumentName = #"Piano  "
      \new Staff = "upper" \upper
      \new Staff = "lower" \lower
    >>
  >>
}
```



3.4.3 Building a score from scratch

After gaining some facility with writing LilyPond code, you may find that it is easier to build a score from scratch rather than modifying one of the templates. You can also develop your own style this way to suit the sort of music you like. Let's see how to put together the score for an organ prelude as an example.

We begin with a header section. Here go the title, name of composer, etc, then come any variable definitions, and finally the score block. Let's start with these in outline and fill in the details later.

We'll use the first two bars of Bach's prelude based on *Jesu, meine Freude* which is written for two manuals and pedal organ. You can see these two bars of music at the bottom of this section. The top manual part has two voices, the lower and pedal organ one each. So we need four music definitions and one to define the time signature and key:

```
\version "2.12.3"
\header {
  title = "Jesu, meine Freude"
  composer = "J S Bach"
}
TimeKey = { \time 4/4 \key c \minor }
ManualOneVoiceOneMusic = {s1}
ManualOneVoiceTwoMusic = {s1}
ManualTwoMusic = {s1}
PedalOrganMusic = {s1}

\score {
}
```

For now we've just used a spacer note, `s1`, instead of the real music. We'll add that later.

Next let's see what should go in the score block. We simply mirror the staff structure we want. Organ music is usually written on three staves, one for each manual and one for the pedals. The manual staves should be bracketed together, so we need to use a PianoStaff for them. The first manual part needs two voices and the second manual part just one.

```
\new PianoStaff <<
  \new Staff = "ManualOne" <<
    \new Voice { \ManualOneVoiceOneMusic }
    \new Voice { \ManualOneVoiceTwoMusic }
  >>  % end ManualOne Staff context
  \new Staff = "ManualTwo" <<
    \new Voice { \ManualTwoMusic }
  >>  % end ManualTwo Staff context
>>  % end PianoStaff context
```

Next we need to add a staff for the pedal organ. This goes underneath the PianoStaff, but it must be simultaneous with it, so we need angle brackets around the two. Missing these out would generate an error in the log file. It's a common mistake which you'll make sooner or later! Try copying the final example at the end of this section, remove these angle brackets, and compile it to see what errors it generates.

```
<<  % PianoStaff and Pedal Staff must be simultaneous
  \new PianoStaff <<
    \new Staff = "ManualOne" <<
      \new Voice { \ManualOneVoiceOneMusic }
      \new Voice { \ManualOneVoiceTwoMusic }
    >>  % end ManualOne Staff context
```

```
    \new Staff = "ManualTwo" <<
      \new Voice { \ManualTwoMusic }
    >>  % end ManualTwo Staff context
  >>  % end PianoStaff context
  \new Staff = "PedalOrgan" <<
    \new Voice { \PedalOrganMusic }
  >>
>>
```

It is not necessary to use the simultaneous construct `<< .. >>` for the manual two staff and the pedal organ staff, since they contain only one music expression, but it does no harm, and always using angle brackets after `\new Staff` is a good habit to cultivate in case there are multiple voices. The opposite is true for Voices: these should habitually be followed by braces `{ .. }` in case your music is coded in several variables which need to run consecutively.

Let's add this structure to the score block, and adjust the indenting. We also add the appropriate clefs, ensure stems, ties and slurs in each voice on the upper staff point to the right direction with `\voiceOne` and `\voiceTwo`, and enter the time signature and key to each staff using our predefined variable, `\TimeKey`.

```
\score {
  <<  % PianoStaff and Pedal Staff must be simultaneous
    \new PianoStaff <<
      \new Staff = "ManualOne" <<
        \TimeKey  % set time signature and key
        \clef "treble"
        \new Voice { \voiceOne \ManualOneVoiceOneMusic }
        \new Voice { \voiceTwo \ManualOneVoiceTwoMusic }
      >>  % end ManualOne Staff context
      \new Staff = "ManualTwo" <<
        \TimeKey
        \clef "bass"
        \new Voice { \ManualTwoMusic }
      >>  % end ManualTwo Staff context
    >>  % end PianoStaff context
    \new Staff = "PedalOrgan" <<
      \TimeKey
      \clef "bass"
      \new Voice { \PedalOrganMusic }
    >>  % end PedalOrgan Staff
  >>
}  % end Score context
```

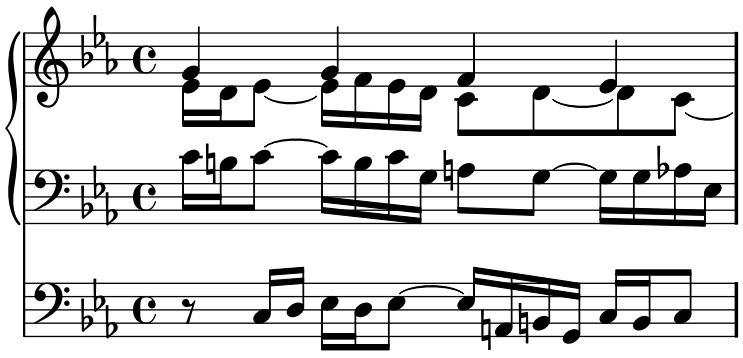
That completes the structure. Any three-staff organ music will have a similar structure, although the number of voices may vary. All that remains now is to add the music, and combine all the parts together.

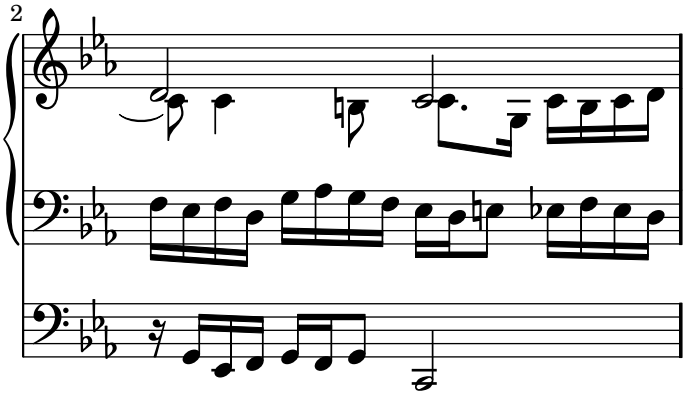
```
\version "2.12.3"

\header {
  title = "Jesu, meine Freude"
  composer = "J S Bach"
}
TimeKey = { \time 4/4 \key c \minor }
ManualOneVoiceOneMusic = \relative g' {
  g4 g f ees | d2 c2 |
```

```
}
ManualOneVoiceTwoMusic = \relative c' {
  ees16 d ees8~ ees16 f ees d c8 d~ d c~ |
  c c4 b8 c8. g16 c b c d |
}
ManualTwoMusic = \relative c' {
  c16 b c8~ c16 b c g a8 g~ g16 g aes ees |
  f ees f d g aes g f ees d e8~ ees16 f ees d |
}
PedalOrganMusic = \relative c {
  r8 c16 d ees d ees8~ ees16 a, b g c b c8 |
  r16 g ees f g f g8 c,2 |
  }

\score {
  <<  % PianoStaff and Pedal Staff must be simultaneous
    \new PianoStaff <<
      \new Staff = "ManualOne" <<
        \TimeKey  % set time signature and key
        \clef "treble"
        \new Voice { \voiceOne \ManualOneVoiceOneMusic }
        \new Voice { \voiceTwo \ManualOneVoiceTwoMusic }
      >>  % end ManualOne Staff context
      \new Staff = "ManualTwo" <<
        \TimeKey
        \clef "bass"
        \new Voice { \ManualTwoMusic }
      >>  % end ManualTwo Staff context
    >>  % end PianoStaff context
    \new Staff = "PedalOrgan" <<
      \TimeKey
      \clef "bass"
      \new Voice { \PedalOrganMusic }
    >>  % end PedalOrgan Staff context
  >>
}  % end Score context
```

Jesu, meine Freude

J S Bach




2

4 Tweaking output

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

4.1 Tweaking basics

4.1.1 Introduction to tweaks

'Tweaking' is a LilyPond term for the various methods available to the user for modifying the actions taken during interpretation of the input file and modifying the appearance of the printed output. Some tweaks are very easy to use; others are more complex. But taken together the methods available for tweaking permit almost any desired appearance of the printed music to be achieved.

In this section we cover the basic concepts required to understand tweaking. Later we give a variety of ready-made commands which can simply be copied to obtain the same effect in your own scores, and at the same time we show how these commands may be constructed so that you may learn how to develop your own tweaks.

Before starting on this Chapter you may wish to review the section Section 3.3 [Contexts and engravers], page 58, as Contexts, Engravers, and the Properties contained within them are fundamental to understanding and constructing Tweaks.

4.1.2 Objects and interfaces

Tweaking involves modifying the internal operation and structures of the LilyPond program, so we must first introduce some terms which are used to describe those internal operations and structures.

The term 'Object' is a generic term used to refer to the multitude of internal structures built by LilyPond during the processing of an input file. So when a command like `\new Staff` is encountered a new object of type `Staff` is constructed. That `Staff` object then holds all the properties associated with that particular staff, for example, its name and its key signature, together with details of the engravers which have been assigned to operate within that staff's context. Similarly, there are objects to hold the properties of all other contexts, such as `Voice` objects, `Score` objects, `Lyrics` objects, as well as objects to represent all notational elements such as bar lines, note heads, ties, dynamics, etc. Every object has its own set of property values.

Some types of object are given special names. Objects which represent items of notation on the printed output such as note heads, stems, slurs, ties, fingering, clefs, etc are called 'Layout objects', often known as 'Graphical Objects', or 'Grobs' for short. These are still objects in the generic sense above, and so they too all have properties associated with them, such as their position, size, color, etc.

Some layout objects are still more specialized. Phrasing slurs, crescendo hairpins, ottava marks, and many other grobs are not localized in a single place – they have a starting point, an ending point, and maybe other properties concerned with their shape. Objects with an extended shape like these are called 'Spanners'.

It remains to explain what 'Interfaces' are. Many objects, even though they are quite different, share common features which need to be processed in the same way. For example, all grobs have a color, a size, a position, etc, and all these properties are processed in the same way during LilyPond's interpretation of the input file. To simplify these internal operations these common actions and properties are grouped together in an object called a `grob-interface`. There are many other groupings of common properties like this, each one given a name ending

in `interface`. In total there are over 100 such interfaces. We shall see later why this is of interest and use to the user.

These, then, are the main terms relating to objects which we shall use in this chapter.

4.1.3 Naming conventions of objects and properties

We met some object naming conventions previously, in Section 3.3 [Contexts and engravers], page 58. Here for reference is a list of the most common object and property types together with the conventions for naming them and a couple of examples of some real names. We have used 'A' to stand for any capitalized alphabetic character and 'aaa' to stand for any number of lower-case alphabetic characters. Other characters are used verbatim.

Object/property type	**Naming convention**	**Examples**
Contexts	Aaaa or AaaaAaaaAaaa	Staff, GrandStaff
Layout Objects	Aaaa or AaaaAaaaAaaa	Slur, NoteHead
Engravers	Aaaa_aaa_engraver	Clef_engraver, Note_heads_engraver
Interfaces	aaa-aaa-interface	grob-interface, break-aligned-interface
Context Properties	aaa or aaaAaaaAaaa	alignAboveContext, skipBars
Layout Object Properties	aaa or aaa-aaa-aaa	direction, beam-thickness

As we shall see shortly, the properties of different types of object are modified by different commands, so it is useful to be able to recognize the type of object from the names of its properties.

4.1.4 Tweaking methods

`\override` command

We have already met the commands `\set` and `\with`, used to change the properties of **contexts** and to remove and add **engravers**, in Section 3.3.4 [Modifying context properties], page 62, and Section 3.3.5 [Adding and removing engravers], page 66. We now must meet some more important commands.

The command to change the properties of **layout objects** is `\override`. Because this command has to modify internal properties deep within LilyPond its syntax is not as simple as the commands you have met so far. It needs to know precisely which property of which object in which context has to be modified, and what its new value is to be. Let's see how this is done.

The general syntax of this command is:

```
\override Context.LayoutObject #'layout-property =
#value
```

This will set the property with the name *layout-property* of the layout object with the name *LayoutObject*, which is a member of the *Context* context, to the value *value*.

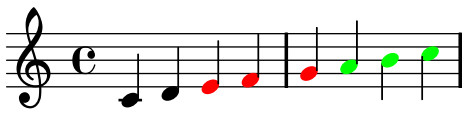
The *Context* can be omitted (and usually is) when the required context is unambiguously implied and is one of lowest level contexts, i.e., `Voice`, `ChordNames` or `Lyrics`, and we shall omit it in many of the following examples. We shall see later when it must be specified.

Later sections deal comprehensively with properties and their values, but to illustrate the format and use of these commands we shall use just a few simple properties and values which are easily understood.

For now, don't worry about the `#'`, which must precede the layout property, and the `#`, which must precede the value. These must always be present in exactly this form. This is the most common command used in tweaking, and most of the rest of this chapter will be directed to

presenting examples of how it is used. Here is a simple example to change the color of the note head:

```
c d
\override NoteHead #'color = #red
e f g
\override NoteHead #'color = #green
a b c
```



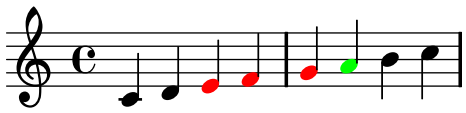
\revert command

Once overridden, the property retains its new value until it is overridden again or a `\revert` command is encountered. The `\revert` command has the following syntax and causes the value of the property to revert to its original default value; note, not its previous value if several `\override` commands have been issued.

```
\revert Context.LayoutObject #'layout-property
```

Again, just like *Context* in the `\override` command, *Context* is often not needed. It will be omitted in many of the following examples. Here we revert the color of the note head to the default value for the final two notes:

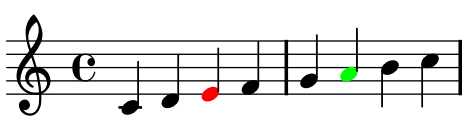
```
c d
\override NoteHead #'color = #red
e f g
\override NoteHead #'color = #green
a
\revert NoteHead #'color
b c
```



\once prefix

Both the `\override` and the `\set` commands may be prefixed by `\once`. This causes the following `\override` or `\set` command to be effective only during the current musical moment before the property reverts back to its default value. Using the same example, we can change the color of a single note like this:

```
c d
\once \override NoteHead #'color = #red
e f g
\once \override NoteHead #'color = #green
a b c
```



\overrideProperty command

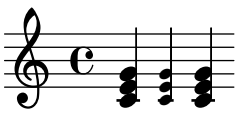
There is another form of the override command, `\overrideProperty`, which is occasionally required. We mention it here for completeness, but for details see Section "Difficult tweaks" in *Notation Reference*.

`\tweak` command

The final tweaking command which is available is `\tweak`. This should be used to change the properties of objects which occur at the same musical moment, such as the notes within a chord. Using `\override` would affect all the notes within a chord, whereas `\tweak` affects just the following item in the input stream.

Here's an example. Suppose we wish to change the size of the middle note head (the E) in a C major chord. Let's first see what `\once \override` would do:

```
<c e g>4
\once \override NoteHead #'font-size = #-3
<c e g>
<c e g>
```

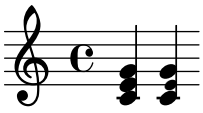


We see the override affects *all* the note heads in the chord. This is because all the notes of a chord occur at the same *musical moment*, and the action of `\once` is to apply the override to all layout objects of the type specified which occur at the same musical moment as the `\override` command itself.

The `\tweak` command operates in a different way. It acts on the immediately following item in the input stream. However, it is effective only on objects which are created directly from the input stream, essentially note heads and articulations; objects such as stems and accidentals are created later and cannot be tweaked in this way. Furthermore, when it is applied to note heads these *must* be within a chord, i.e., within single angle brackets, so to tweak a single note the `\tweak` command must be placed inside single angle brackets with the note.

So to return to our example, the size of the middle note of a chord would be changed in this way:

```
<c e g>4
<c \tweak #'font-size #-3 e g>4
```

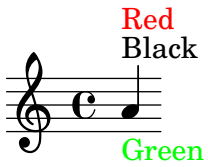


Note that the syntax of `\tweak` is different from that of the `\override` command. Neither the context nor the layout object should be specified; in fact, it would generate an error to do so. These are both implied by the following item in the input stream. Note also that an equals sign should not be present. So the general syntax of the `\tweak` command is simply

```
\tweak #'layout-property #value
```

A `\tweak` command can also be used to modify just one in a series of articulations, as shown here:

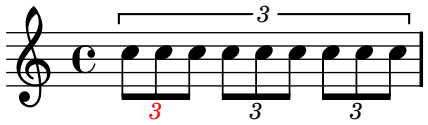
```
a ^Black
  -\tweak #'color #red ^Red
  -\tweak #'color #green _Green
```



Note that the `\tweak` command must be preceded by an articulation mark as if it were an articulation itself.

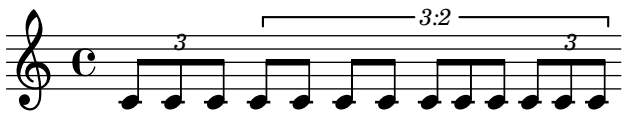
The `\tweak` command must also be used to change the appearance of one of a set of nested tuplets which begin at the same musical moment. In the following example, the long tuplet bracket and the first of the three short brackets begin at the same musical moment, so any `\override` command would apply to both of them. In the example, `\tweak` is used to distinguish between them. The first `\tweak` command specifies that the long tuplet bracket is to be placed above the notes and the second one specifies that the tuplet number is to be printed in red on the first short tuplet bracket.

```
\tweak #'direction #up
\times 4/3 {
  \tweak #'color #red
  \times 2/3 { c8[ c8 c8] }
  \times 2/3 { c8[ c8 c8] }
  \times 2/3 { c8[ c8 c8] }
}
```



If nested tuplets do not begin at the same moment, their appearance may be modified in the usual way with `\override` commands:

```
\times 2/3 { c8[ c c]}
\once \override TupletNumber
  #'text = #tuplet-number::calc-fraction-text
\times 2/3 {
  c[ c]
  c[ c]
  \once \override TupletNumber #'transparent = ##t
  \times 2/3 { c8[ c c] }
\times 2/3 { c8[ c c]}
}
```



See also

Notation Reference: Section "The tweak command" in *Notation Reference*.

4.2 The Internals Reference manual

4.2.1 Properties of layout objects

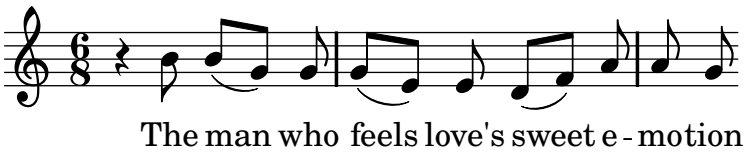
Suppose you have a slur in a score which, to your mind, appears too thin and you'd like to draw it a little heavier. How do you go about doing this? You know from the statements earlier about the flexibility of LilyPond that such a thing should be possible, and you would probably guess

that an `\override` command would be needed. But is there a heaviness property for a slur, and if there is, how might it be modified? This is where the Internals Reference manual comes in. It contains all the information you might need to construct this and all other `\override` commands.

Before we look at the Internals Reference a word of warning. This is a **reference** document, which means there is little or no explanation contained within it: its purpose is to present information precisely and concisely. This means it might look daunting at first sight. Don't worry! The guidance and explanation presented here will enable you to extract the information from the Internals Reference for yourself with just a little practice.

Let's use a concrete example with a simple fragment of real music:

```
{
  \time 6/8
  {
    r4 b8 b[( g]) g |
    g[( e]) e d[( f]) a |
    a g
  }
  \addlyrics {
    The man who feels love's sweet e -- mo -- tion
  }
}
```



Suppose now that we decide we would like the slurs to be a little heavier. Is this possible? The slur is certainly a layout object, so the question is, 'Is there a property belonging to a slur which controls the heaviness?' To answer this we must look in the Internals Reference, or IR for short.

The IR for the version of LilyPond you are using may be found on the LilyPond website at http://lilypond.org. Go to the documentation page and click on the Internals Reference link. For learning purposes you should use the standard HTML version, not the 'one big page' or the PDF. For the next few paragraphs to make sense you will need to actually do this as you read.

Under the heading **Top** you will see five links. Select the link to the *Backend*, which is where information about layout objects is to be found. There, under the heading **Backend**, select the link to *All layout objects*. The page that appears lists all the layout objects used in your version of LilyPond, in alphabetic order. Select the link to Slur, and the properties of Slurs are listed.

An alternative way of finding this page is from the Notation Reference. On one of the pages that deals with slurs you may find a link to the Internals Reference. This link will take you directly to this page, but if you have an idea about the name of the layout object to be tweaked, it is easier to go straight to the IR and search there.

This Slur page in the IR tells us first that Slur objects are created by the Slur_engraver. Then it lists the standard settings. Note these are **not** in alphabetic order. Browse down them looking for a property that might control the heaviness of slurs, and you should find

```
thickness (number)
     1.2
     Line thickness, generally measured in line-thickness
```

This looks a good bet to change the heaviness. It tells us that the value of `thickness` is a simple *number*, that the default value is 1.2, and that the units are in another property called `line-thickness`.

As we said earlier, there are few to no explanations in the IR, but we already have enough information to try changing the slur thickness. We see that the name of the layout object is `Slur`, that the name of the property to change is `thickness` and that the new value should be a number somewhat larger than 1.2 if we are to make slurs thicker.

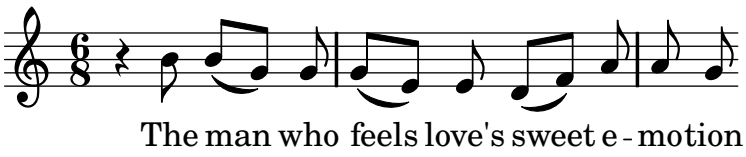
We can now construct the `\override` command by simply substituting the values we have found for the names, omitting the context. Let's use a very large value for the thickness at first, so we can be sure the command is working. We get:

```
\override Slur #'thickness = #5.0
```

Don't forget the `#'` preceding the property name and a `#` preceding the new value!

The final question is, 'Where should this command be placed?' While you are unsure and learning, the best answer is, 'Within the music, before the first slur and close to it.' Let's do that:

```
{
  \time 6/8
  {
    % Increase thickness of all following slurs from 1.2 to 5.0
    \override Slur #'thickness = #5.0
    r4 b8 b[( g]) g |
    g[( e]) e d[( f]) a |
    a g
  }
  \addlyrics {
    The man who feels love's sweet e -- mo -- tion
  }
}
```



and we see that the slur is indeed heavier.

So this is the basic way of constructing `\override` commands. There are a few more complications that we shall meet in later sections, but you now know all the essentials required to make up your own – but you will still need some practice. This is provided in the examples which follow.

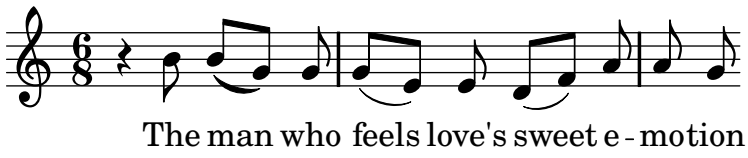
Finding the context

But first, what if we had needed to specify the Context? What should it be? We could guess that slurs are in the Voice context, as they are clearly closely associated with individual lines of music, but can we be sure? To find out, go back to the top of the IR page describing the Slur, where it says 'Slur objects are created by: Slur engraver'. So slurs will be created in whichever context the `Slur_engraver` is in. Follow the link to the `Slur_engraver` page. At the very bottom it tells us that `Slur_engraver` is part of five Voice contexts, including the standard voice context, `Voice`, so our guess was correct. And because `Voice` is one of the lowest level contexts which is implied unambiguously by the fact that we are entering notes, we can omit it in this location.

Overriding once only

As you can see, *all* the slurs are thicker in the final example above. But what if we wanted just the first slur to be thicker? This is achieved with the `\once` command. Placed immediately before the `\override` command it causes it to change only the slur which begins on the **immediately following** note. If the immediately following note does not begin a slur the command has no effect at all – it is not remembered until a slur is encountered, it is simply discarded. So the command with `\once` must be repositioned as follows:

```
{
  \time 6/8
  {
    r4 b8
    % Increase thickness of immediately following slur only
    \once \override Slur #'thickness = #5.0
    b[( g]) g |
    g[( e]) e d[( f]) a |
    a g
  }
  \addlyrics {
    The man who feels love's sweet e -- mo -- tion
  }
}
```



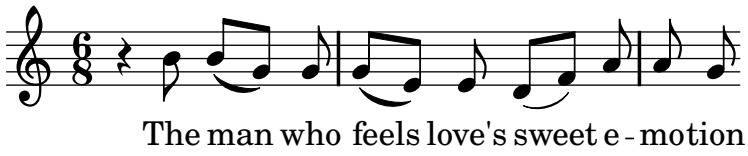
Now only the first slur is made heavier.

The `\once` command can also be used before the `\set` command.

Reverting

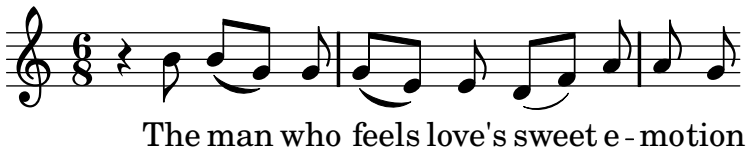
Finally, what if we wanted just the first two slurs to be heavier? Well, we could use two commands, each preceded by `\once` placed immediately before each of the notes where the slurs begin:

```
{
  \time 6/8
  {
    r4 b8
    % Increase thickness of immediately following slur only
    \once \override Slur #'thickness = #5.0
    b[( g]) g |
    % Increase thickness of immediately following slur only
    \once \override Slur #'thickness = #5.0
    g[( e]) e d[( f]) a |
    a g
  }
  \addlyrics {
    The man who feels love's sweet e -- mo -- tion
  }
}
```



or we could omit the `\once` command and use the `\revert` command to return the `thickness` property to its default value after the second slur:

```
{
  \time 6/8
  {
    r4 b8
    % Increase thickness of all following slurs from 1.2 to 5.0
    \override Slur #'thickness = #5.0
    b[( g]) g |
    g[( e])
    % Revert thickness of all following slurs to default of 1.2
    \revert Slur #'thickness
    e d[( f]) a |
    a g
  }
  \addlyrics {
    The man who feels love's sweet e -- mo -- tion
  }
}
```



The `\revert` command can be used to return any property changed with `\override` back to its default value. You may use whichever method best suits what you want to do.

That concludes our introduction to the IR, and the basic method of tweaking. Several examples follow in the later sections of this Chapter, partly to introduce you to some of the additional features of the IR, and partly to give you more practice in extracting information from it. These examples will contain progressively fewer words of guidance and explanation.

4.2.2 Properties found in interfaces

Suppose now that we wish to print the lyrics in italics. What form of `\override` command do we need to do this? We first look in the IR page listing 'All layout objects', as before, and look for an object that might control lyrics. We find `LyricText`, which looks right. Clicking on this shows the settable properties for lyric text. These include the `font-series` and `font-size`, but nothing that might give an italic shape. This is because the shape property is one that is common to all font objects, so, rather than including it in every layout object, it is grouped together with other similar common properties and placed in an **Interface**, the `font-interface`.

So now we need to learn how to find the properties of interfaces, and to discover what objects use these interface properties.

Look again at the IR page which describes LyricText. At the bottom of the page is a list of clickable interfaces which LyricText supports. The list has several items, including `font-interface`. Clicking on this brings up the properties associated with this interface, which are also properties of all the objects which support it, including LyricText.

Now we see all the user-settable properties which control fonts, including `font-shape(symbol)`, where `symbol` can be set to `upright`, `italics` or `caps`.

You will notice that `font-series` and `font-size` are also listed there. This immediately raises the question: Why are the common font properties `font-series` and `font-size` listed under `LyricText` as well as under the interface `font-interface` but `font-shape` is not? The answer is that `font-series` and `font-size` are changed from their global default values when a `LyricText` object is created, but `font-shape` is not. The entries in `LyricText` then tell you the values for those two properties which apply to `LyricText`. Other objects which support `font-interface` will set these properties differently when they are created.

Let's see if we can now construct the `\override` command to change the lyrics to italics. The object is `LyricText`, the property is `font-shape` and the value is `italic`. As before, we'll omit the context.

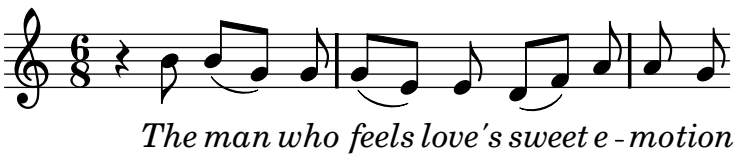
As an aside, although it is an important one, note that because the values of `font-shape` are symbols they must be introduced with a single apostrophe, `'`. That is why apostrophes are needed before `thickness` in the earlier example and `font-shape`. These are both symbols too. Symbols are then read internally by LilyPond. Some of them are the names of properties, like `thickness` or `font-shape`, others are used as values that can be given to properties, like `italic`. Note the distinction from arbitrary text strings, which would appear as `"a text string"`; for more details about symbols and strings, see Appendix B [Scheme tutorial], page 173.

Ok, so the `\override` command we need to print the lyrics in italics should be

```
\override LyricText #'font-shape = #'italic
```

and this should be placed just in front of and close to the lyrics which it should affect, like this:

```
{
  \time 6/8
  {
    r4 b8 b[( g]) g |
    g[( e]) e d[( f]) a |
    a g
  }
  \addlyrics {
    \override LyricText #'font-shape = #'italic
    The man who feels love's sweet e -- mo -- tion
  }
}
```



and the lyrics are all printed in italics.

Specifying the context in lyric mode

In the case of lyrics, if you try specifying the context in the format given earlier the command will fail. A syllable entered in lyricmode is terminated by either a space, a newline or a digit. All other characters are included as part of the syllable. For this reason a space or newline must appear before the terminating `}` to prevent it being included as part of the final syllable. Similarly, spaces must be inserted before and after the period or dot, '.', separating the context name from the object name, as otherwise the two names are run together and the interpreter cannot recognize them. So the command should be:

```
\override Lyrics . LyricText #'font-shape = #'italic
```

Note: In lyrics always leave whitespace between the final syllable and the terminating brace.

Note: In overrides in lyrics always place spaces around the dot between the context name and the object name.

See also

Learning Manual: Appendix B [Scheme tutorial], page 173.

4.2.3 Types of properties

So far we have seen two types of property: `number` and `symbol`. To be valid, the value given to a property must be of the correct type and obey the rules for that type. The type of property is always shown in brackets after the property name in the IR. Here is a list of the types you may need, together with the rules for that type, and some examples. You must always add a hash symbol, `#`, of course, to the front of these values when they are entered in the `\override` command.

Property type	**Rules**	**Examples**
Boolean	Either True or False, represented by #t or #f	`#t`, `#f`
Dimension (in staff space)	A positive decimal number (in units of staff space)	`2.5`, `0.34`
Direction	A valid direction constant or its numerical equivalent (decimal values between -1 and 1 are allowed)	`LEFT`, `CENTER`, `UP`, `1`, `-1`
Integer	A positive whole number	`3`, `1`
List	A set of values separated by spaces, enclosed in parentheses and preceded by an apostrophe	`'(left-edge staff-bar)`, `'(1)`, `'(1.0 0.25 0.5)`
Markup	Any valid markup	`\markup { \italic "cresc." }`
Moment	A fraction of a whole note constructed with the make-moment function	`(ly:make-moment 1 4)`, `(ly:make-moment 3 8)`
Number	Any positive or negative decimal value	`3.5`, `-2.45`
Pair (of numbers)	Two numbers separated by a 'space . space' and enclosed in brackets preceded by an apostrophe	`'(2 . 3.5)`, `'(0.1 . -3.2)`
Symbol	Any of the set of permitted symbols for that property, preceded by an apostrophe	`'italic`, `'inside`
Unknown	A procedure, or `#f` to cause no action	`bend::print`, `ly:text-interface::print`, `#f`
Vector	A list of three items enclosed in parentheses and preceded by apostrophe-hash, `'#`.	`'#(#t #t #f)`

See also

Learning Manual: Appendix B [Scheme tutorial], page 173.

4.3 Appearance of objects

Let us now put what we have learned into practice with a few examples which show how tweaks may be used to change the appearance of the printed music.

4.3.1 Visibility and color of objects

In the educational use of music we might wish to print a score with certain elements omitted as an exercise for the student, who is required to supply them. As a simple example, let us suppose the exercise is to supply the missing bar lines in a piece of music. But the bar lines are normally inserted automatically. How do we prevent them printing?

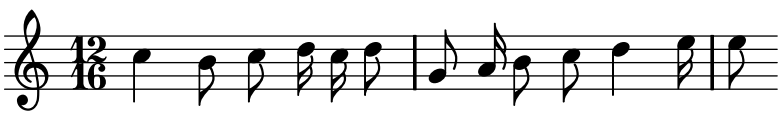
Before we tackle this, let us remember that object properties are grouped in what are called *interfaces* – see Section 4.2.2 [Properties found in interfaces], page 88. This is simply to group together those properties that may be used together to tweak a graphical object – if one of them is allowed for an object, so are the others. Some objects then use the properties in some interfaces, others use them from other interfaces. The interfaces which contain the properties used by a particular grob are listed in the IR at the bottom of the page describing that grob, and those properties may be viewed by looking at those interfaces.

We explained how to find information about grobs in Section 4.2.1 [Properties of layout objects], page 84. Using the same approach, we go to the IR to find the layout object which prints bar lines. Going via *Backend* and *All layout objects* we find there is a layout object called `BarLine`. Its properties include two that control its visibility: `break-visibility` and `stencil`. Barline also supports a number of interfaces, including the `grob-interface`, where we find the `transparent` and the `color` properties. All of these can affect the visibility of bar lines (and, of course, by extension, many other layout objects too.) Let's consider each of these in turn.

stencil

This property controls the appearance of the bar lines by specifying the symbol (glyph) which should be printed. In common with many other properties, it can be set to print nothing by setting its value to `#f`. Let's try it, as before, omitting the implied Context, `Voice`:

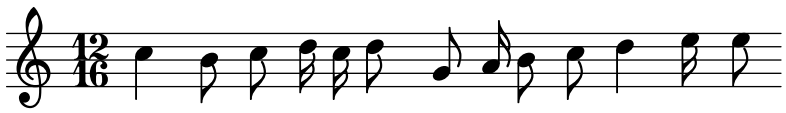
```
{
  \time 12/16
  \override BarLine #'stencil = ##f
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```



The bar lines are still printed. What is wrong? Go back to the IR and look again at the page giving the properties of BarLine. At the top of the page it says "Barline objects are created by: Bar_engraver". Go to the `Bar_engraver` page. At the bottom it gives a list of Contexts in which the bar engraver operates. All of them are of the type `Staff`, so the reason the `\override` command failed to work as expected is because `Barline` is not in the default `Voice` context. If the context is specified wrongly, the command simply does not work. No error message is produced, and nothing is logged in the log file. Let's try correcting it by adding the correct context:

```
{
  \time 12/16
```

```
  \override Staff.BarLine #'stencil = ##f
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```

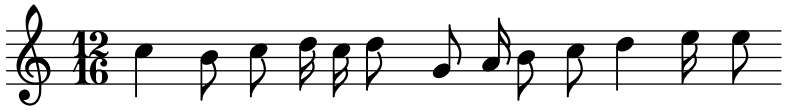


Now the bar lines have vanished.

break-visibility

We see from the `BarLine` properties in the IR that the `break-visibility` property requires a vector of three booleans. These control respectively whether bar lines are printed at the end of a line, in the middle of lines, and at the beginning of lines. For our example we want all bar lines to be suppressed, so the value we need is `'#(#f #f #f)`. Let's try that, remembering to include the `Staff` context. Note also that in writing this value we have `#'#` before the opening bracket. The `'#` is required as part of the value to introduce a vector, and the first `#` is required, as always, to precede the value itself in the `\override` command.

```
{
  \time 12/16
  \override Staff.BarLine #'break-visibility = #'#(#f #f #f)
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```

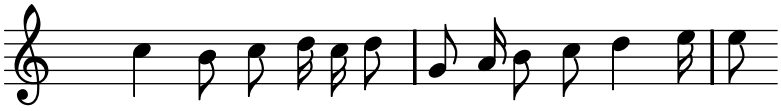


And we see this too removes all the bar lines.

transparent

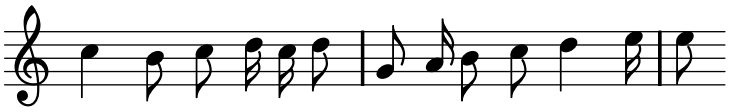
We see from the properties specified in the `grob-interface` page in the IR that the `transparent` property is a boolean. This should be set to `#t` to make the grob transparent. In this next example let us make the time signature invisible rather than the bar lines. To do this we need to find the grob name for the time signature. Back to the 'All layout objects' page in the IR to find the properties of the `TimeSignature` layout object. This is produced by the `Time_signature_engraver` which you can check also lives in the `Staff` context and also supports the `grob-interface`. So the command to make the time signature transparent is:

```
{
  \time 12/16
  \override Staff.TimeSignature #'transparent = ##t
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```



The time signature is gone, but this command leaves a gap where the time signature should be. Maybe this is what is wanted for an exercise for the student to fill it in, but in other circumstances a gap might be undesirable. To remove it, the stencil for the time signature should be set to `#f` instead:

```
{
  \time 12/16
  \override Staff.TimeSignature #'stencil = ##f
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```



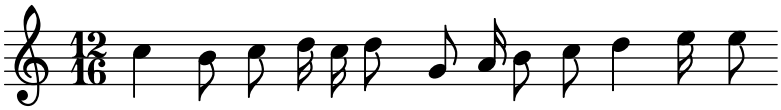
and the difference is obvious: setting the stencil to `#f` removes the object entirely; making the object `transparent` leaves it where it is, but makes it invisible.

color

Finally let us try making the bar lines invisible by coloring them white. (There is a difficulty with this in that the white bar line may or may not blank out the staff lines where they cross. You may see in some of the examples below that this happens unpredictably. The details of why this is so and how to control it are covered in Section "Painting objects white" in *Notation Reference*. But at the moment we are learning about color, so please just accept this limitation for now.)

The `grob-interface` specifies that the color property value is a list, but there is no explanation of what that list should be. The list it requires is actually a list of values in internal units, but, to avoid having to know what these are, several ways are provided to specify colors. The first way is to use one of the 'normal' colors listed in the first table in Section "List of colors" in *Notation Reference*. To set the bar lines to white we write:

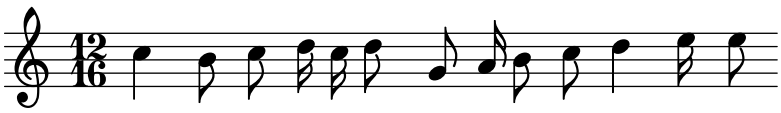
```
{
  \time 12/16
  \override Staff.BarLine #'color = #white
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```



and again, we see the bar lines are not visible. Note that *white* is not preceded by an apostrophe – it is not a symbol, but a *function*. When called, it provides the list of internal values required to set the color to white. The other colors in the normal list are functions too. To convince yourself this is working you might like to change the color to one of the other functions in the list.

The second way of changing the color is to use the list of X11 color names in the second list in Section "List of colors" in *Notation Reference*. However, these must be preceded by another function, which converts X11 color names into the list of internal values, `x11-color`, like this:

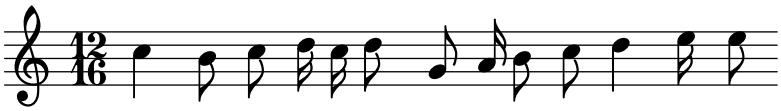
```
{
  \time 12/16
  \override Staff.BarLine #'color = #(x11-color 'white)
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```



Note that in this case the function `x11-color` takes a symbol as an argument, so the symbol must be preceded by an apostrophe and the two enclosed in brackets.

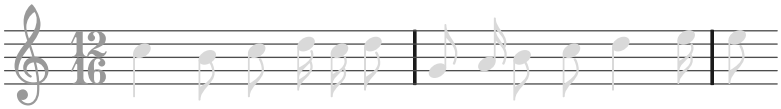
There is yet a third function, one which converts RGB values into internal colors – the `rgb-color` function. This takes three arguments giving the intensities of the red, green and blue colors. These take values in the range 0 to 1. So to set the color to red the value should be `(rgb-color 1 0 0)` and to white it should be `(rgb-color 1 1 1)`:

```
{
  \time 12/16
  \override Staff.BarLine #'color = #(rgb-color 1 1 1)
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```



Finally, there is also a grey scale available as part of the X11 set of colors. These range from black, `'grey0'`, to white, `'grey100`, in steps of 1. Let's illustrate this by setting all the layout objects in our example to various shades of grey:

```
{
  \time 12/16
  \override Staff.StaffSymbol   #'color = #(x11-color 'grey30)
  \override Staff.TimeSignature #'color = #(x11-color 'grey60)
  \override Staff.Clef          #'color = #(x11-color 'grey60)
  \override Voice.NoteHead      #'color = #(x11-color 'grey85)
  \override Voice.Stem          #'color = #(x11-color 'grey85)
  \override Staff.BarLine       #'color = #(x11-color 'grey10)
  c4 b8 c d16 c d8 |
  g, a16 b8 c d4 e16 |
  e8
}
```

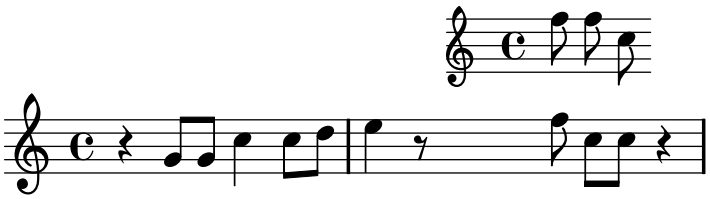


Note the contexts associated with each of the layout objects. It is important to get these right, or the commands will not work! Remember, the context is the one in which the appropriate engraver is placed. The default context for engravers can be found by starting from the layout object, going from there to the engraver which produces it, and on the engraver page in the IR it tells you in which context the engraver will normally be found.

4.3.2 Size of objects

Let us begin by looking again at the earlier example see Section 3.1.3 [Nesting music expressions], page 41) which showed how to introduce a new temporary staff, as in an Section "ossia" in *Music Glossary*.

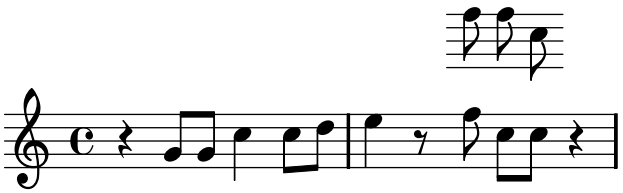
```
\new Staff ="main" {
       \relative g' {
         r4 g8 g c4 c8 d |
         e4 r8
         <<
           { f c c }
           \new Staff \with {
             alignAboveContext = #"main" }
           { f8 f c }
         >>
         r4 |
       }
     }
```



Ossia are normally written without clef and time signature, and are usually printed slightly smaller than the main staff. We already know now how to remove the clef and time signature – we simply set the stencil of each to `#f`, as follows:

```
\new Staff ="main" {
  \relative g' {
    r4 g8 g c4 c8 d |
    e4 r8
    <<
      { f c c }
      \new Staff \with {
        alignAboveContext = #"main"
      }
      {
        \override Staff.Clef #'stencil = ##f
        \override Staff.TimeSignature #'stencil = ##f
        { f8 f c }
      }
    >>
    r4 |
  }
```

```
}
```



where the extra pair of braces after the `\with` clause are required to ensure the enclosed overrides and music are applied to the ossia staff.

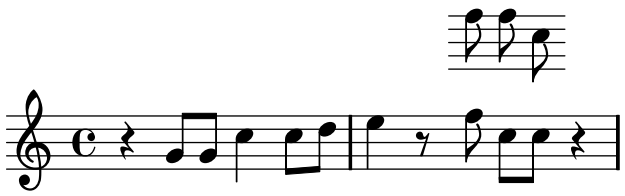
But what is the difference between modifying the staff context by using `\with` and modifying the stencils of the clef and the time signature with \override? The main difference is that changes made in a `\with` clause are made at the time the context is created, and remain in force as the **default** values for the duration of that context, whereas `\set` or `\override` commands embedded in the music are dynamic – they make changes synchronized with a particular point in the music. If changes are unset or reverted using `\unset` or `\revert` they return to their default values, which will be the ones set in the `\with` clause, or if none have been set there, the normal default values.

Some context properties can be modified only in `\with` clauses. These are those properties which cannot sensibly be changed after the context has been created. `alignAboveContext` and its partner, `alignBelowContext`, are two such properties – once the staff has been created its alignment is decided and it would make no sense to try to change it later.

The default values of layout object properties can also be set in `\with` clauses. Simply use the normal `\override` command leaving out the context name, since this is unambiguously defined as the context which the `\with` clause is modifying. If fact, an error will be generated if a context is specified in this location.

So we could replace the example above with

```
\new Staff ="main" {
  \relative g' {
    r4 g8 g c4 c8 d |
    e4 r8
    <<
      { f c c }
      \new Staff \with {
        alignAboveContext = #"main"
        % Don't print clefs in this staff
        \override Clef #'stencil = ##f
        % Don't print time signatures in this staff
        \override TimeSignature #'stencil = ##f
      }
        { f8 f c }
    >>
    r4 |
  }
}
```



Finally we come to changing the size of layout objects.

Some layout objects are created as glyphs selected from a typeface font. These include note heads, accidentals, markup, clefs, time signatures, dynamics and lyrics. Their size is changed by modifying the `font-size` property, as we shall shortly see. Other layout objects such as slurs and ties – in general, spanner objects – are drawn individually, so there is no `font-size` associated with them. These objects generally derive their size from the objects to which they are attached, so usually there is no need to change their size manually. Still other properties such as the length of stems and bar lines, thickness of beams and other lines, and the separation of staff lines all need to be modified in special ways.

Returning to the ossia example, let us first change the font-size. We can do this in two ways. We can either change the size of the fonts of each object type, like `NoteHead`s with commands like

```
\override NoteHead #'font-size = #-2
```

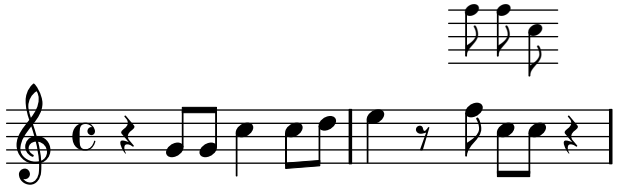
or we can change the size of all fonts by setting a special property, `fontSize`, using `\set`, or by including it in a `\with` clause (but without the `\set`).

```
\set fontSize = #-2
```

Both of these statements would cause the font size to be reduced by 2 steps from its previous value, where each step reduces or increases the size by approximately 12%.

Let's try it in our ossia example:

```
\new Staff ="main" {
  \relative g' {
    r4 g8 g c4 c8 d |
    e4 r8
    <<
      { f c c }
      \new Staff \with {
        alignAboveContext = #"main"
        \override Clef #'stencil = ##f
        \override TimeSignature #'stencil = ##f
        % Reduce all font sizes by ~24%
        fontSize = #-2
      }
        { f8 f c }
    >>
    r4 |
  }
}
```



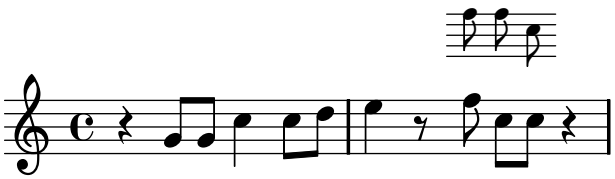
This is still not quite right. The note heads and flags are smaller, but the stems are too long in proportion and the staff lines are spaced too widely apart. These need to be scaled down in proportion to the font reduction. The next sub-section discusses how this is done.

4.3.3 Length and thickness of objects

Distances and lengths in LilyPond are generally measured in staff-spaces, the distance between adjacent lines in the staff, (or occasionally half staff spaces) while most `thickness` properties are measured in units of an internal property called `line-thickness`. For example, by default, the lines of hairpins are given a thickness of 1 unit of `line-thickness`, while the `thickness` of a note stem is 1.3. Note, though, that some thickness properties are different; for example, the thickness of beams is measured in staff-spaces.

So how are lengths to be scaled in proportion to the font size? This can be done with the help of a special function called `magstep` provided for exactly this purpose. It takes one argument, the change in font size (#-2 in the example above) and returns a scaling factor suitable for reducing other objects in proportion. It is used like this:

```
\new Staff ="main" {
  \relative g' {
    r4 g8 g c4 c8 d |
    e4 r8
    <<
      { f c c }
      \new Staff \with {
        alignAboveContext = #"main"
        \override Clef #'stencil = ##f
        \override TimeSignature #'stencil = ##f
        fontSize = #-2
        % Reduce stem length and line spacing to match
        \override StaffSymbol #'staff-space = #(magstep -2)
      }
        { f8 f c }
    >>
    r4 |
  }
}
```



Since the length of stems and many other length-related properties are always calculated relative to the value of the `staff-space` property these are automatically scaled down in length too. Note that this affects only the vertical scale of the ossia – the horizontal scale is determined by the layout of the main music in order to remain synchronized with it, so it is not affected by any of these changes in size. Of course, if the scale of all the main music were changed in this way then the horizontal spacing would be affected. This is discussed later in the layout section.

This, then, completes the creation of an ossia. The sizes and lengths of all other objects may be modified in analogous ways.

For small changes in scale, as in the example above, the thickness of the various drawn lines such as bar lines, beams, hairpins, slurs, etc does not usually require global adjustment. If

the thickness of any particular layout object needs to be adjusted this can be best achieved by overriding its `thickness` property. An example of changing the thickness of slurs was shown above in Section 4.2.1 [Properties of layout objects], page 84. The thickness of all drawn objects (i.e., those not produced from a font) may be changed in the same way.

4.4 Placement of objects

4.4.1 Automatic behavior

There are some objects in musical notation that belong to the staff and there are other objects that should be placed outside the staff. These are called within-staff objects and outside-staff objects respectively.

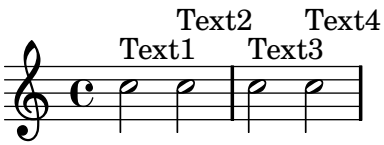
Within-staff objects are those that are located on the staff – note heads, stems, accidentals, etc. The positions of these are usually fixed by the music itself – they are vertically positioned on specific lines of the staff or are tied to other objects that are so positioned. Collisions of note heads, stems and accidentals in closely set chords are normally avoided automatically. There are commands and overrides which can modify this automatic behavior, as we shall shortly see.

Objects belonging outside the staff include things such as rehearsal marks, text and dynamic markings. LilyPond's rule for the vertical placement of outside-staff objects is to place them as close to the staff as possible but not so close that they collide with any other object. LilyPond uses the `outside-staff-priority` property to determine the order in which the objects should be placed, as follows.

First, LilyPond places all the within-staff objects. Then it sorts the outside-staff objects according to their `outside-staff-priority`. The outside-staff objects are taken one by one, beginning with the object with the lowest `outside-staff-priority`, and placed so that they do not collide with any objects that have already been placed. That is, if two outside-staff grobs are competing for the same space, the one with the lower `outside-staff-priority` will be placed closer to the staff. If two objects have the same `outside-staff-priority` the one encountered first will be placed closer to the staff.

In the following example all the markup texts have the same priority (since it is not explicitly set). Note that 'Text3' is automatically positioned close to the staff again, nestling under 'Text2'.

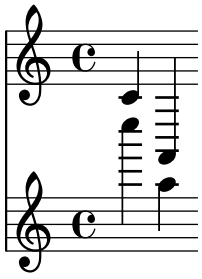
```
c2^"Text1"
c^"Text2"
c^"Text3"
c^"Text4"
```



Staves are also positioned, by default, as closely together as possible (subject to a minimum separation). If notes project a long way towards an adjacent staff they will force the staves further apart only if an overlap of the notation would otherwise occur. The following example demonstrates this 'nestling' of the notes on adjacent staves:

```
<<
  \new Staff {
    \relative c' { c a, }
  }
  \new Staff {
    \relative c'''' { c a, }
```

```
  }
>>
```



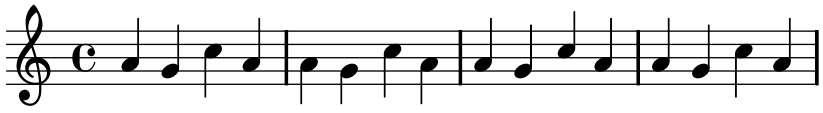
4.4.2 Within-staff objects

We have already seen how the commands `\voiceXXX` affect the direction of slurs, ties, fingering and everything else which depends on the direction of the stems. These commands are essential when writing polyphonic music to permit interweaving melodic lines to be distinguished. But occasionally it may be necessary to override this automatic behavior. This can be done for whole sections of music or even for an individual note. The property which controls this behavior is the `direction` property of each layout object. We first explain what this does, and then introduce a number of ready-made commands which avoid your having to code explicit overrides for the more common modifications.

Some layout objects like slurs and ties curve, bend or point either up or down; others like stems and flags also move to right or left when they point up or down. This is controlled automatically when `direction` is set.

The following example shows in bar 1 the default behavior of stems, with those on high notes pointing down and those on low notes pointing up, followed by four notes with all stems forced down, four notes with all stems forced up, and finally four notes reverted back to the default behavior.

```
a4 g c a
\override Stem #'direction = #DOWN
a g c a
\override Stem #'direction = #UP
a g c a
\revert Stem #'direction
a g c a
```



Here we use the constants `DOWN` and `UP`. These have the values `-1` and `+1` respectively, and these numerical values may be used instead. The value `0` may also be used in some cases. It is simply treated as meaning `UP` for stems, but for some objects it means 'center'. There is a constant, `CENTER` which has the value `0`.

However, these explicit overrides are not usually used, as there are simpler equivalent predefined commands available. Here is a table of the commonest. The meaning of each is stated where it is not obvious.

Down/Left	**Up/Right**	**Revert**	**Effect**
`\arpeggioArrowDown`	`\arpeggioArrowUp`	`\arpeggioNormal`	Arrow is at bottom, at top, or no arrow

`\dotsDown`	`\dotsUp`	`\dotsNeutral`	Direction of movement to avoid staff lines
`\dynamicDown`	`\dynamicUp`	`\dynamicNeutral`	
`\phrasingSlurDown`	`\phrasingSlurUp`	`\phrasingSlurNeutral`	Note: distinct from slur commands
`\slurDown`	`\slurUp`	`\slurNeutral`	
`\stemDown`	`\stemUp`	`\stemNeutral`	
`\textSpannerDown`	`\textSpannerUp`	`\textSpannerNeutral`	Text entered as spanner is below/above staff
`\tieDown`	`\tieUp`	`\tieNeutral`	
`\tupletDown`	`\tupletUp`	`\tupletNeutral`	Tuplets are below/above notes

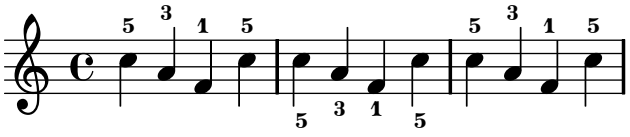
Note that these predefined commands may **not** be preceded by `\once`. If you wish to limit the effect to a single note you must either use the equivalent `\once \override` command or use the predefined command followed after the affected note by the corresponding `\xxxNeutral` command.

Fingering

The placement of fingering on single notes can also be controlled by the `direction` property, but changing `direction` has no effect on chords. As we shall see, there are special commands which allow the fingering of individual notes of chords to be controlled, with the fingering being placed above, below, to the left or to the right of each note.

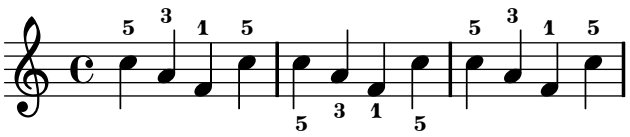
First, here's the effect of `direction` on the fingering attached to single notes. The first bar shows the default behaviour, and the following two bars shows the effect of specifying `DOWN` and `UP`:

```
c-5 a-3 f-1 c'-5
\override Fingering #'direction = #DOWN
c-5 a-3 f-1 c'-5
\override Fingering #'direction = #UP
c-5 a-3 f-1 c'-5
```



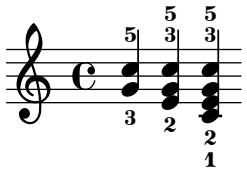
However, overriding the `direction` property is not the easiest way of manually setting the fingering above or below the notes; using `_` or `^` instead of `-` before the fingering number is usually preferable. Here is the previous example using this method:

```
c-5 a-3 f-1 c'-5
c_5 a_3 f_1 c'_5
c^5 a^3 f^1 c'^5
```



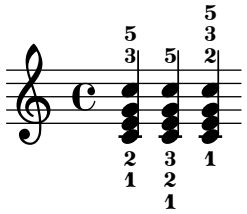
The `direction` property is ignored for chords, but the directional prefixes, `_` and `^` do work. By default, the fingering is automatically placed both above and below the notes of a chord, as shown:

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



but this may be overriden to manually force all or any of the individual fingering numbers above or below:

```
<c-5 g-3 e-2 c-1>
<c^5 g_3 e_2 c_1>
<c^5 g^3 e^2 c_1>
```



Even greater control over the placement of fingering of the individual notes in a chord is possible by using the `\set fingeringOrientations` command. The format of this command is:

```
\set fingeringOrientations = #'([up] [left/right] [down])
```

`\set` is used because `fingeringOrientations` is a property of the `Voice` context, created and used by the `New_fingering_engraver`.

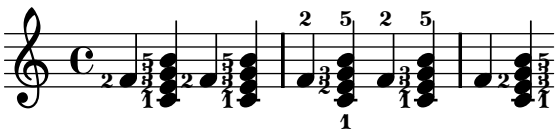
The property may be set to a list of one to three values. It controls whether fingerings may be placed above (if `up` appears in the list), below (if `down` appears), to the left (if `left` appears, or to the right (if `right` appears). Conversely, if a location is not listed, no fingering is placed there. LilyPond takes these constraints and works out the best placement for the fingering of the notes of the following chords. Note that `left` and `right` are mutually exclusive – fingering may be placed only on one side or the other, not both.

> **Note:** To control the placement of the fingering of a single note using this command it is necessary to write it as a single note chord by placing angle brackets round it.

Here are a few examples:

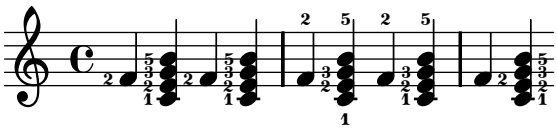
```
\set fingeringOrientations = #'(left)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(left)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(up left down)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(up left)
```

```
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(right)
<f-2>
< c-1  e-2 g-3 b-5 > 4
```



If the fingering seems a little crowded the `font-size` could be reduced. The default value can be seen from the `Fingering` object in the IR to be `-5`, so let's try `-7`:

```
\override Fingering #'font-size = #-7
\set fingeringOrientations = #'(left)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(left)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(up left down)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(up left)
<f-2>
< c-1  e-2 g-3 b-5 > 4
\set fingeringOrientations = #'(right)
<f-2>
< c-1  e-2 g-3 b-5 > 4
```



4.4.3 Outside-staff objects

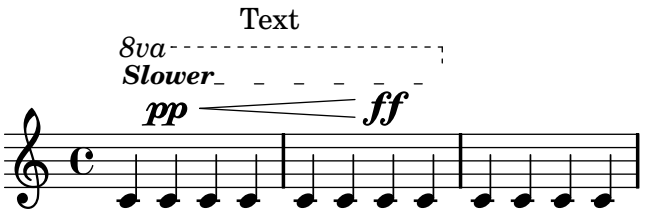
Outside-staff objects are automatically placed to avoid collisions. Objects with the lower value of the `outside-staff-priority` property are placed nearer to the staff, and other outside-staff objects are then raised as far as necessary to avoid collisions. The `outside-staff-priority` is defined in the `grob-interface` and so is a property of all layout objects. By default it is set to `#f` for all within-staff objects, and to a numerical value appropriate to each outside-staff object when the object is created. The following table shows the default numerical values for some of the commonest outside-staff objects which are, by default, placed in the `Staff` or `Voice` contexts.

Layout Object	**Priority**	**Controls position of:**
`MultiMeasureRestText`	450	Text over full-bar rests
`TextScript`	450	Markup text
`OttavaBracket`	400	Ottava brackets
`TextSpanner`	350	Text spanners
`DynamicLineSpanner`	250	All dynamic markings
`VoltaBracketSpanner`	100	Volta brackets

TrillSpanner	50	Spanning trills

Here is an example showing the default placement of some of these.

```
% Set details for later Text Spanner
\override TextSpanner #'(bound-details left text)
    = \markup { \small \bold Slower }
% Place dynamics above staff
\dynamicUp
% Start Ottava Bracket
\ottava #1
c' \startTextSpan
% Add Dynamic Text
c\pp
% Add Dynamic Line Spanner
c\<
% Add Text Script
c^Text
c c
% Add Dynamic Text
c\ff c \stopTextSpan
% Stop Ottava Bracket
\ottava #0
c, c c c
```



This example also shows how to create Text Spanners – text with extender lines above a section of music. The spanner extends from the `\startTextSpan` command to the `\stopTextSpan` command, and the format of the text is defined by the `\override TextSpanner` command. For more details see Section "Text spanners" in *Notation Reference*.

It also shows how ottava brackets are created.

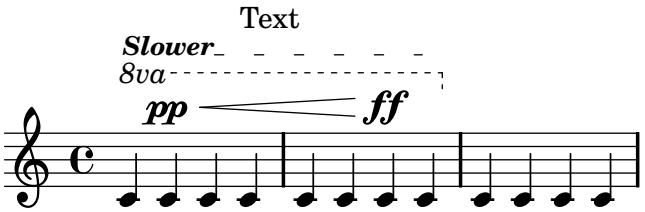
Note that bar numbers, metronome marks and rehearsal marks are not shown. By default these are created in the `Score` context and their `outside-staff-priority` is ignored relative to the layout objects which are created in the `Staff` context. If you wish to place bar numbers, metronome marks or rehearsal marks in accordance with the value of their `outside-staff-priority` the `Bar_number_engraver`, `Metronome_mark_engraver` or `Mark_engraver` respectively should be removed from the `Score` context and placed in the top `Staff` context. If this is done, these marks will be given the following default `outside-staff-priority` values:

Layout Object	**Priority**
`RehearsalMark`	`1500`
`MetronomeMark`	`1000`
`BarNumber`	`100`

If the default values of `outside-staff-priority` do not give you the placing you want, the priority of any of the objects may be overridden. Suppose we would like the ottava bracket to be placed below the text spanner in the example above. All we need to do is to look up the

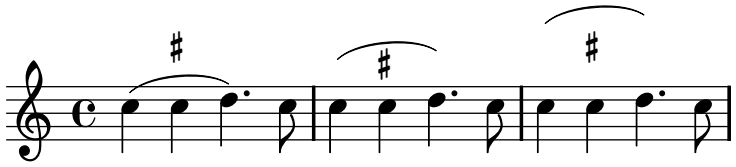
priority of `OttavaBracket` in the IR or in the tables above, and reduce it to a value lower than that of a `TextSpanner`, remembering that `OttavaBracket` is created in the `Staff` context:

```
% Set details for later Text Spanner
\override TextSpanner #'(bound-details left text)
    = \markup { \small \bold Slower }
% Place dynamics above staff
\dynamicUp
%Place following Ottava Bracket below Text Spanners
\once \override Staff.OttavaBracket #'outside-staff-priority = #340
% Start Ottava Bracket
\ottava #1
c' \startTextSpan
% Add Dynamic Text
c\pp
% Add Dynamic Line Spanner
c\<
% Add Text Script
c^Text
c c
% Add Dynamic Text
c\ff c \stopTextSpan
% Stop Ottava Bracket
\ottava #0
c, c c c
```



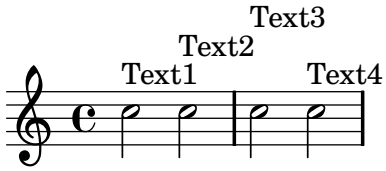
Slurs by default are classed as within-staff objects, but they often appear above the staff if the notes to which they are attached are high on the staff. This can push outside-staff objects such as articulations too high, as the slur will be placed first. The `avoid-slur` property of the articulation can be set to `'inside` to bring the articulation inside the slur, but the `avoid-slur` property is effective only if the `outside-staff-priority` is also set to `#f`. Alternatively, the `outside-staff-priority` of the slur can be set to a numerical value to cause it to be placed along with other outside-staff objects according to that value. Here's an example showing the effect of the two methods:

```
c4( c^\markup\tiny\sharp d4.) c8
c4(
\once \override TextScript #'avoid-slur = #'inside
\once \override TextScript #'outside-staff-priority = ##f
c^\markup\tiny\sharp d4.) c8
\once \override Slur #'outside-staff-priority = #500
c4( c^\markup\tiny\sharp d4.) c8
```



Changing the `outside-staff-priority` can also be used to control the vertical placement of individual objects, although the results may not always be desirable. Suppose we would like "Text3" to be placed above "Text4" in the example under Automatic behavior, above (see Section 4.4.1 [Automatic behavior], page 99). All we need to do is to look up the priority of `TextScript` in the IR or in the tables above, and increase the priority of "Text3" to a higher value:

```
c2^"Text1"
c^"Text2"
\once \override TextScript #'outside-staff-priority = #500
c^"Text3"
c^"Text4"
```

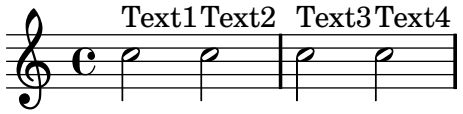


This certainly lifts "Text3" above "Text4" but it also lifts it above "Text2", and "Text4" now drops down. Perhaps this is not so good. What we would really like to do is to position all the annotation at the same distance above the staff. To do this, we clearly will need to space the notes out horizontally to make more room for the text. This is done using the `textLengthOn` command.

\textLengthOn

By default, text produced by markup takes up no horizontal space as far as laying out the music is concerned. The `\textLengthOn` command reverses this behavior, causing the notes to be spaced out as far as is necessary to accommodate the text:

```
\textLengthOn  % Cause notes to space out to accommodate text
c2^"Text1"
c^"Text2"
c^"Text3"
c^"Text4"
```

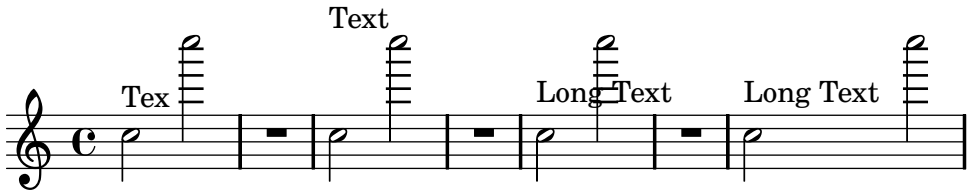


The command to revert to the default behavior is `\textLengthOff`. Remember `\once` only works with `\override`, `\set`, `\revert` or `unset`, so cannot be used with `\textLengthOn`.

Markup text will also avoid notes which project above the staff. If this is not desired, the automatic displacement upwards may be turned off by setting the priority to `#f`. Here's an example to show how markup text interacts with such notes.

```
% This markup is short enough to fit without collision
c2^"Tex"
c''2
R1
% This is too long to fit, so it is displaced upwards
```

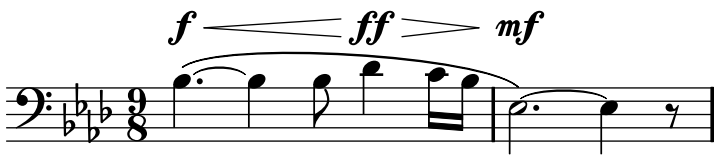
```
c,,2^"Text"
c''2
R1
% Turn off collision avoidance
\once \override TextScript #'outside-staff-priority = ##f
c,,2^"Long Text   "
c''2
R1
% Turn off collision avoidance
\once \override TextScript #'outside-staff-priority = ##f
\textLengthOn  % and turn on textLengthOn
c,,2^"Long Text   "  % Spaces at end are honored
c''2
```



Dynamics

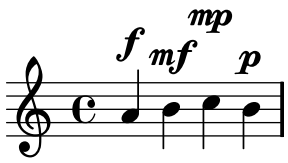
Dynamic markings will normally be positioned beneath the staff, but may be positioned above with the `dynamicUp` command. They will be positioned vertically relative to the note to which they are attached, and will float below (or above) all within-staff objects such as phrasing slurs and bar numbers. This can give quite acceptable results, as this example shows:

```
\clef "bass"
\key aes \major
\time 9/8
\dynamicUp
bes4.~\f\< \( bes4 bes8 des4\ff\> c16 bes\! |
ees,2.~\)\mf ees4 r8 |
```



However, if the notes and attached dynamics are close together the automatic placement will avoid collisions by displacing later dynamic markings further away, but this may not be the optimum placement, as this rather artificial example shows:

```
\dynamicUp
a4\f b\mf c\mp b\p
```



Should a similar situation arise in ‘real’ music, it may be preferable to space out the notes a little further, so the dynamic markings can all fit at the same vertical distance from the staff.

We were able to do this for markup text by using the `\textLengthOn` command, but there is no equivalent command for dynamic marks. So we shall have to work out how to do this using `\override` commands.

Grob sizing

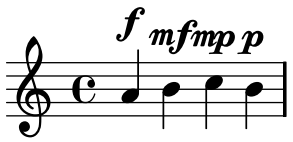
First we must learn how grobs are sized. All grobs have a reference point defined within them which is used to position them relative to their parent object. This point in the grob is then positioned at a horizontal distance, `X-offset`, and at a vertical distance, `Y-offset`, from its parent. The horizontal extent of the object is given by a pair of numbers, `X-extent`, which say where the left and right edges are relative to the reference point. The vertical extent is similarly defined by a pair of numbers, `Y-extent`. These are properties of all grobs which support the `grob-interface`.

By default, outside-staff objects are given a width of zero so that they may overlap in the horizontal direction. This is done by the trick of adding infinity to the leftmost extent and minus infinity to the rightmost extent by setting the `extra-spacing-width` to `'(+inf.0 . -inf.0)`. So to ensure they do not overlap in the horizontal direction we must override this value of `extra-spacing-width` to `'(0 . 0)` so the true width shines through. This is the command to do this for dynamic text:

```
\override DynamicText #'extra-spacing-width = #'(0 . 0)
```

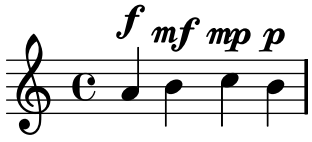
Let's see if this works in our previous example:

```
\dynamicUp
\override DynamicText #'extra-spacing-width = #'(0 . 0)
a4\f b\mf c\mp b\p
```



Well, it has certainly stopped the dynamic marks being displaced, but two problems remain. The marks should be spaced a little further apart and it would be better if they were all the same distance from the staff. We can solve the first problem easily. Instead of making the `extra-spacing-width` zero we could add a little more to it. The units are the space between two staff lines, so moving the left edge half a unit to the left and the right edge half a unit to the right should do it:

```
\dynamicUp
% Extend width by 1 staff space
\override DynamicText #'extra-spacing-width = #'(-0.5 . 0.5)
a4\f b\mf c\mp b\p
```



This looks better, but maybe we would prefer the dynamic marks to be aligned along the same baseline rather than going up and down with the notes. The property to do this is `staff-padding` which is covered in the following section.

4.5 Collisions of objects

4.5.1 Moving objects

This may come as a surprise, but LilyPond is not perfect. Some notation elements can overlap. This is unfortunate, but in fact rather rare. Usually the need to move objects is for clarity or aesthetic reasons – they would look better with a little more or a little less space around them.

There are three main approaches to resolving overlapping notation. They should be considered in the following order:

1. The **direction** of one of the overlapping objects may be changed using the predefined commands listed above for within-staff objects (see Section 4.4.2 [Within-staff objects], page 100). Stems, slurs, beams, ties, dynamics, text and tuplets may be repositioned easily in this way. The limitation is that you have a choice of only two positions, and neither may be suitable.
2. The **object properties**, which LilyPond uses when positioning layout objects, may be modified using `\override`. The advantages of making changes to this type of property are (a) that some other objects will be moved automatically if necessary to make room and (b) the single override can apply to all instances of the same type of object. Such properties include:

   - `direction`

     This has already been covered in some detail – see Section 4.4.2 [Within-staff objects], page 100.

   - `padding`, `left-padding`, `right-padding`, `staff-padding`

     As an object is being positioned the value of its `padding` property specifies the gap that must be left between itself and the nearest edge of the object against which it is being positioned. Note that it is the `padding` value of the object **being placed** that is used; the `padding` value of the object which is already placed is ignored. Gaps specified by `padding` can be applied to all objects which support the `side-position-interface`.

     Instead of `padding`, the placement of groups of accidentals is controlled by `left-padding` and `right-padding`. These properties are to be found in the `AccidentalPlacement` object which, note, lives in the **staff** context. In the type-setting process the note heads are type-set first and then the accidentals, if any, are added to the left of the note heads using the `right-padding` property to determine the separation from the note heads. So only the `right-padding` property of the `AccidentalPlacement` object has any effect on the placement of the accidentals.

     The `staff-padding` property is closely related to the `padding` property: `padding` controls the minimum amount of space between any object which supports the `side-position-interface` and the nearest other object (generally the note or the staff lines); `staff-padding` applies only to those objects which are always set outside the staff – it controls the minimum amount of space that should be inserted between that object and the staff. Note that `staff-padding` has no effect on objects which are positioned relative to the note rather than the staff, even though it may be overridden without error for such objects – it is simply ignored.

     To discover which padding property is required for the object you wish to reposition, you need to return to the IR and look up the object's properties. Be aware that the padding properties might not be located in the obvious object, so look in objects that appear to be related.

     All padding values are measured in staff spaces. For most objects, this value is set by default to be around 1.0 or less (it varies with each object). It may be overridden if a larger (or smaller) gap is required.

- `self-alignment-X`

  This property can be used to align the object to the left, to the right, or to center it with respect to the parent object's reference point. It may be used with all objects which support the `self-alignment-interface`. In general these are objects that contain text. The values are `LEFT`, `RIGHT` or `CENTER`. Alternatively, a numerical value between `-1` and `+1` may be specified, where `-1` is left-aligned, `+1` is right-aligned, and numbers in between move the text progressively from left-aligned to right-aligned. Numerical values greater than `1` may be specified to move the text even further to the left, or less than `-1` to move the text even further to the right. A change of `1` in the value corresponds to a movement of half the text's length.

- `extra-spacing-width`

  This property is available for all objects which support the `item-interface`. It takes two numbers, the first is added to the leftmost extent and the second is added to the rightmost extent. Negative numbers move the edge to the left, positive to the right, so to widen an object the first number must be negative, the second positive. Note that not all objects honor both numbers. For example, the `Accidental` object only takes notice of the first (left edge) number.

- `staff-position`

  `staff-position` is a property of the `staff-symbol-referencer-interface`, which is supported by objects which are positioned relative to the staff. It specifies the vertical position of the object relative to the center line of the staff in half staff-spaces. It is useful in resolving collisions between layout objects like multi-measure rests, ties and notes in different voices.

- `force-hshift`

  Closely spaced notes in a chord, or notes occurring at the same time in different voices, are arranged in two, occasionally more, columns to prevent the note heads overlapping. These are called note columns, and an object called `NoteColumn` is created to lay out the notes in that column.

  The `force-hshift` property is a property of a `NoteColumn` (actually of the `note-column-interface`). Changing it permits a note column to be moved in units appropriate to a note column, viz. the note head width of the first voice note. It should be used in complex situations where the normal `\shiftOn` commands (see Section 3.2.2 [Explicitly instantiating voices], page 48) do not resolve the note conflict. It is preferable to the `extra-offset` property for this purpose as there is no need to work out the distance in staff-spaces, and moving the notes into or out of a `NoteColumn` affects other actions such as merging note heads.

3. Finally, when all else fails, objects may be manually repositioned relative to the staff center line vertically, or by displacing them by any distance to a new position. The disadvantages are that the correct values for the repositioning have to be worked out, often by trial and error, for every object individually, and, because the movement is done after LilyPond has placed all other objects, the user is responsible for avoiding any collisions that might ensue. But the main difficulty with this approach is that the repositioning values may need to be reworked if the music is later modified. The properties that can be used for this type of manual repositioning are:

   `extra-offset`

   This property applies to any layout object supporting the `grob-interface`. It takes a pair of numbers which specify the extra displacement in the horizontal and vertical directions. Negative numbers move the object to the left or down. The units are staff-spaces. The extra displacement is made after the typeset-

ting of objects is finished, so an object may be repositioned anywhere without affecting anything else.

`positions`
: This is most useful for manually adjusting the slope and height of beams, slurs, and tuplets. It takes a pair of numbers giving the position of the left and right ends of the beam, slur, etc. relative to the center line of the staff. Units are staff-spaces. Note, though, that slurs and phrasing slurs cannot be repositioned by arbitrarily large amounts. LilyPond first generates a list of possible positions for the slur and by default finds the slur that "looks best". If the `positions` property has been overridden the slur that is closest to the requested positions is selected from the list.

A particular object may not have all of these properties. It is necessary to go to the IR to look up which properties are available for the object in question.

Here is a list of the objects which are most likely to be involved in collisions, together with the name of the object which should be looked up in the IR in order to discover which properties should be used to move them.

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

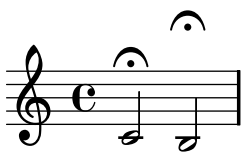
4.5.2 Fixing overlapping notation

Let's now see how the properties in the previous section can help to resolve overlapping notation.

padding property

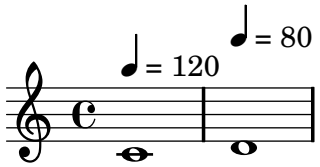
The `padding` property can be set to increase (or decrease) the distance between symbols that are printed above or below notes.

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



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

```
\tempo 4=80
d1
```



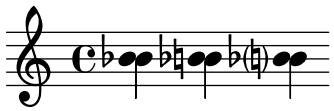
Note in the second example how important it is to figure out what context handles a certain object. Since the `MetronomeMark` object is handled in the `Score` context, property changes in the `Voice` context will not be noticed. For more details, see Section "Modifying properties" in *Notation Reference*.

If the `padding` property of an object is increased when that object is in a stack of objects being positioned according to their `outside-staff-priority`, then that object and all objects outside it are moved.

left-padding and right-padding

The `right-padding` property affects the spacing between the accidental and the note to which it applies. It is not often required, but the following example shows one situation where it is needed. Suppose we wish to show a chord containing both a B-natural and a B-flat. To avoid ambiguity we would like to precede the notes with both a natural and a flat sign. Here are a few attempts to do this:

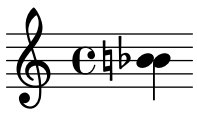
```
<b bes>
<b! bes>
<b? bes>
```



None work, with the second two showing bad collisions between the two signs.

One way of achieving this is to override the accidental stencil with a markup containing the natural and flat symbols in the order we would like, like this:

```
naturalplusflat = \markup { \natural \flat }
\relative c'' {
  \once \override Accidental
    #'stencil = #ly:text-interface::print
  \once \override Accidental #'text = #naturalplusflat
  \once \override Score.AccidentalPlacement #'right-padding = #1.5
  <b bes>
}
```

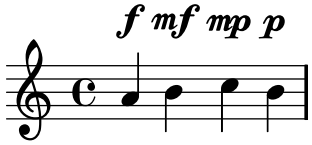


This necessarily uses an override for the accidental stencil which will not be covered until later. The stencil type must be a procedure, here changed to print the contents of the `text` property of `Accidental`, which itself is set to be a natural sign followed by a flat sign. These are then moved further away from the note head by overriding `right-padding`.

staff-padding property

`staff-padding` can be used to align objects such as dynamics along a baseline at a fixed height above the staff, rather than at a height dependent on the position of the note to which they are attached. It is not a property of `DynamicText` but of `DynamicLineSpanner`. This is because the baseline should apply equally to **all** dynamics, including those created as extended spanners. So this is the way to align the dynamic marks in the example taken from the previous section:

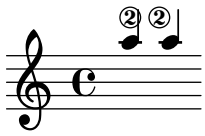
```
\dynamicUp
% Extend width by 1 unit
\override DynamicText #'extra-spacing-width = #'(-0.5 . 0.5)
% Align dynamics to a base line 2 units above staff
\override DynamicLineSpanner #'staff-padding = #2
a4\f b\mf c\mp b\p
```



self-alignment-X property

The following example shows how this can resolve the collision of a string fingering object with a note's stem by aligning the right edge with the reference point of the parent note:

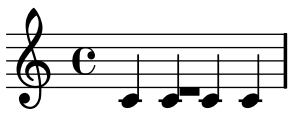
```
\voiceOne
< a \2 >
\once \override StringNumber #'self-alignment-X = #RIGHT
< a \2 >
```



staff-position property

Multimeasure rests in one voice can collide with notes in another. Since these rests are typeset centered between the bar lines, it would require significant effort for LilyPond to figure out which other notes might collide with it, since all the current collision handling between notes and between notes and rests is done only for notes and rests that occur at the same time. Here's an example of a collision of this type:

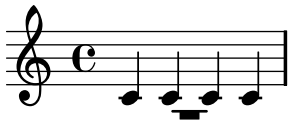
```
<< {c c c c} \\ {R1} >>
```



The best solution here is to move the multimeasure rest down, since the rest is in voice two. The default in `\voiceTwo` (i.e. in the second voice of a `<<{...} \\ {...}>>` construct) is that `staff-position` is set to -4 for MultiMeasureRest, so we need to move it, say, four half-staff spaces down to `-8`.

```
<<
  {c c c c}
\\
```

```
  \override MultiMeasureRest #'staff-position = #-8
  {R1}
>>
```



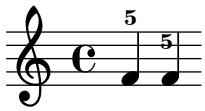
This is better than using, for example, `extra-offset`, because the ledger line above the rest is inserted automatically.

extra-offset property

The `extra-offset` property provides complete control over the positioning of an object both horizontally and vertically.

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

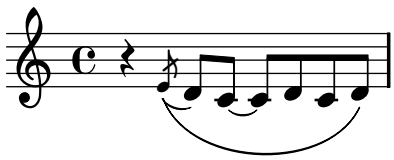
```
\stemUp
f-5
\once \override Fingering
    #'extra-offset = #'(-0.3 . -1.8)
f-5
```



positions property

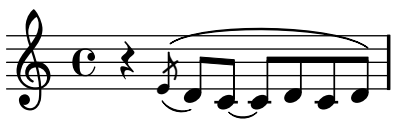
The `positions` property allows the position and slope of tuplets, slurs, phrasing slurs and beams to be controlled manually. Here's an example which has an ugly phrasing slur due to its trying to avoid the slur on the acciaccatura.

```
r4  \acciaccatura e8\( d8 c ~c d c d\)
```



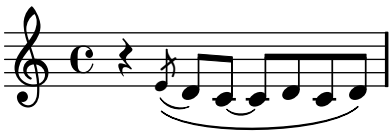
We could simply move the phrasing slur above the notes, and this would be the preferred solution:

```
r4
\phrasingSlurUp
\acciaccatura e8\( d8 c ~c d c d\)
```



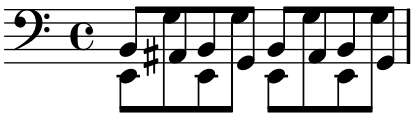
But if there were some reason why this could not be done the other alternative would be to move the left end of the phrasing slur down a little using the `positions` property. This also resolves the rather nasty shape.

```
r4
\once \override PhrasingSlur #'positions = #'(-4 . -3)
\acciaccatura
e8\( d8 c ~c d c d\)
```



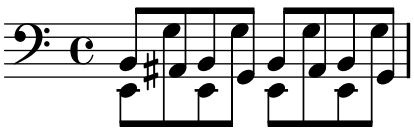
Here's a further example taken from the opening of the left-hand staff of Chopin's Prelude Op 28 No. 2. We see that the beam collides with the upper notes:

```
{
\clef "bass"
<< {b,8 ais, b, g,} \\ {e, g e, g} >>
<< {b,8 ais, b, g,} \\ {e, g e, g} >>
}
```



This can be resolved by manually moving both ends of the beam up from their position at 2 staff-spaces above the center line to, say, 3:

```
{
  \clef "bass"
  <<
    \override Beam #'positions = #'(3 . 3)
    {b,8 ais, b, g,}
  \\
    {e, g e, g}
  >>
  << {b,8 ais, b, g,} \\ {e, g e, g} >>
}
```



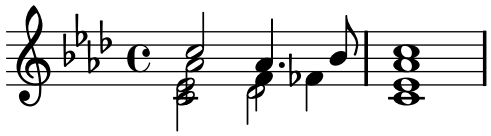
Note that the override continues to apply in the first voice of the second block of quavers, but not to any of the beams in the second voice.

force-hshift property

We can now see how to apply the final corrections to the Chopin example introduced at the end of Section 3.2.1 [I'm hearing Voices], page 44, which was left looking like this:

```
\new Staff \relative c'' {
  \key aes \major
  <<
    { c2 aes4. bes8 } \\
    { aes2 f4 fes   } \\
```

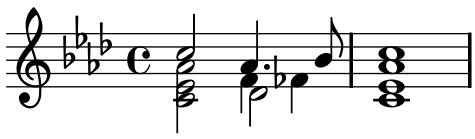
```
    { \voiceFour
      <ees c>2
      des2
    }
  >> |
  <c ees aes c>1 |
}
```



The lower two notes of the first chord (i.e, those in the third voice) should not be shifted away from the note column of the higher two notes. To correct this we set `force-hshift`, which is a property of `NoteColumn`, of these notes to zero. The lower note of the second chord is best placed just to the right of the higher notes. We achieve this by setting `force-hshift` of this note to 0.5, ie half a note head's width to the right of the note column of the higher notes.

Here's the final result:

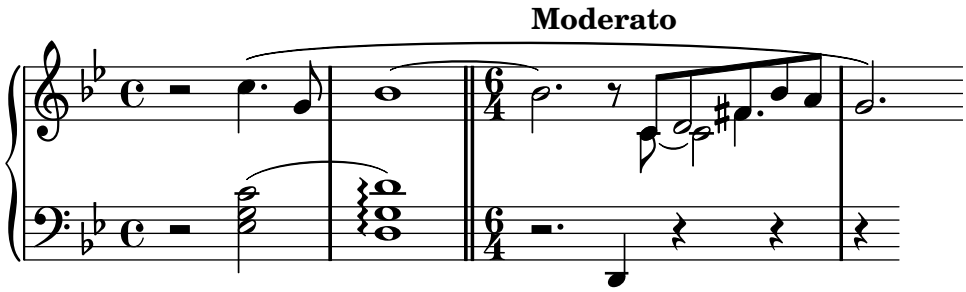
```
\new Staff \relative c'' {
  \key aes \major
  <<
    { c2 aes4. bes8 } \\
    { aes2 f4 fes   } \\
    { \voiceFour
      \once \override NoteColumn #'force-hshift = #0 <ees c>2
      \once \override NoteColumn #'force-hshift = #0.5 des2
    }
  >> |
  <c ees aes c>1 |
}
```



4.5.3 Real music example

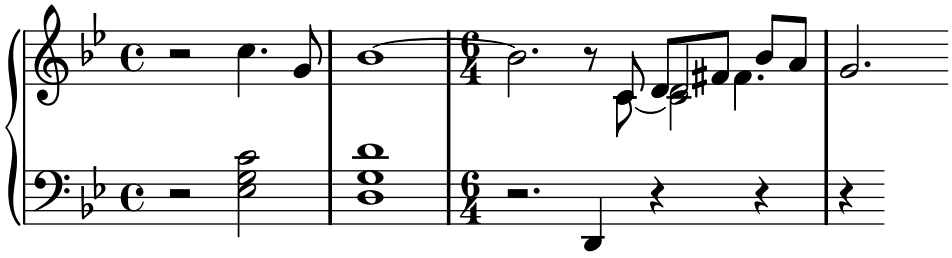
We end this section on Tweaks by showing the steps to be taken to deal with a tricky example which needs several tweaks to produce the desired output. The example has been deliberately chosen to illustrate the use of the Notation Reference to resolve unusual problems with notation. It is not representative of more usual engraving process, so please do not let these difficulties put you off! Fortunately, difficulties like these are not very common!

The example is from Chopin's Première Ballade, Op. 23, bars 6 to 9, the transition from the opening Lento to Moderato. Here, first, is what we want the output to look like, but to avoid over-complicating the example too much we have left out the dynamics, fingering and pedalling.



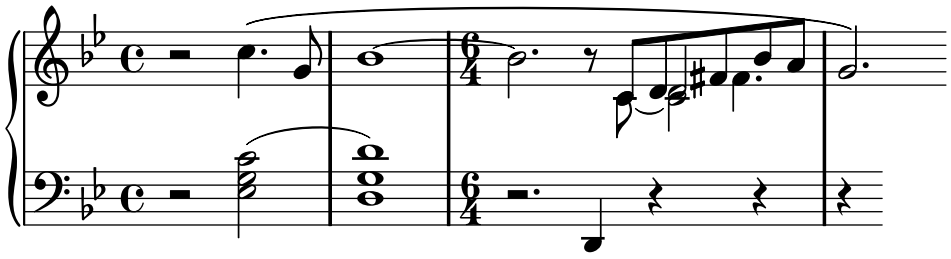
We note first that the right hand part in the third bar requires four voices. These are the five beamed eighth notes, the tied C, the half-note D which is merged with the eighth note D, and the dotted quarter note F-sharp, which is also merged with the eighth note at the same pitch. Everything else is in a single voice, so the easiest way is to introduce these four voices temporarily at the time they are needed. If you have forgotten how to do this, look at Section 3.2.1 [I'm hearing Voices], page 44. Let us begin by entering the notes as two variables and setting up the staff structure in a score block, and see what LilyPond produces by default:

```
rhMusic = \relative c'' {
  r2 c4. g8 |
  bes1~ |
  \time 6/4
  bes2. r8
  % Start polyphonic section of four voices
  <<
    {c,8 d fis bes a | }
  \\
    {c,8~ c2 | }
  \\
    {s8 d2 | }
  \\
    {s4 fis4. | }
  >>
  g2.
}

lhMusic = \relative c' {
  r2 <c g ees>2 |
  <d g, d>1 |
  r2. d,,4 r4 r |
  r4
}

\score {
  \new PianoStaff <<
    \new Staff = "RH"  <<
      \key g \minor
      \rhMusic
    >>
    \new Staff = "LH" <<
      \key g \minor
      \clef "bass"
      \lhMusic
    >>
  >>
}
```



All the notes are right, but the appearance is far from satisfactory. The tie clashes with the change in time signature, the beaming in the third bar is wrong, the notes are not merged together, and several notation elements are missing. Let's first deal with the easier things. We can correct the beaming by inserting a beam manually, and we can easily add the left hand slur and the right hand phrasing slur, since these were all covered in the Tutorial. Doing this gives:

```
rhMusic = \relative c'' {
  r2 c4.\( g8 |
  bes1~ |
  \time 6/4
  bes2. r8
  % Start polyphonic section of four voices
  <<
    {c,8[ d fis bes a] | }
  \\
    {c,8~ c2 | }
  \\
    {s8 d2 | }
  \\
    {s4 fis4. | }
  >>
  g2.\)
}

lhMusic = \relative c' {
  r2 <c g ees>2( |
  <d g, d>1) |
  r2. d,,4 r4 r |
  r4
}

\score {
  \new PianoStaff <<
    \new Staff = "RH"  <<
      \key g \minor
      \rhMusic
    >>
    \new Staff = "LH" <<
      \key g \minor
      \clef "bass"
      \lhMusic
    >>
  >>
}
```



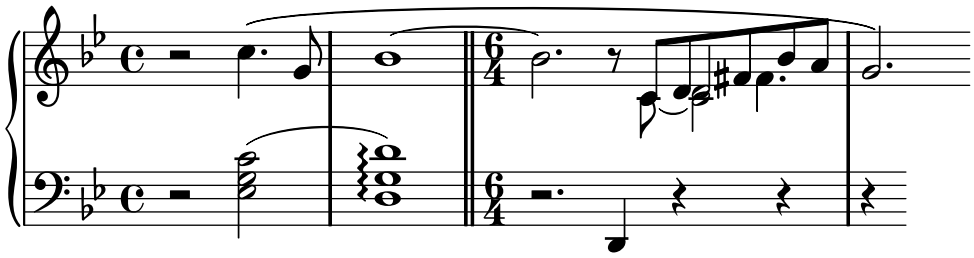
The first bar is now correct. The second bar contains an arpeggio and is terminated by a double bar line. How do we do these, as they have not been mentioned in this Learning Manual? This is where we need to turn to the Notation Reference. Looking up 'arpeggio' and 'bar line' in the index quickly shows us that an arpeggio is produced by appending `\arpeggio` to a chord, and a double bar line is produced by the `\bar "||"` command. That's easily done. We next need to correct the collision of the tie with the time signature. This is best done by moving the tie upwards. Moving objects was covered earlier in Section 4.5.1 [Moving objects], page 109, which says that objects positioned relative to the staff can be moved by overriding their `staff-position` property, which is specified in half staff spaces relative to the center line of the staff. So the following override placed just before the first tied note would move the tie up to 3.5 half staff spaces above the center line:

```
\once \override Tie #'staff-position = #3.5
```

This completes bar two, giving:

```
rhMusic = \relative c'' {
  r2 c4.\( g8 |
  \once \override Tie #'staff-position = #3.5
  bes1~ |
  \bar "||"
  \time 6/4
  bes2. r8
  % Start polyphonic section of four voices
  <<
    {c,8[ d fis bes a] | }
  \\
    {c,8~ c2 | }
  \\
    {s8 d2 | }
  \\
    {s4 fis4. | }
  >>
  g2.\)
}

lhMusic = \relative c' {
  r2 <c g ees>2( |
  <d g, d>1)\arpeggio |
  r2. d,,4 r4 r |
  r4
}

\score {
  \new PianoStaff <<
    \new Staff = "RH"  <<
      \key g \minor
      \rhMusic
    >>
```

```
    \new Staff = "LH" <<
      \key g \minor
      \clef "bass"
      \lhMusic
    >>
  >>
}
```



On to bar three and the start of the Moderato section. The tutorial showed how to add embolded text with the `\markup` command, so adding 'Moderato' in bold is easy. But how do we merge notes in different voices together? This is where we need to turn to the Notation Reference for help. A search for "merge" in the Notation Reference index quickly leads us to the commands for merging differently headed and differently dotted notes in Section "Collision resolution" in *Notation Reference*. In our example we need to merge both types of note for the duration of the polyphonic section in bar 3, so using the information we find in the Notation Reference we add

```
\mergeDifferentlyHeadedOn
\mergeDifferentlyDottedOn
```

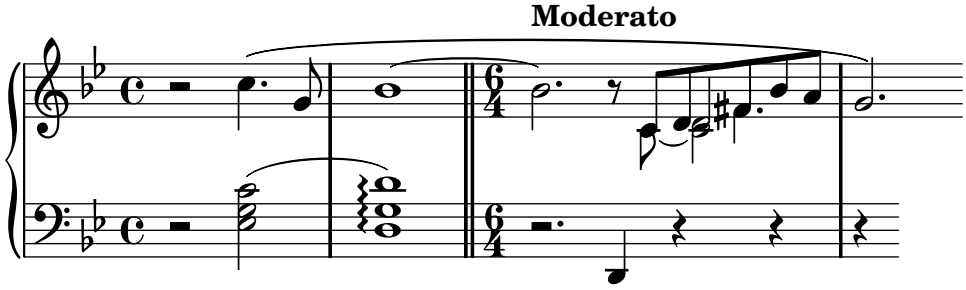
to the start of that section and

```
\mergeDifferentlyHeadedOff
\mergeDifferentlyDottedOff
```

to the end, giving:

```
rhMusic = \relative c'' {
  r2 c4.\( g8 |
  \once \override Tie #'staff-position = #3.5
  bes1~ |
  \bar "||"
  \time 6/4
  bes2.^\markup {\bold "Moderato"} r8
  \mergeDifferentlyHeadedOn
  \mergeDifferentlyDottedOn
  % Start polyphonic section of four voices
  <<
    {c,8[ d fis bes a] | }
  \\
    {c,8~ c2 | }
  \\
    {s8 d2 | }
  \\
    {s4 fis4. | }
  >>
  \mergeDifferentlyHeadedOff
  \mergeDifferentlyDottedOff
```

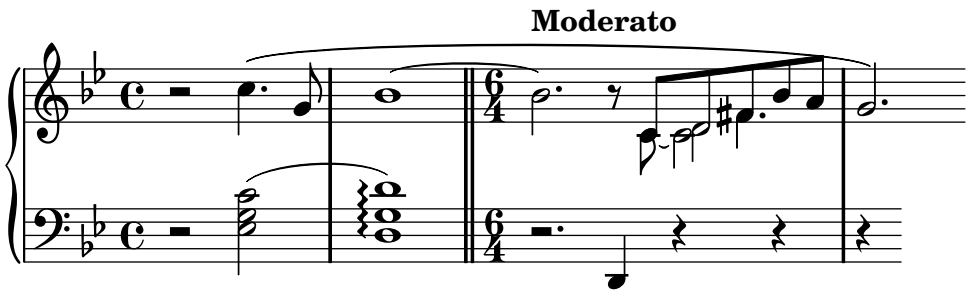
```
  g2.\)
}

lhMusic = \relative c' {
  r2 <c g ees>2( |
  <d g, d>1)\arpeggio |
  r2. d,,4 r4 r |
  r4
}

\score {
  \new PianoStaff <<
    \new Staff = "RH"  <<
      \key g \minor
      \rhMusic
    >>
    \new Staff = "LH" <<
      \key g \minor
      \clef "bass"
      \lhMusic
    >>
  >>
}
```



These overrides have merged the two F-sharp notes, but not the two on D. Why not? The answer is there in the same section in the Notation Reference – notes being merged must have stems in opposite directions and two notes cannot be merged successfully if there is a third note in the same note column. Here the two D's both have upward stems and there is a third note – the C. We know how to change the stem direction using `\stemDown`, and the Notation Reference also says how to move the C – apply a shift using one of the `\shift` commands. But which one? The C is in voice two which has shift off, and the two D's are in voices one and three, which have shift off and shift on, respectively. So we have to shift the C a further level still using `\shiftOnn` to avoid it interfering with the two D's. Applying these changes gives:

```
rhMusic = \relative c'' {
  r2 c4.\( g8 |
  \once \override Tie #'staff-position = #3.5
  bes1~ |
  \bar "||"
  \time 6/4
  bes2.^\markup {\bold "Moderato"} r8
  \mergeDifferentlyHeadedOn
  \mergeDifferentlyDottedOn
  % Start polyphonic section of four voices
```

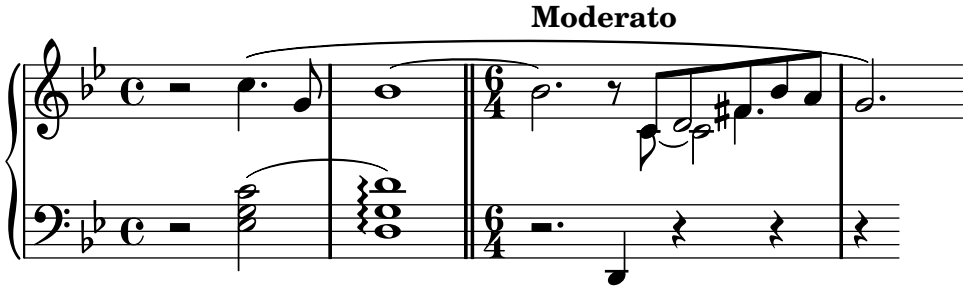
```
  <<
    {c,8[ d fis bes a] | }
  \\
    % Move the c2 out of the main note column so the merge will work
    {c,8~ \shiftOnn c2 | }
  \\
    % Stem on the d2 must be down to permit merging
    {s8 \stemDown d2 | }
  \\
    {s4 fis4. | }
  >>
  \mergeDifferentlyHeadedOff
  \mergeDifferentlyDottedOff
  g2.\)
}

lhMusic = \relative c' {
  r2 <c g ees>2( |
  <d g, d>1)\arpeggio |
  r2. d,,4 r4 r |
  r4
}

\score {
  \new PianoStaff <<
    \new Staff = "RH"  <<
      \key g \minor
      \rhMusic
    >>
    \new Staff = "LH" <<
      \key g \minor
      \clef "bass"
      \lhMusic
    >>
  >>
}
```



Nearly there. Only two problems remain: The downward stem on the merged D should not be there, and the C would be better positioned to the right of the D's. We know how to do both of these from the earlier tweaks: we make the stem transparent, and move the C with the `force-hshift` property. Here's the final result:

```
rhMusic = \relative c'' {
  r2
```

```
  c4.\( g8 |
  \once \override Tie #'staff-position = #3.5
  bes1~ |
  \bar "||"
  \time 6/4
  bes2.^\markup {\bold "Moderato"} r8
  \mergeDifferentlyHeadedOn
  \mergeDifferentlyDottedOn
  <<
    {c,8[ d fis bes a] | }
  \\
    % Reposition the c2 to the right of the merged note
    {c,8~ \once \override NoteColumn #'force-hshift = #1.0
    % Move the c2 out of the main note column so the merge will work
    \shiftOnn c2}
  \\
    % Stem on the d2 must be down to permit merging
    {s8 \stemDown \once \override Stem #'transparent = ##t d2}
  \\
    {s4 fis4.}
  >>
  \mergeDifferentlyHeadedOff
  \mergeDifferentlyDottedOff
  g2.\)
}

lhMusic = \relative c' {
  r2 <c g ees>2( |
  <d g, d>1)\arpeggio |
  r2. d,,4 r4 r |
  r4
}

\score {
  \new PianoStaff <<
    \new Staff = "RH"  <<
      \key g \minor
      \rhMusic
    >>
    \new Staff = "LH" <<
      \key g \minor
      \clef "bass"
      \lhMusic
    >>
  >>
}
```

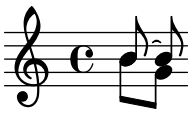


4.6 Further tweaking

4.6.1 Other uses for tweaks

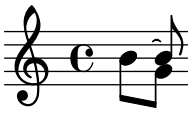
Tying notes across voices

The following example demonstrates how to connect notes in different voices using ties. Normally, only two notes in the same voice can be connected with ties. By using two voices, with the tied notes in one of them



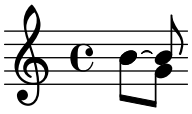
and blanking the first up-stem in that voice, the tie appears to cross voices:

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



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

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

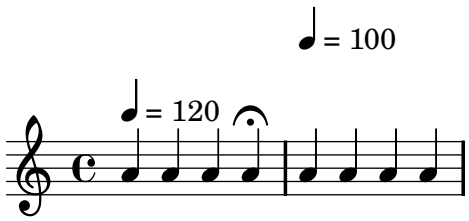


Simulating a fermata in MIDI

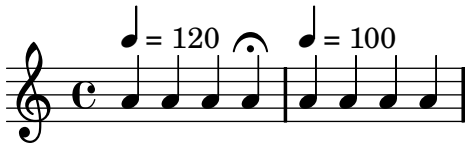
For outside-staff objects it is usually better to override the object's `stencil` property rather than its `transparent` property when you wish to remove it from the printed output. Setting the `stencil` property of an object to `#f` will remove that object entirely from the printed output. This means it has no effect on the placement of other objects placed relative to it.

For example, if we wished to change the metronome setting in order to simulate a fermata in the MIDI output we would not want the metronome markings to appear in the printed output, and we would not want it to influence the spacing between the two systems or the positions of adjacent annotations on the staff. So setting its `stencil` property to `#f` would be the best way. We show here the effect of the two methods:

```
\score {
  \relative c'' {
    % Visible tempo marking
    \tempo 4=120
    a4 a a
    \once \override Score.MetronomeMark #'transparent = ##t
    % Invisible tempo marking to lengthen fermata in MIDI
    \tempo 4=80
    a\fermata
    % New tempo for next section
    \tempo 4=100
    a a a a
  }
  \layout { }
  \midi { }
}
```




```
\score {
  \relative c'' {
    % Visible tempo marking
    \tempo 4=120
    a4 a a
    \once \override Score.MetronomeMark #'stencil = ##f
    % Invisible tempo marking to lengthen fermata in MIDI
    \tempo 4=80
    a\fermata
    % New tempo for next section
    \tempo 4=100
    a a a a
  }
  \layout { }
  \midi { }
}
```



Both methods remove the metronome mark which lengthens the fermata from the printed output, and both affect the MIDI timing as required, but the transparent metronome mark in the first line forces the following tempo indication too high while the second (with the stencil removed) does not.

4.6.2 Using variables for tweaks

Override commands are often long and tedious to type, and they have to be absolutely correct. If the same overrides are to be used many times it may be worth defining variables to hold them.

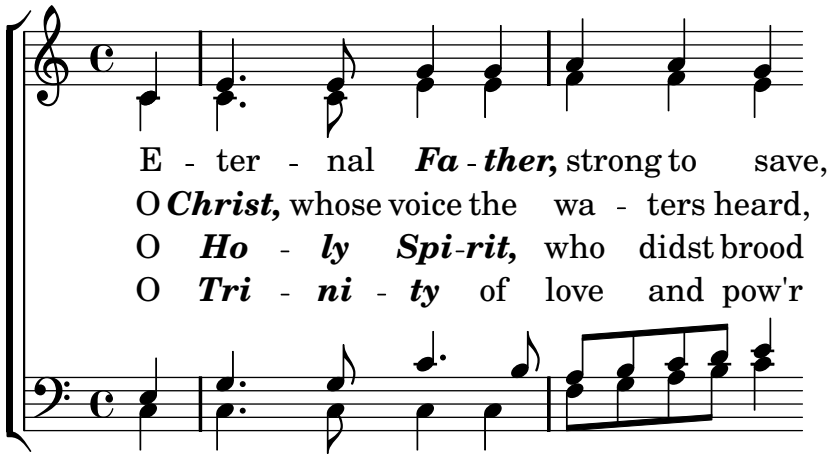
Suppose we wish to emphasize certain words in lyrics by printing them in bold italics. The `\italic` and `\bold` commands only work within lyrics if they are embedded, together with the word or words to be modified, within a `\markup` block, which makes them tedious to enter. The need to embed the words themselves prevents their use in simple variables. As an alternative can we use `\override` and `\revert` commands?

```
\override Lyrics . LyricText #'font-shape = #'italic
\override Lyrics . LyricText #'font-series = #'bold

\revert Lyrics . LyricText #'font-shape
\revert Lyrics . LyricText #'font-series
```

These would also be extremely tedious to enter if there were many words requiring emphasis. But we *can* define these as two variables and use those to bracket the words to be emphasized. Another advantage of using variables for these overrides is that the spaces around the dot are not necessary, since they are not being interpreted in `\lyricmode` directly. Here's an example of this, although in practice we would choose shorter names for the variables to make them quicker to type:

```
emphasize = {
  \override Lyrics.LyricText #'font-shape = #'italic
  \override Lyrics.LyricText #'font-series = #'bold
}
normal = {
  \revert Lyrics.LyricText #'font-shape
  \revert Lyrics.LyricText #'font-series
}

global = { \time 4/4 \partial 4 \key c \major}
SopranoMusic   = \relative c' { c4 | e4. e8 g4  g  | a a g }
AltoMusic  = \relative c' { c4 | c4. c8 e4  e  | f f e }
TenorMusic = \relative c  { e4 | g4. g8 c4. b8 | a8 b c d e4 }
BassMusic  = \relative c  { c4 | c4. c8 c4  c  | f8 g a b c4 }
VerseOne   = \lyrics { E -- | ter -- nal \emphasize Fa -- ther, \normal | strong to sav
VerseTwo   = \lyricmode { O | \emphasize Christ, \normal whose voice the | wa -- ters l
VerseThree = \lyricmode { O | \emphasize Ho -- ly Spi -- rit, \normal | who didst brooc
VerseFour  = \lyricmode { O | \emphasize Tri -- ni -- ty \normal of | love and pow'r }

\score {
  \new ChoirStaff <<
    \new Staff <<
      \clef "treble"
      \new Voice = "Soprano"  { \voiceOne \global \SopranoMusic }
```

```
        \new Voice = "Alto" { \voiceTwo \AltoMusic }
        \new Lyrics \lyricsto "Soprano" { \VerseOne   }
        \new Lyrics \lyricsto "Soprano" { \VerseTwo   }
        \new Lyrics \lyricsto "Soprano" { \VerseThree }
        \new Lyrics \lyricsto "Soprano" { \VerseFour  }
      >>
      \new Staff <<
        \clef "bass"
        \new Voice = "Tenor" { \voiceOne \TenorMusic }
        \new Voice = "Bass"  { \voiceTwo \BassMusic  }
      >>
    >>
}
```



4.6.3 Other sources of information

The Internals Reference documentation contains a lot of information about LilyPond, but even more information can be gathered by looking at the internal LilyPond files. To explore these, you must first find the directory appropriate to your system. The location of this directory depends (a) on whether you obtained LilyPond by downloading a precompiled binary from lilypond.org or whether you installed it from a package manager (i.e. distributed with Linux, or installed under fink or cygwin) or compiled it from source, and (b) on which operating system it is being used:

Downloaded from lilypond.org

- Linux

  Navigate to '*INSTALLDIR*`/lilypond/usr/share/lilypond/current/`'
- MacOS X

  Navigate to '*INSTALLDIR*`/LilyPond.app/Contents/Resources/share/lilypond/current/`' by either `cd`-ing into this directory from the Terminal, or control-clicking on the LilyPond application and selecting 'Show Package Contents'.
- Windows

  Using Windows Explorer, navigate to '*INSTALLDIR*`/LilyPond/usr/share/lilypond/current/`'

Installed from a package manager or compiled from source

Navigate to '*PREFIX*`/share/lilypond/`*X.Y.Z*`/`', where *PREFIX* is set by your package manager or `configure` script, and *X.Y.Z* is the LilyPond version number.

Within this directory the two interesting subdirectories are

- ‘ly/’ - contains files in LilyPond format
- ‘scm/’ - contains files in Scheme format

Let’s begin by looking at some files in ‘ly/’. Open ‘ly/property-init.ly’ in a text editor. The one you normally use for .ly files will be fine. This file contains the definitions of all the standard LilyPond predefined commands, such as \stemUp and \slurDotted. You will see that these are nothing more than definitions of variables containing one or a group of \override commands. For example, /tieDotted is defined to be:

```
tieDotted = {
  \override Tie #'dash-period = #0.75
  \override Tie #'dash-fraction = #0.1
}
```

If you do not like the default values these predefined commands can be redefined easily, just like any other variable, at the head of your input file.

The following are the most useful files to be found in ‘ly/’:

Filename	Contents
‘ly/engraver-init.ly’	Definitions of engraver Contexts
‘ly/paper-defaults-init.ly’	Specifications of paper-related defaults
‘ly/performer-init.ly’	Definitions of performer Contexts
‘ly/property-init.ly’	Definitions of all common predefined commands
‘ly/spanner-init.ly’	Definitions of spanner-related predefined commands

Other settings (such as the definitions of markup commands) are stored as .scm (Scheme) files. The Scheme programming language is used to provide a programmable interface into LilyPond internal operation. Further explanation of these files is currently outside the scope of this manual, as a knowledge of the Scheme language is required. Users should be warned that a substantial amount of technical knowledge or time is required to understand Scheme and these files (see Appendix B [Scheme tutorial], page 173).

If you have this knowledge, the Scheme files which may be of interest are:

Filename	Contents
‘scm/auto-beam.scm’	Sub-beaming defaults
‘scm/define-grobs.scm’	Default settings for grob properties
‘scm/define-markup-commands.scm’	Specify all markup commands
‘scm/midi.scm’	Default settings for MIDI output
‘scm/output-lib.scm’	Settings that affect appearance of frets, colors, accidentals, bar lines, etc
‘scm/parser-clef.scm’	Definitions of supported clefs
‘scm/script.scm’	Default settings for articulations

4.6.4 Avoiding tweaks with slower processing

LilyPond can perform extra checks while it processes input files. These checks will take extra time to perform, but fewer manual tweaks may be required to obtain an acceptable result. If a text script or part of the lyrics extends over the margins these checks will compress that line of the score just enough to fit within the margins.

To be effective under all circumstances these checks must be enabled by placing the overrides in a Score \with block, rather than in-line in music, as follows:

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

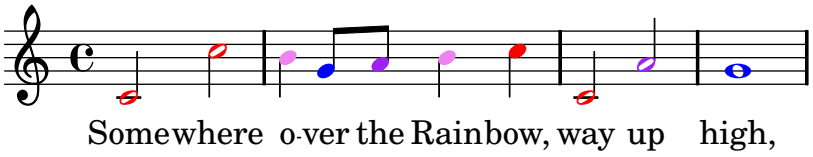
```
  \override NonMusicalPaperColumn #'keep-inside-line = ##t
} {
   ..
}
```

4.6.5 Advanced tweaks with Scheme

Although many things are possible with the `\override` and `\tweak` commands, an even more powerful way of modifying the action of LilyPond is available through a programmable interface to the LilyPond internal operation. Code written in the Scheme programming language can be incorporated directly in the internal operation of LilyPond. Of course, at least a basic knowledge of programming in Scheme is required to do this, and an introduction is provided in the Appendix B [Scheme tutorial], page 173.

As an illustration of one of the many possibilities, instead of setting a property to a constant it can be set to a Scheme procedure which is then called whenever that property is accessed by LilyPond. The property can then be set dynamically to a value determined by the procedure at the time it is called. In this example we color the note head in accordance with its position on the staff.

```
#(define (color-notehead grob)
  "Color the notehead according to its position on the staff."
  (let ((mod-position (modulo (ly:grob-property grob 'staff-position) 7)))
    (case mod-position
      ;;   Return rainbow colors
      ((1) (x11-color 'red    ))  ; for C
      ((2) (x11-color 'orange ))  ; for D
      ((3) (x11-color 'yellow ))  ; for E
      ((4) (x11-color 'green  ))  ; for F
      ((5) (x11-color 'blue   ))  ; for G
      ((6) (x11-color 'purple ))  ; for A
      ((0) (x11-color 'violet ))  ; for B
    )
  )
)

\relative c' {
  % Arrange to obtain color from color-notehead procedure
  \override NoteHead #'color = #color-notehead
  c2 c' |
  b4 g8 a b4 c |
  c,2 a' |
  g1 |
}
\addlyrics {
  Some -- where o -- ver the Rain -- bow, way up high,
}
```



Further examples showing the use of these programmable interfaces can be found in Section B.1 [Tweaking with Scheme], page 174.

5 Working on LilyPond projects

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

5.1 Suggestions for writing LilyPond input files

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

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

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

5.1.1 General suggestions

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

- **Include `\version` numbers in every file**. Note that all templates contain `\version` information. We highly recommend that you always include the `\version`, no matter how small your file is. Speaking from personal experience, it's quite frustrating to try to remember which version of LilyPond you were using a few years ago. `convert-ly` requires you to declare which version of LilyPond you used.
- **Include checks**: Section "Bar and bar number checks" in *Notation Reference*, Section "Octave checks" in *Notation Reference*. If you include checks every so often, then if you make a mistake, you can pinpoint it quicker. How often is 'every so often'? It depends on the complexity of the music. For very simple music, perhaps just once or twice. For very complex music, perhaps every bar.
- **One bar per line of text**. If there is anything complicated, either in the music itself or in the output you desire, it's often good to write only one bar per line. Saving screen space by cramming eight bars per line just isn't worth it if you have to 'debug' your input files.
- **Comment your input files**. Use either bar numbers (every so often) or references to musical themes ('second theme in violins,' 'fourth variation,' etc.). You may not need comments when you're writing the piece for the first time, but if you want to go back to change something two or three years later, or if you pass the source over to a friend, it will be much more challenging to determine your intentions or how your file is structured if you didn't comment the file.
- **Indent your braces**. A lot of problems are caused by an imbalance in the number of `{` and `}`.
- **Explicitly add durations** at the beginnings of sections and variables. If you specify `c4 d e` at the beginning of a phrase (instead of just `c d e`) you can save yourself some problems if you rearrange your music later.
- **Separate tweaks** from music definitions. See Section 5.1.4 [Saving typing with variables and functions], page 133, and Section 5.1.5 [Style sheets], page 134.

5.1.2 Typesetting existing music

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

- Enter one manuscript (the physical copy) system at a time (but still only one bar per line of text), and check each system when you finish it. You may use the `showLastLength` or `showFirstLength` properties to speed up processing – see Section "Skipping corrected music" in *Notation Reference*.
- Define `mBreak = { \break }` and insert `\mBreak` in the input file whenever the manuscript has a line break. This makes it much easier to compare the LilyPond music to the original music. When you are finished proofreading your score, you may define `mBreak = { }` to remove all those line breaks. This will allow LilyPond to place line breaks wherever it feels are best.
- When entering a part for a transposing instrument into a variable, it is recommended that the notes are wrapped in

  ```
  \transpose c natural-pitch {...}
  ```

  (where `natural-pitch` is the open pitch of the instrument) so that the music in the variable is effectively in C. You can transpose it back again when the variable is used, if required, but you might not want to (e.g., when printing a score in concert pitch, converting a trombone part from treble to bass clef, etc.) Mistakes in transpositions are less likely if all the music in variables is at a consistent pitch.

  Also, only ever transpose to/from C. That means that the only other keys you will use are the natural pitches of the instruments - bes for a B-flat trumpet, aes for an A-flat clarinet, etc.

5.1.3 Large projects

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

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

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

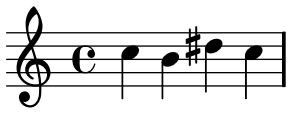
- **Separate tweaks from music definitions**. This point was made in previously, but for large projects it is absolutely vital. We might need to change the definition of `fthenp`, but then we only need to do this once, and we can still avoid touching anything inside `violin`.

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

5.1.4 Saving typing with variables and functions

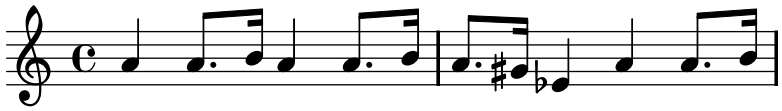
By this point, you've seen this kind of thing:

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



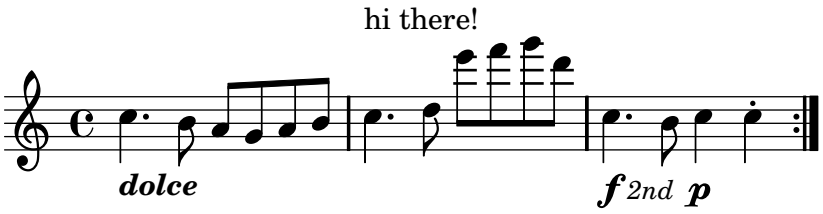
You may even realize that this could be useful in minimalist music:

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



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

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



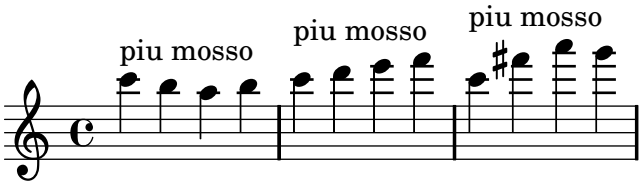
These variables are obviously useful for saving typing. But they're worth considering even if you only use them once – they reduce complexity. Let's look at the previous example without any variables. It's a lot harder to read, especially the last line.

```
violin = \relative c'' {
  \repeat volta 2 {
    c4._\markup{ \italic \bold dolce } b8 a8 g a b |
    \once \override TextScript #'padding = #5.0
    c4.^"hi there!" d8 e' f g d |
    c,4.\markup{ \dynamic f \italic \small { 2nd }
      \hspace #0.1 \dynamic p } b8 c4 c-. |
  }
}
```

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

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

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

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



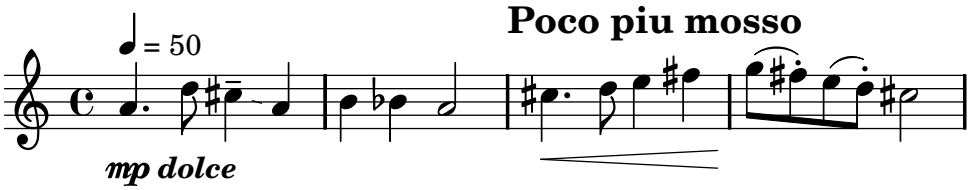
Using variables is also a good way to reduce work if the LilyPond input syntax changes (see Section 5.2.1 [Updating old input files], page 138). If you have a single definition (such as `\dolce`) for all your input files (see Section 5.1.5 [Style sheets], page 134), then if the syntax changes, you only need to update your single `\dolce` definition, instead of making changes throughout every `.ly` file.

5.1.5 Style sheets

The output that LilyPond produces can be heavily modified; see Chapter 4 [Tweaking output], page 80, for details. But what if you have many input files that you want to apply your tweaks to? Or what if you simply want to separate your tweaks from the actual music? This is quite easy to do.

Let's look at an example. Don't worry if you don't understand the parts with all the `#()`. This is explained in Section 4.6.5 [Advanced tweaks with Scheme], page 129.

```
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line(#:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'extra-spacing-width = #'(+inf.0 . -inf.0)
  \mark \markup { \bold $markp }
#})

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

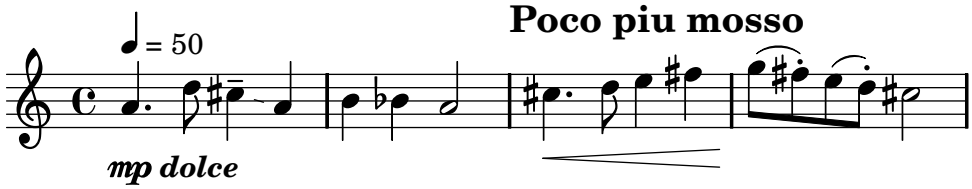


There are some problems with overlapping output; we'll fix those using the techniques in Section 4.5.1 [Moving objects], page 109. But let's also do something about the `mpdolce` and `tempoMark` definitions. They produce the output we desire, but we might want to use them in another piece. We could simply copy-and-paste them at the top of every file, but that's an annoyance. It also leaves those definitions in our input files, and I personally find all the `#()` somewhat ugly. Let's hide them in another file:

```
%%% save this to a file called "definitions.ly"
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line(#:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'extra-spacing-width = #'(+inf.0 . -inf.0)
  \mark \markup { \bold $markp }
#})
```

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

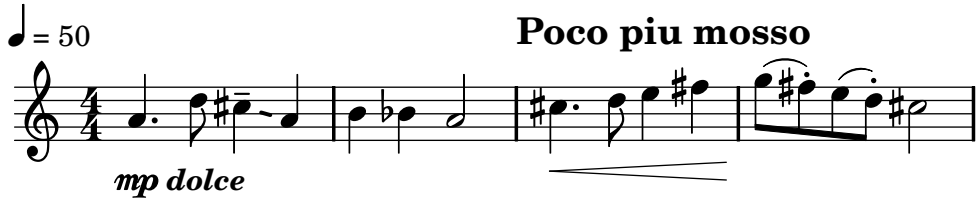
```
\include "definitions.ly"

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



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

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

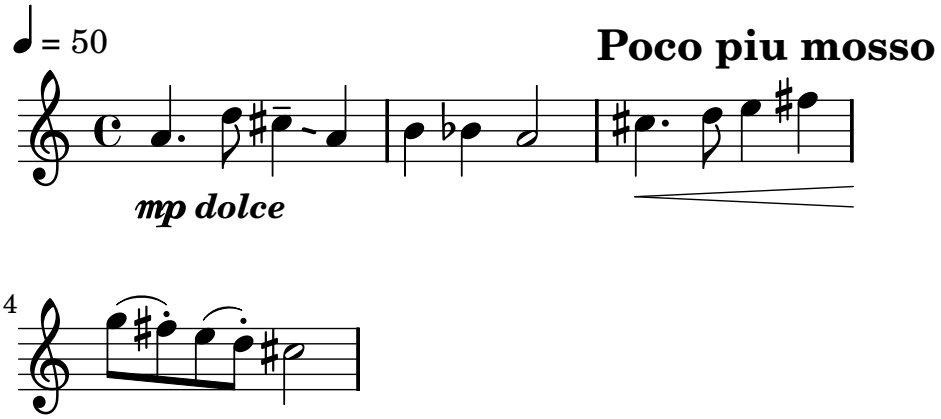
```
%%%  definitions.ly
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line( #:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'extra-spacing-width = #'(+inf.0 . -inf.0)
  \mark \markup { \bold $markp }
#})

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



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

```
%%%  definitions.ly
mpdolce = #(make-dynamic-script (markup #:hspace 1 #:translate (cons 5 0)
  #:line( #:dynamic "mp" #:text #:italic "dolce" )))
tempoMark = #(define-music-function (parser location markp) (string?)
#{
  \once \override Score . RehearsalMark #'self-alignment-X = #left
  \once \override Score . RehearsalMark #'extra-spacing-width = #'(+inf.0 . -inf.0)
```

```
  \mark \markup { \bold $markp }
#})

#(set-global-staff-size 23)
\layout{
  \context { \Score
    \override MetronomeMark #'extra-offset = #'(-9 . 0)
    \override MetronomeMark #'padding = #'3
  }
  \context { \Staff
  }
  \context { \Voice
    \override Glissando #'thickness = #3
    \override Glissando #'gap = #0.1
  }
}
```



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

```
\include "definitions.ly"

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

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

```
%%%   global.ly
\version "2.12.3"
#(ly:set-option 'point-and-click #f)
\include "../init/init-defs.ly"
\include "../init/init-layout.ly"
\include "../init/init-headers.ly"
\include "../init/init-paper.ly"
```

5.2 When things don't work

5.2.1 Updating old input files

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

LilyPond comes with a file that makes this updating easier: `convert-ly`. For details about how to run this program, see Section "Updating files with convert-ly" in *Application Usage*.

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

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

5.2.2 Troubleshooting (taking it all apart)

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

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

In an extreme case, you might end up with only

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

(in other words, a file without any music)

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

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

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

Another very useful debugging technique is constructing Section 5.2.3 [Minimal examples], page 139.

5.2.3 Minimal examples

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

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

To construct an example which is as small as possible, the rule is quite simple: remove anything which is not necessary. When trying to remove unnecessary parts of a file, it is a very good idea to comment out lines instead of deleting them. That way, if you discover that you actually *do* need some lines, you can uncomment them, instead of typing them in from scratch.

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

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

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

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

5.3 Scores and parts

TODO: this is really old stuff from the really old tutorial. Rewrite, fix, etc. Or maybe delete entirely. -gp Include section on tags -td and then move to section 5. Working ... -td

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

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

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

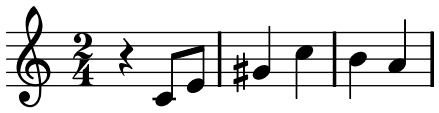
Then, an individual part is made by putting the following in a file

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

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

The line

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

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



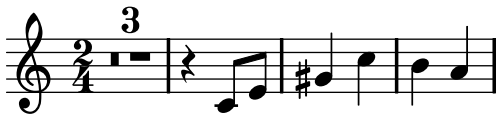
In ensemble pieces, one of the voices often does not play for many measures. This is denoted by a special rest, the multi-measure rest. It is entered with a capital `R` followed by a duration (`1` for a whole note, `2` for a half note, etc.). By multiplying the duration, longer rests can be constructed.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

```
R2*3
```

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

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

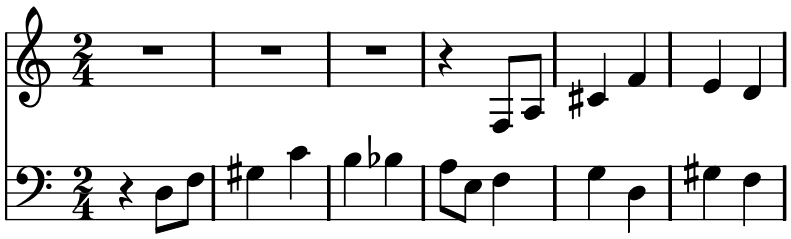
This command sets the property `skipBars` in the `Score` context to true (`##t`). Prepending the rest and this option to the music above, leads to the following result



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

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

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

leading to



Appendix A Templates

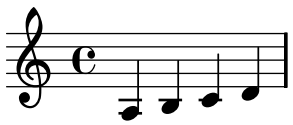
This section of the manual contains templates with the LilyPond score already set up for you. Just add notes, run LilyPond, and enjoy beautiful printed scores!

A.1 Single staff

A.1.1 Notes only

This very simple template gives you a staff with notes, suitable for a solo instrument or a melodic fragment. Cut and paste this into a file, add notes, and you're finished!

```
\version "2.12.3"
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

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

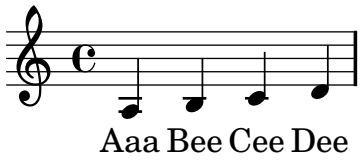


A.1.2 Notes and lyrics

This small template demonstrates a simple melody with lyrics. Cut and paste, add notes, then words for the lyrics. This example turns off automatic beaming, which is common for vocal parts. To use automatic beaming, change or comment out the relevant line.

```
\version "2.12.3"
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

text = \lyricmode {
  Aaa Bee Cee Dee
}

\score{
  <<
    \new Voice = "one" {
```

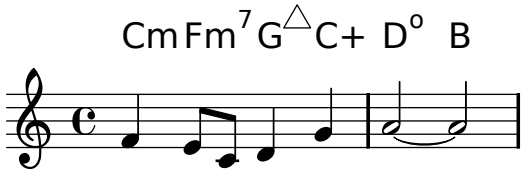
```
      \autoBeamOff
      \melody
    }
    \new Lyrics \lyricsto "one" \text
  >>
  \layout { }
  \midi { }
}
```



A.1.3 Notes and chords

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

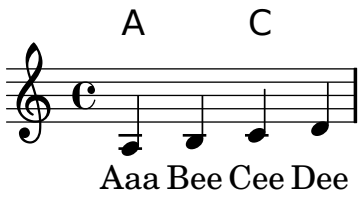
```
melody = \relative c' {
  \clef treble
  \key c \major
  \time 4/4

  f4 e8[ c] d4 g
  a2 ~ a
}

harmonies = \chordmode {
  c4:m f:min7 g:maj c:aug
  d2:dim b:sus
}

\score {
  <<
    \new ChordNames {
      \set chordChanges = ##t
      \harmonies
    }
    \new Staff \melody
  >>
  \layout{ }
  \midi { }
}
```



A.1.4 Notes, lyrics, and chords.

This template allows the preparation of a song with melody, words, and chords.

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

  a4 b c d
}

text = \lyricmode {
  Aaa Bee Cee Dee
}

harmonies = \chordmode {
  a2 c
}

\score {
  <<
    \new ChordNames {
      \set chordChanges = ##t
      \harmonies
    }
    \new Voice = "one" { \autoBeamOff \melody }
    \new Lyrics \lyricsto "one" \text
  >>
  \layout { }
  \midi { }
}
```



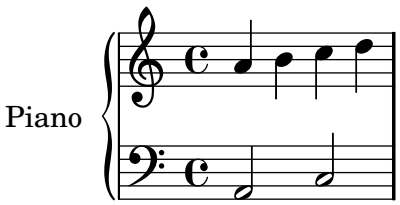
A.2 Piano templates

A.2.1 Solo piano

Here is a simple piano staff with some notes.

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

  a4 b c d
}
```

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

  a2 c
}

\score {
  \new PianoStaff <<
    \set PianoStaff.instrumentName = #"Piano  "
    \new Staff = "upper" \upper
    \new Staff = "lower" \lower
  >>
  \layout { }
  \midi { }
}
```

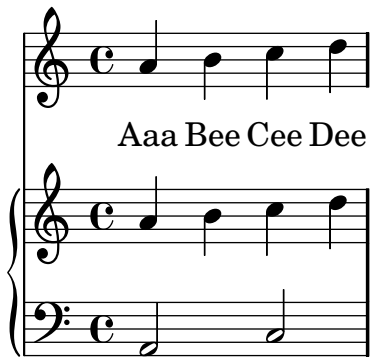


A.2.2 Piano and melody with lyrics

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

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

  a b c d
}

text = \lyricmode {
  Aaa Bee Cee Dee
}

upper = \relative c'' {
  \clef treble
  \key c \major
  \time 4/4

  a4 b c d
}

lower = \relative c {
  \clef bass
```

```
  \key c \major
  \time 4/4

  a2 c
}

\score {
  <<
    \new Voice = "mel" { \autoBeamOff \melody }
    \new Lyrics \lyricsto mel \text
    \new PianoStaff <<
      \new Staff = "upper" \upper
      \new Staff = "lower" \lower
    >>
  >>
  \layout {
    \context { \RemoveEmptyStaffContext }
  }
  \midi { }
}
```

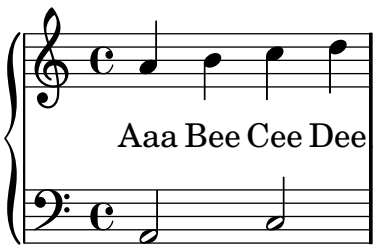


A.2.3 Piano centered lyrics

Instead of having a full staff for the melody and lyrics, lyrics can be centered between the staves of a piano staff.

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

  a4 b c d
}

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

  a2 c
```

```
}

text = \lyricmode {
  Aaa Bee Cee Dee
}

\score {
  \new GrandStaff <<
    \new Staff = upper { \new Voice = "singer" \upper }
    \new Lyrics \lyricsto "singer" \text
    \new Staff = lower { \lower }
  >>
  \layout {
    \context {
      \GrandStaff
      \accepts "Lyrics"
    }
    \context {
      \Lyrics
      \consists "Bar_engraver"
    }
  }
  \midi { }
}
```



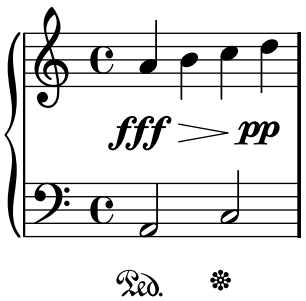
A.2.4 Piano centered dynamics

Many piano scores have the dynamics centered between the two staves. This requires a bit of tweaking to implement, but since the template is right here, you don't have to do the tweaking yourself.

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

upper = \relative c'' {
  \clef treble
  a4 b c d
}

lower = \relative c {
  \clef bass
  a2 c
```

```
}

dynamics = {
  s2\fff\> s4 s\!\pp
}

pedal = {
  s2\sustainOn s\sustainOff
}

\score {
  \new PianoStaff = "PianoStaff_pf" <<
    \new Staff = "Staff_pfUpper" << \global \upper >>
    \new Dynamics = "Dynamics_pf" \dynamics
    \new Staff = "Staff_pfLower" << \global \lower >>
    \new Dynamics = "pedal" \pedal
  >>

  \layout {
    % define Dynamics context
    \context {
      \type "Engraver_group"
      \name Dynamics
      \alias Voice
      \consists "Output_property_engraver"
      \consists "Piano_pedal_engraver"
      \consists "Script_engraver"
      \consists "New_dynamic_engraver"
      \consists "Dynamic_align_engraver"
      \consists "Text_engraver"
      \consists "Skip_event_swallow_translator"
      \consists "Axis_group_engraver"

      pedalSustainStrings = #'("Ped." "*Ped." "*")
      pedalUnaCordaStrings = #'("una corda" "" "tre corde")
      \override DynamicLineSpanner #'Y-offset = #0
      \override TextScript #'font-size = #2
      \override TextScript #'font-shape = #'italic
      \override VerticalAxisGroup #'minimum-Y-extent = #'(-1 . 1)
    }
    % modify PianoStaff context to accept Dynamics context
    \context {
      \PianoStaff
      \accepts Dynamics
    }
  }
}

\score {
  \new PianoStaff = "PianoStaff_pf" <<
    \new Staff = "Staff_pfUpper" << \global \upper \dynamics \pedal >>
    \new Staff = "Staff_pfLower" << \global \lower \dynamics \pedal >>
```

```
  >>
  \midi { }
}
```



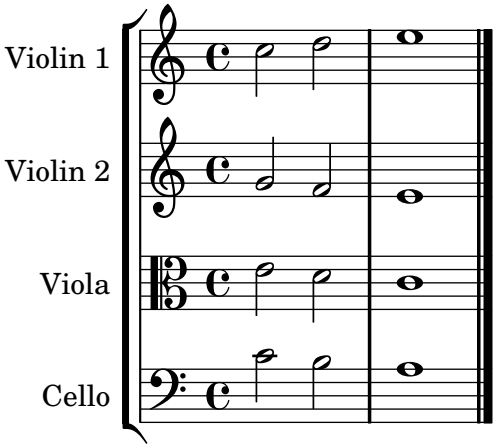
A.3 String quartet

A.3.1 String quartet

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

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

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

  c2 d
  e1

  \bar "|."
}

violinTwo = \new Voice \relative c'' {
  \set Staff.instrumentName = #"Violin 2 "

  g2 f
  e1

  \bar "|."
}

viola = \new Voice \relative c' {
  \set Staff.instrumentName = #"Viola "
  \clef alto

  e2 d
  c1

  \bar "|."
}
```

```
cello = \new Voice \relative c' {
  \set Staff.instrumentName = #"Cello "
  \clef bass

  c2 b
  a1

  \bar "|."
}

\score {
  \new StaffGroup <<
    \new Staff << \global \violinOne >>
    \new Staff << \global \violinTwo >>
    \new Staff << \global \viola >>
    \new Staff << \global \cello >>
  >>
  \layout { }
  \midi { }
}
```



A.3.2 String quartet parts

The "String quartet template" snippet produces a nice string quartet, but what if you needed to print parts? This new template demonstrates how to use the `\tag` feature to easily split a piece into individual parts.

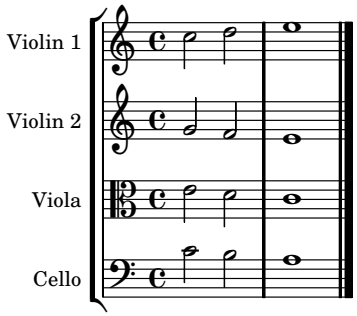
You need to split this template into separate files; the filenames are contained in comments at the beginning of each file. `piece.ly` contains all the music definitions. The other files – `score.ly`, `vn1.ly`, `vn2.ly`, `vla.ly`, and `vlc.ly` – produce the appropriate part.

Do not forget to remove specified comments when using separate files!

```
%%%%% piece.ly
%%%%% (This is the global definitions file)

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

```
}

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

  c2 d e1

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

  g2 f e1

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

  e2 d c1

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

  c2 b a1

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

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

%%% These are the other files you need to save on your computer

%%%%% score.ly
%%%%% (This is the main file)


%\include "piece.ly"             %%% uncomment this line when using a separate file
#(set-global-staff-size 14)
\score {
  \new StaffGroup \keepWithTag #'score \music
  \layout { }
  \midi { }
}
```

```
%{ Uncomment this block when using separate files

%%%%% vn1.ly
%%%%% (This is the Violin 1 part file)

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


%%%%% vn2.ly
%%%%% (This is the Violin 2 part file)

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


%%%%% vla.ly
%%%%% (This is the Viola part file)

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


%%%%% vlc.ly
%%%%% (This is the Cello part file)

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

%}
```



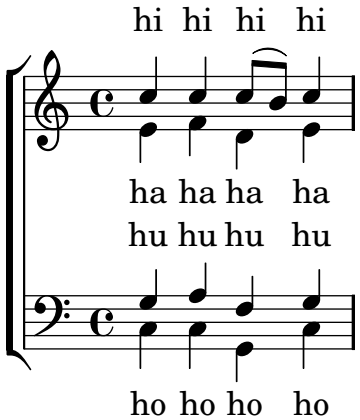
A.4 Vocal ensembles

A.4.1 SATB vocal score

Here is a standard four-part SATB vocal score. With larger ensembles, it is often useful to include a section which is included in all parts. For example, the time signature and key signature are almost always the same for all parts. Like in the "Hymn" template, the four voices are regrouped on only two staves.

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

sopMusic = \relative c'' {
  c4 c c8[( b)] c4
}
sopWords = \lyricmode {
  hi hi hi hi
}

altoMusic = \relative c' {
  e4 f d e
}
altoWords = \lyricmode {
  ha ha ha ha
}

tenorMusic = \relative c' {
  g4 a f g
}
tenorWords = \lyricmode {
  hu hu hu hu
}

bassMusic = \relative c {
  c4 c g c
}
bassWords = \lyricmode {
  ho ho ho ho
}

\score {
  \new ChoirStaff <<
    \new Lyrics = sopranos { s1 }
```

```
    \new Staff = women <<
      \new Voice = "sopranos" {
        \voiceOne
        << \global \sopMusic >>
      }
      \new Voice = "altos" {
        \voiceTwo
        << \global \altoMusic >>
      }
    >>
    \new Lyrics = "altos" { s1 }
    \new Lyrics = "tenors" { s1 }
    \new Staff = men <<
      \clef bass
      \new Voice = "tenors" {
        \voiceOne
        << \global \tenorMusic >>
      }
      \new Voice = "basses" {
        \voiceTwo << \global \bassMusic >>
      }
    >>
    \new Lyrics = basses { s1 }
    \context Lyrics = sopranos \lyricsto sopranos \sopWords
    \context Lyrics = altos \lyricsto altos \altoWords
    \context Lyrics = tenors \lyricsto tenors \tenorWords
    \context Lyrics = basses \lyricsto basses \bassWords
  >>
  \layout {
    \context {
      % a little smaller so lyrics
      % can be closer to the staff
      \Staff
      \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
    }
  }
}
```

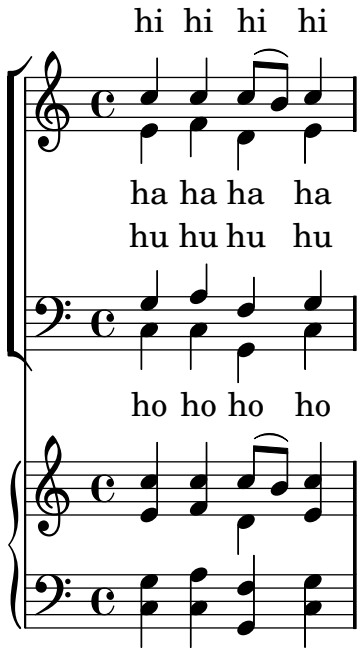


A.4.2 SATB vocal score and automatic piano reduction

This template adds an automatic piano reduction to the standard SATB vocal score demonstrated in "Vocal ensemble template". This demonstrates one of the strengths of LilyPond – you can use a music definition more than once. If any changes are made to the vocal notes (say, `tenorMusic`), then the changes will also apply to the piano reduction.

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

sopMusic = \relative c'' {
  c4 c c8[( b)] c4
}
sopWords = \lyricmode {
  hi hi hi hi
}

altoMusic = \relative c' {
  e4 f d e
}
altoWords =\lyricmode {
  ha ha ha ha
}

tenorMusic = \relative c' {
  g4 a f g
}
tenorWords = \lyricmode {
  hu hu hu hu
}

bassMusic = \relative c {
  c4 c g c
}
bassWords = \lyricmode {
  ho ho ho ho
}

\score {
  <<
    \new ChoirStaff <<
      \new Lyrics = sopranos { s1 }
      \new Staff = women <<
        \new Voice = sopranos { \voiceOne << \global \sopMusic >> }
        \new Voice = altos { \voiceTwo << \global \altoMusic >> }
      >>
      \new Lyrics = altos { s1 }
      \new Lyrics = tenors { s1 }
      \new Staff = men <<
        \clef bass
        \new Voice = tenors { \voiceOne <<\global \tenorMusic >> }
```

```
        \new Voice = basses { \voiceTwo <<\global \bassMusic >> }
      >>
      \new Lyrics = basses { s1 }
      \context Lyrics = sopranos \lyricsto sopranos \sopWords
      \context Lyrics = altos \lyricsto altos \altoWords
      \context Lyrics = tenors \lyricsto tenors \tenorWords
      \context Lyrics = basses \lyricsto basses \bassWords
    >>
    \new PianoStaff <<
      \new Staff <<
        \set Staff.printPartCombineTexts = ##f
        \partcombine
        << \global \sopMusic >>
        << \global \altoMusic >>
      >>
      \new Staff <<
        \clef bass
        \set Staff.printPartCombineTexts = ##f
        \partcombine
        << \global \tenorMusic >>
        << \global \bassMusic >>
      >>
    >>
  >>
  \layout {
    \context {
      % a little smaller so lyrics
      % can be closer to the staff
      \Staff
      \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
    }
  }
}
```

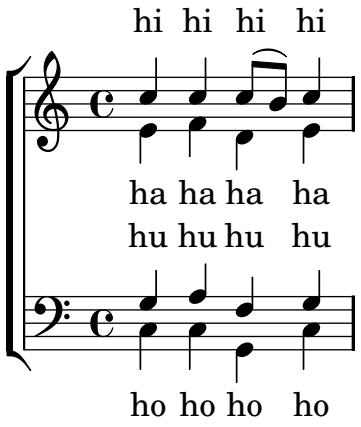


A.4.3 SATB with aligned contexts

This template is basically the same as the simple "Vocal ensemble" template, with the exception that here all the lyrics lines are placed using `alignAboveContext` and `alignBelowContext`.

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

sopMusic = \relative c'' {
  c4 c c8[( b)] c4
}
sopWords = \lyricmode {
  hi hi hi hi
}

altoMusic = \relative c' {
  e4 f d e
}
altoWords = \lyricmode {
  ha ha ha ha
}

tenorMusic = \relative c' {
  g4 a f g
}
tenorWords = \lyricmode {
  hu hu hu hu
}

bassMusic = \relative c {
  c4 c g c
}
bassWords = \lyricmode {
  ho ho ho ho
```

```
}

\score {
  \new ChoirStaff <<
    \new Staff = women <<
      \new Voice = "sopranos" { \voiceOne << \global \sopMusic >> }
      \new Voice = "altos" { \voiceTwo << \global \altoMusic >> }
    >>
    \new Lyrics \with { alignAboveContext = women } \lyricsto sopranos \sopWords
    \new Lyrics \with { alignBelowContext = women } \lyricsto altos \altoWords
    % we could remove the line about this with the line below, since we want
    % the alto lyrics to be below the alto Voice anyway.
    % \new Lyrics \lyricsto altos \altoWords

    \new Staff = men <<
      \clef bass
      \new Voice = "tenors" { \voiceOne << \global \tenorMusic >> }
      \new Voice = "basses" { \voiceTwo << \global \bassMusic >> }
    >>
    \new Lyrics \with { alignAboveContext = men } \lyricsto tenors \tenorWords
    \new Lyrics \with { alignBelowContext = men } \lyricsto basses \bassWords
    % again, we could replace the line above this with the line below.
    % \new Lyrics \lyricsto basses \bassWords
  >>
  \layout {
    \context {
      % a little smaller so lyrics
      % can be closer to the staff
      \Staff
      \override VerticalAxisGroup #'minimum-Y-extent = #'(-3 . 3)
    }
  }
}
```



A.5 Ancient notation templates

A.5.1 Transcription of mensural music

When transcribing mensural music, an incipit at the beginning of the piece is useful to indicate the original key and tempo. While today musicians are used to bar lines in order to faster recognize rhythmic patterns, bar lines were not yet invented during the period of mensural music; in fact, the meter often changed after every few notes. As a compromise, bar lines are often printed between the staves rather than on the staves.

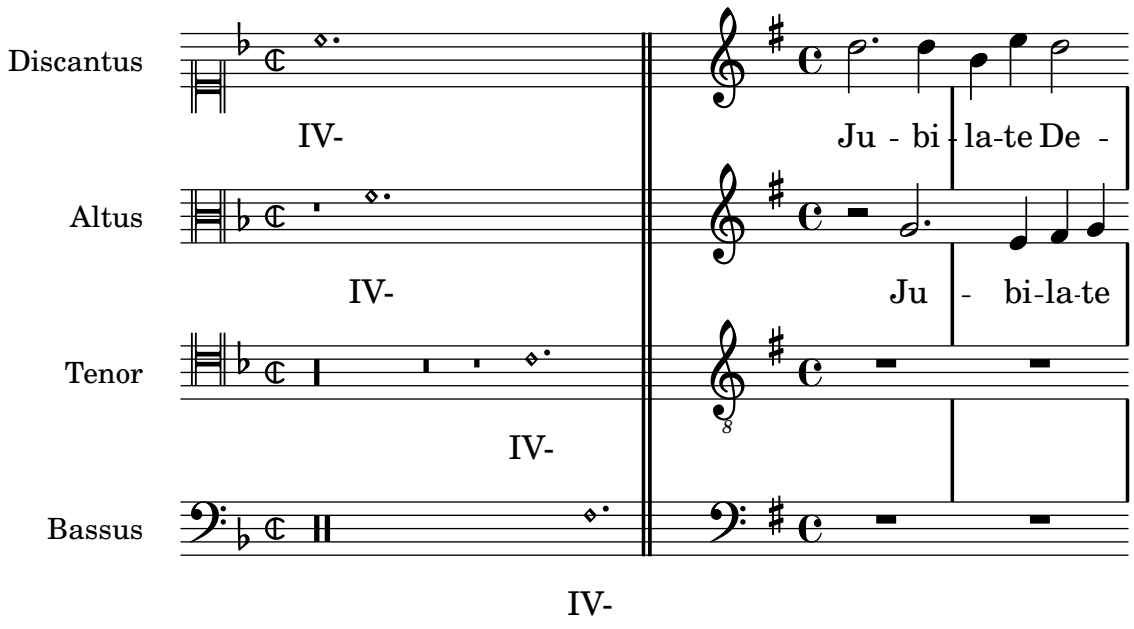
```
global = {
  \set Score.skipBars = ##t

  % incipit
  \once \override Score.SystemStartBracket #'transparent = ##t
  \override Score.SpacingSpanner #'spacing-increment = #1.0 % tight spacing
  \key f \major
  \time 2/2
  \once \override Staff.TimeSignature #'style = #'neomensural
  \override Voice.NoteHead #'style = #'neomensural
  \override Voice.Rest #'style = #'neomensural
  \set Staff.printKeyCancellation = ##f
  \cadenzaOn % turn off bar lines
  \skip 1*10
  \once \override Staff.BarLine #'transparent = ##f
  \bar "||"
  \skip 1*1 % need this extra \skip such that clef change comes
            % after bar line
  \bar ""

  % main
  \revert Score.SpacingSpanner #'spacing-increment % CHECK: no effect?
  \cadenzaOff % turn bar lines on again
  \once \override Staff.Clef #'full-size-change = ##t
  \set Staff.forceClef = ##t
  \key g \major
  \time 4/4
  \override Voice.NoteHead #'style = #'default
  \override Voice.Rest #'style = #'default

  % FIXME: setting printKeyCancellation back to #t must not
  % occur in the first bar after the incipit.  Dto. for forceClef.
  % Therefore, we need an extra \skip.
  \skip 1*1
  \set Staff.printKeyCancellation = ##t
  \set Staff.forceClef = ##f

  \skip 1*7 % the actual music

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

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

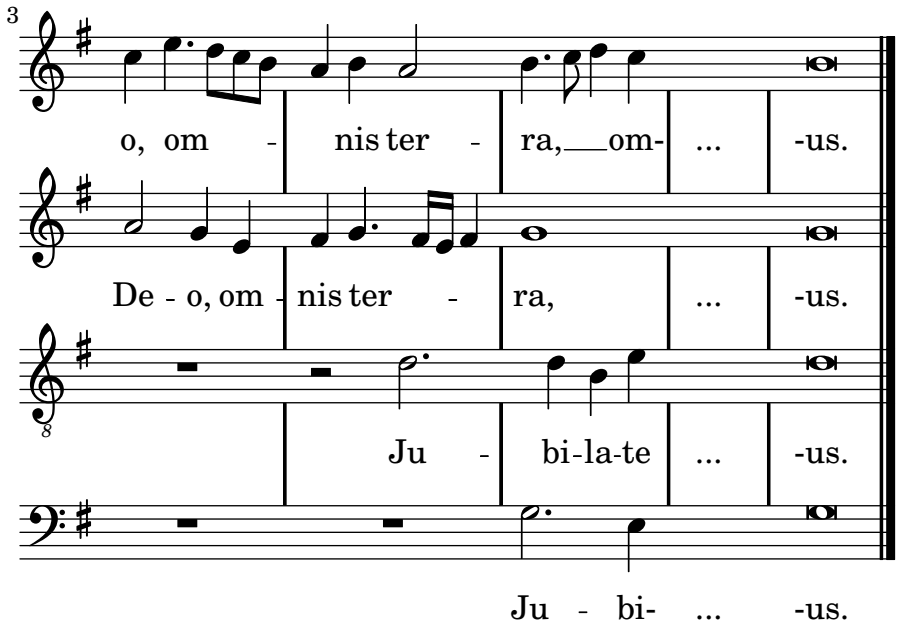
```
discantusNotes = {
  \transpose c' c'' {
    \set Staff.instrumentName = #"Discantus  "

    % incipit
    \clef "neomensural-c1"
    c'1. s2   % two bars
    \skip 1*8 % eight bars
    \skip 1*1 % one bar

    % main
    \clef "treble"
    d'2. d'4 |
    b e' d'2 |
    c'4 e'4.( d'8 c' b |
    a4) b a2 |
    b4.( c'8 d'4) c'4 |
    \once \override NoteHead #'transparent = ##t c'1 |
    b\breve |
  }
}

discantusLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi -- |
  la -- te De -- |
  o, om --
  nis ter -- |
  ra, __ om- |
  "..." |
  -us. |
}

altusNotes = {
  \transpose c' c'' {
    \set Staff.instrumentName = #"Altus  "

    % incipit
    \clef "neomensural-c3"
    r1        % one bar
    f1. s2    % two bars
    \skip 1*7 % seven bars
    \skip 1*1 % one bar

    % main
    \clef "treble"
    r2 g2. e4 fis g | % two bars
    a2 g4 e |
```

```
    fis g4.( fis16 e fis4) |
    g1 |
    \once \override NoteHead #'transparent = ##t g1 |
    g\breve |
  }
}

altusLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi -- la -- te | % two bars
  De -- o, om -- |
  nis ter -- ra, |
  "..." |
  -us. |
}

tenorNotes = {
  \transpose c' c' {
    \set Staff.instrumentName = #"Tenor  "

    % incipit
    \clef "neomensural-c4"
    r\longa   % four bars
    r\breve   % two bars
    r1        % one bar
    c'1. s2   % two bars
    \skip 1*1 % one bar
    \skip 1*1 % one bar

    % main
    \clef "treble_8"
    R1 |
    R1 |
    R1 |
    r2 d'2. d'4 b e' | % two bars
    \once \override NoteHead #'transparent = ##t e'1 |
    d'\breve |
  }
}

tenorLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi -- la -- te | % two bars
  "..." |
  -us. |
}
```

```
bassusNotes = {
  \transpose c' c' {
    \set Staff.instrumentName = #"Bassus  "

    % incipit
    \clef "bass"
    r\maxima  % eight bars
    f1. s2    % two bars
    \skip 1*1 % one bar

    % main
    \clef "bass"
    R1 |
    R1 |
    R1 |
    R1 |
    g2. e4 |
    \once \override NoteHead #'transparent = ##t e1 |
    g\breve |
  }
}

bassusLyrics = \lyricmode {
  % incipit
  IV-

  % main
  Ju -- bi- |
  "..." |
  -us. |
}

\score {
  \new StaffGroup = choirStaff <<
    \new Voice =
      "discantusNotes" << \global \discantusNotes >>
    \new Lyrics =
      "discantusLyrics" \lyricsto discantusNotes { \discantusLyrics }
    \new Voice =
      "altusNotes" << \global \altusNotes >>
    \new Lyrics =
      "altusLyrics" \lyricsto altusNotes { \altusLyrics }
    \new Voice =
      "tenorNotes" << \global \tenorNotes >>
    \new Lyrics =
      "tenorLyrics" \lyricsto tenorNotes { \tenorLyrics }
    \new Voice =
      "bassusNotes" << \global \bassusNotes >>
    \new Lyrics =
      "bassusLyrics" \lyricsto bassusNotes { \bassusLyrics }
  >>
```

```
  \layout {
    \context {
      \Score

      % no bars in staves
      \override BarLine #'transparent = ##t

      % incipit should not start with a start delimiter
      \remove "System_start_delimiter_engraver"
    }
    \context {
      \Voice

      % no slurs
      \override Slur #'transparent = ##t

      % Comment in the below "\remove" command to allow line
      % breaking also at those barlines where a note overlaps
      % into the next bar.  The command is commented out in this
      % short example score, but especially for large scores, you
      % will typically yield better line breaking and thus improve
      % overall spacing if you comment in the following command.
      %\remove "Forbid_line_break_engraver"
    }
  }
}
```



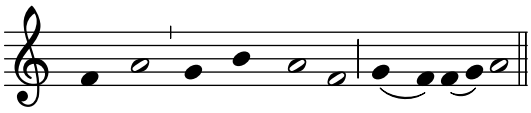


A.5.2 Gregorian transcription template

This example demonstrates how to do modern transcription of Gregorian music. Gregorian music has no measure, no stems; it uses only half and quarter note heads, and special marks, indicating rests of different length.

```
\include "gregorian.ly"

chant = \relative c' {
  \set Score.timing = ##f
  f4 a2 \divisioMinima
  g4 b a2 f2 \divisioMaior
  g4( f) f( g) a2 \finalis
}

verba = \lyricmode {
  Lo -- rem ip -- sum do -- lor sit a -- met
}

\score {
  \new Staff <<
    \new Voice = "melody" \chant
    \new Lyrics = "one" \lyricsto melody \verba
  >>
  \layout {
    \context {
      \Staff
      \remove "Time_signature_engraver"
      \remove "Bar_engraver"
      \override Stem #'transparent = ##t
    }
    \context {
      \Voice
      \override Stem #'length = #0
    }
    \context {
      \Score
```

```
      barAlways = ##t
    }
  }
}
```



A.6 Jazz combo

This is quite an advanced template, for a jazz ensemble. Note that all instruments are notated in `\key c \major`. This refers to the key in concert pitch; the key will be automatically transposed if the music is within a `\transpose` section.

```
\header {
  title = "Song"
  subtitle = "(tune)"
  composer = "Me"
  meter = "moderato"
  piece = "Swing"
  tagline = \markup {
    \column {
      "LilyPond example file by Amelie Zapf,"
      "Berlin 07/07/2003"
    }
  }
}

%#(set-global-staff-size 16)
\include "english.ly"

%%%%%%%%%%% Some macros %%%%%%%%%%%%%%%%%%

sl = {
  \override NoteHead #'style = #'slash
  \override Stem #'transparent = ##t
}
nsl = {
  \revert NoteHead #'style
  \revert Stem #'transparent
}
crOn = \override NoteHead #'style = #'cross
crOff = \revert NoteHead #'style

%% insert chord name style stuff here.

jazzChords = { }

%%%%%%%%%%% Keys'n'thangs %%%%%%%%%%%%%%%%
```

```
global = { \time 4/4 }

Key = { \key c \major }

% ############ Horns ############

% ------ Trumpet ------
trpt = \transpose c d \relative c'' {
  \Key
  c1 | c | c |
}
trpHarmony = \transpose c' d {
  \jazzChords
}
trumpet = {
  \global
  \set Staff.instrumentName = #"Trumpet"
  \clef treble
  <<
    \trpt
  >>
}

% ------ Alto Saxophone ------
alto = \transpose c a \relative c' {
  \Key
  c1 | c | c |
}
altoHarmony = \transpose c' a {
  \jazzChords
}
altoSax = {
  \global
  \set Staff.instrumentName = #"Alto Sax"
  \clef treble
  <<
    \alto
  >>
}

% ------ Baritone Saxophone ------
bari = \transpose c a' \relative c {
  \Key
  c1
  c1
  \sl
  d4^"Solo" d d d
  \nsl
}
bariHarmony = \transpose c' a \chordmode {
  \jazzChords s1 s d2:maj e:m7
}
```

```
bariSax = {
  \global
  \set Staff.instrumentName = #"Bari Sax"
  \clef treble
  <<
    \bari
  >>
}

% ------ Trombone ------
tbone = \relative c {
  \Key
  c1 | c | c
}
tboneHarmony = \chordmode {
  \jazzChords
}
trombone = {
  \global
  \set Staff.instrumentName = #"Trombone"
  \clef bass
  <<
    \tbone
  >>
}

% ############ Rhythm Section #############

% ------ Guitar ------
gtr = \relative c'' {
  \Key
  c1
  \sl
  b4 b b b
  \nsl
  c1
}
gtrHarmony = \chordmode {
  \jazzChords
  s1 c2:min7+ d2:maj9
}
guitar = {
  \global
  \set Staff.instrumentName = #"Guitar"
  \clef treble
  <<
    \gtr
  >>
}

%% ------ Piano ------
rhUpper = \relative c'' {
```

```
  \voiceOne
  \Key
  c1 | c | c
}
rhLower = \relative c' {
  \voiceTwo
  \Key
  e1 | e | e
}

lhUpper = \relative c' {
  \voiceOne
  \Key
  g1 | g | g
}
lhLower = \relative c {
  \voiceTwo
  \Key
  c1 | c | c
}

PianoRH = {
  \clef treble
  \global
  \set Staff.midiInstrument = #"acoustic grand"
  <<
    \new Voice = "one" \rhUpper
    \new Voice = "two" \rhLower
  >>
}
PianoLH = {
  \clef bass
  \global
  \set Staff.midiInstrument = "acoustic grand"
  <<
    \new Voice = "one" \lhUpper
    \new Voice = "two" \lhLower
  >>
}

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

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

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

% ------ Drums ------
up = \drummode {
  \voiceOne
  hh4 <hh sn> hh <hh sn>
  hh4 <hh sn> hh <hh sn>
  hh4 <hh sn> hh <hh sn>
}
down = \drummode {
  \voiceTwo
  bd4 s bd s
  bd4 s bd s
  bd4 s bd s
}

drumContents = {
  \global
  <<
    \set DrumStaff.instrumentName = #"Drums"
    \new DrumVoice \up
    \new DrumVoice \down
  >>
}

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

\score {
  <<
    \new StaffGroup = "horns" <<
      \new Staff = "trumpet" \trumpet
      \new Staff = "altosax" \altoSax
      \new ChordNames = "barichords" \bariHarmony
      \new Staff = "barisax" \bariSax
      \new Staff = "trombone" \trombone
    >>

    \new StaffGroup = "rhythm" <<
      \new ChordNames = "chords" \gtrHarmony
      \new Staff = "guitar" \guitar
      \new PianoStaff = "piano" \piano
      \new Staff = "bass" \bass
      \new DrumStaff \drumContents
    >>
```

```
  >>

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

  \midi { }
}
```

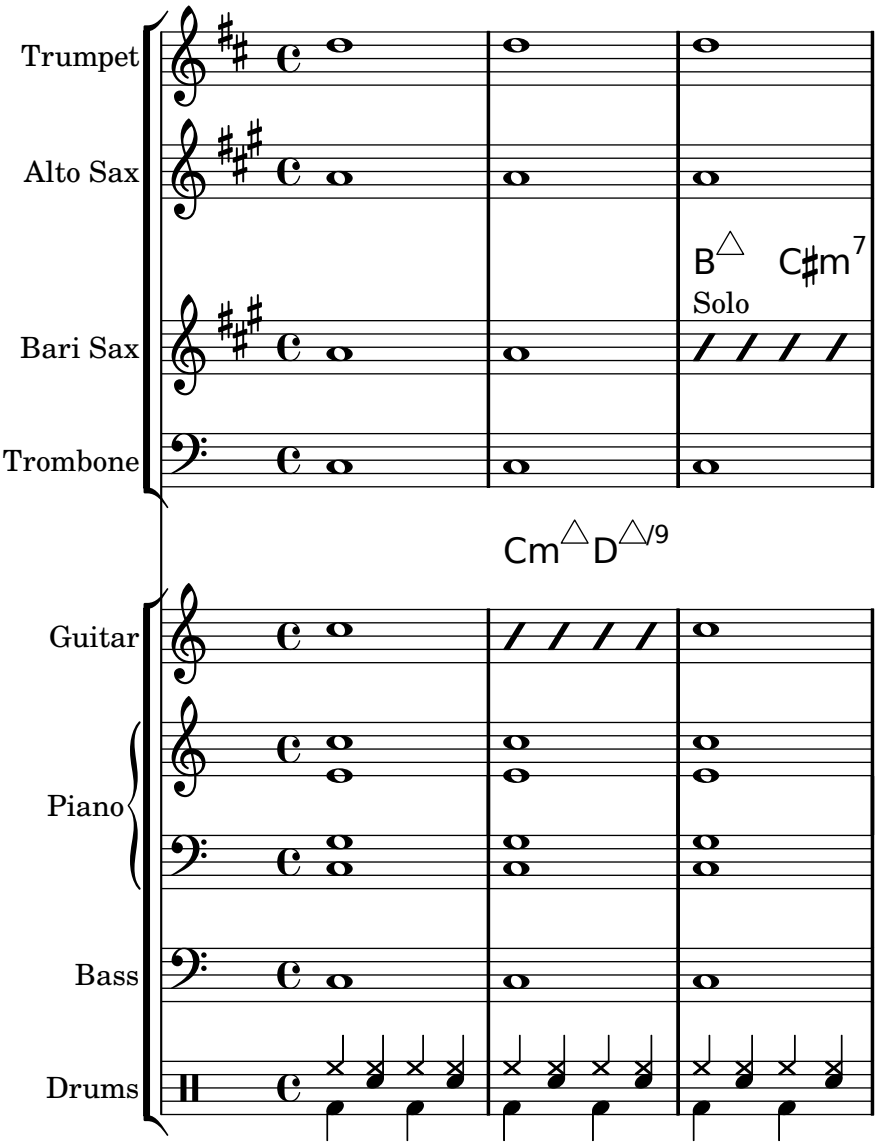
Song

(tune)

Me

moderato

Swing



A.7 lilypond-book templates

These templates are for use with `lilypond-book`. If you're not familiar with this program, please refer to Section "LilyPond-book" in *Application Usage*.

A.7.1 LaTeX

You can include LilyPond fragments in a LaTeX document.

```
\documentclass[]{article}

\begin{document}

Normal LaTeX text.

\begin{lilypond}
\relative c'' {
a4 b c d
}
\end{lilypond}

More LaTeX text.
```

```
\begin{lilypond}
\relative c'' {
d4 c b a
}
\end{lilypond}
\end{document}
```

A.7.2 Texinfo

You can include LilyPond fragments in Texinfo; in fact, this entire manual is written in Texinfo.

```
\input texinfo
@node Top

Texinfo text

@lilypond[verbatim,fragment,ragged-right]
a4 b c d
@end lilypond

More Texinfo text

@lilypond[verbatim,fragment,ragged-right]
d4 c b a
@end lilypond

@bye
```

A.7.3 xelatex

```
\documentclass{article}
\usepackage{ifxetex}
\ifxetex
%xetex specific stuff
\usepackage{xunicode,fontspec,xltxtra}
\setmainfont[Numbers=OldStyle]{Times New Roman}
\setsansfont{Arial}
\else
%This can be empty if you are not going to use pdftex
\usepackage[T1]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage{mathptmx}%Times
\usepackage{helvet}%Helvetica
\fi
%Here you can insert all packages that pdftex also understands
\usepackage[ngerman,finnish,english]{babel}
\usepackage{graphicx}

\begin{document}
\title{A short document with LilyPond and xelatex}
\maketitle

Normal \textbf{font} commands inside the \emph{text} work,
because they \textsf{are supported by \LaTeX{} and XeteX.}
```

```
If you want to use specific commands like \verb+\XeTeX+, you
should include them again in a \verb+\ifxetex+ environment.
You can use this to print the \ifxetex \XeTeX{} command \else
XeTeX command \fi which is not known to normal \LaTeX .

In normal text you can easily use LilyPond commands, like this:

\begin{lilypond}
{a2 b c'8 c' c' c'}
\end{lilypond}

\noindent
and so on.

The fonts of snippets set with LilyPond will have to be set from
inside
of the snippet. For this you should read the AU on how to use
lilypond-book.

\selectlanguage{ngerman}
Auch Umlaute funktionieren ohne die \LaTeX -Befehle, wie auch alle
anderen
seltsamen Zeichen: __ ______, wenn sie von der Schriftart
unterst__tzt werden.
\end{document}
```

Appendix B Scheme tutorial

LilyPond uses the Scheme programming language, both as part of the input syntax, and as internal mechanism to glue modules of the program together. This section is a very brief overview of entering data in Scheme. If you want to know more about Scheme, see http://www.schemers.org.

The most basic thing of a language is data: numbers, character strings, lists, etc. Here is a list of data types that are relevant to LilyPond input.

Booleans Boolean values are True or False. The Scheme for True is `#t` and False is `#f`.

Numbers Numbers are entered in the standard fashion, `1` is the (integer) number one, while `-1.5` is a floating point number (a non-integer number).

Strings Strings are enclosed in double quotes,

```
"this is a string"
```

Strings may span several lines

```
"this
is
a string"
```

Quotation marks and newlines can also be added with so-called escape sequences. The string `a said "b"` is entered as

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

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

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

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

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

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

```
#(+ 1 2)
  ⇒ #3
```

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

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

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

```
twelve = #12
```

variables can also be used in expressions, here

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

the number 24 is stored in the variable `twentyFour`. The same assignment can be done in completely in Scheme as well,

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

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

```
#'twentyFour
```

The quote mark `'` prevents the Scheme interpreter from substituting `24` for the `twentyFour`. Instead, we get the name `twentyFour`.

This syntax will be used very frequently, since many of the layout tweaks involve assigning (Scheme) values to internal variables, for example

```
\override Stem #'thickness = #2.6
```

This instruction adjusts the appearance of stems. The value `2.6` is put into the `thickness` variable of a `Stem` object. `thickness` is measured relative to the thickness of staff lines, so these stem lines will be `2.6` times the width of staff lines. This makes stems almost twice as thick as their normal size. To distinguish between variables defined in input files (like `twentyFour` in the example above) and variables of internal objects, we will call the latter 'properties' and the former 'variables.' So, the stem object has a `thickness` property, while `twentyFour` is an variable.

Two-dimensional offsets (X and Y coordinates) as well as object sizes (intervals with a left and right point) are entered as `pairs`. A pair[1] is entered as `(first . second)` and, like symbols, they must be quoted,

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

This assigns the pair (1, 2) to the `extra-offset` property of the TextScript object. These numbers are measured in staff-spaces, so this command moves the object 1 staff space to the right, and 2 spaces up.

The two elements of a pair may be arbitrary values, for example

```
#'(1 . 2)
#'(#t . #f)
#'("blah-blah" . 3.14159265)
```

A list is entered by enclosing its elements in parentheses, and adding a quote. For example,

```
#'(1 2 3)
#'(1 2 "string" #f)
```

We have been using lists all along. A calculation, like `(+ 1 2)` is also a list (containing the symbol `+` and the numbers 1 and 2). Normally lists are interpreted as calculations, and the Scheme interpreter substitutes the outcome of the calculation. To enter a list, we stop the evaluation. This is done by quoting the list with a quote `'` symbol. So, for calculations do not use a quote.

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

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

B.1 Tweaking with Scheme

We have seen how LilyPond output can be heavily modified using commands like `\override TextScript #'extra-offset = ( 1 . -1)`. But we have even more power if we use Scheme. For a full explanation of this, see the Appendix B [Scheme tutorial], page 173, and Section "Interfaces for programmers" in *Notation Reference*.

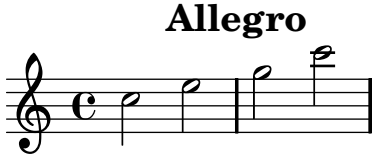
We can use Scheme to simply `\override` commands,

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

TODO Find a simple example

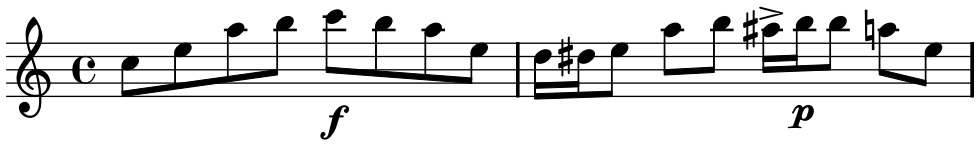
We can use it to create new commands:

```
tempoMark = #(define-music-function (parser location padding marktext)
                                    (number? string?)
#{
  \once \override Score . RehearsalMark #'padding = $padding
  \once \override Score . RehearsalMark #'extra-spacing-width = #'(+inf.0 . -inf.0)
  \mark \markup { \bold $marktext }
#})

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



Even music expressions can be passed in:

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

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



Appendix C GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

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

This License is a kind of 'copyleft', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The 'Document', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as 'you'.

A 'Modified Version' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A 'Secondary Section' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The 'Invariant Sections' are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The 'Cover Texts' are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A 'Transparent' copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file

format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not 'Transparent' is called 'Opaque'.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The 'Title Page' means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, 'Title Page' means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled 'History', and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled 'History' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the 'History' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled 'Acknowledgments' or 'Dedications', preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled 'Endorsements'. Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as 'Endorsements' or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to

the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled 'Endorsements', provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled 'History' in the various original documents, forming one section entitled 'History'; likewise combine any sections entitled 'Acknowledgments', and any sections entitled 'Dedications'. You must delete all sections entitled 'Endorsements.'

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an 'aggregate', and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License 'or any later version' applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
  Copyright (C)  year  your name.
  Permission is granted to copy, distribute and/or modify this document
  under the terms of the GNU Free Documentation License, Version 1.1
  or any later version published by the Free Software Foundation;
  with the Invariant Sections being list their titles, with the
  Front-Cover Texts being list, and with the Back-Cover Texts being list.
  A copy of the license is included in the section entitled ‘GNU
Free Documentation License’
.
```

If you have no Invariant Sections, write ‘with no Invariant Sections’ instead of saying which ones are invariant. If you have no Front-Cover Texts, write ‘no Front-Cover Texts’ instead of ‘Front-Cover Texts being *list*’; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix D LilyPond index

!

! 21

#

..... 173
##f 173
##t 173
#'symbol 174

%

% 16
%{ ... %} 16

'

' 13

(

(...) 19

,

, ... 13

.

... . 16

<

< 21
< 27
< ... > 27
<< 25, 28
<< ... >> 25
<< ... \\ ... >> 28
<< \\ >> 44

>

> 21
> 27
>> 25, 28

[

[..... 22
[...] 22

]

] 22

^

^ 21

_

_ 21

\

\! 21
\(... \) 19
\< 21
\> 21
\\ 28, 44
\acciaccatura 23
\addlyrics 28
\addlyrics example 85
\addlyrics, example 89
\appoggiatura 23
\autoBeamOff 22, 52
\autoBeamOn 22
\book 37
\book 38, 57
\clef 15
\consists 66
\context 65
\f 21
\ff 21
\grace 23
\header 34, 38
\key 18
\layout 38, 67
\lyricmode 52
\lyricsto 52
\major 18
\markup 22
\mf 21
\midi 38
\minor 18
\mp 21
\new 25, 59
\new ChoirStaff 52
\new Lyrics 52
\new Staff 25
\new Voice 48
\once 82, 87
\oneVoice 48
\override 81
\overrideProperty 82
\p 21
\partial 23
\pp 21
\relative 13
\remove 66
\revert 82, 87
\score 37
\score 39
\set 62
\set, example of using 102
\shiftOff 51
\shiftOn 51
\shiftOnn 51
\shiftOnnn 51
\startTextSpan 104

\stopTextSpan ... 104
\textLengthOff ... 106
\textLengthOn ... 106
\time ... 15
\times ... 23
\tweak ... 83
\tweak, example ... 83, 84
\unset ... 62
\version ... 34
\voiceFour ... 48
\voiceFourStyle ... 46
\voiceNeutralStyle ... 46
\voiceOne ... 48
\voiceOneStyle ... 46
\voiceThree ... 48
\voiceThreeStyle ... 46
\voiceTwo ... 48
\voiceTwoStyle ... 46
\with ... 65
\with, example ... 95, 96, 97, 98

~

~ ... 19

A

absolute mode ... 35
absolute note names ... 35
absolute values for pitches ... 35
accent ... 20
accessing Scheme ... 173
acciaccatura ... 23
accidental ... 13, 18
Accidental, example of overriding ... 112
AccidentalPlacement, example of overriding ... 112
accidentals ... 18
Accidentals ... 19
accidentals and key signature ... 18
accidentals and key signatures ... 18
accidentals and relative mode ... 13
adding engravers ... 66
adding text ... 22
addlyrics ... 28
alignAboveContext property, example ... 95, 96, 97, 98
aligning lyrics ... 29
aligning objects on a baseline ... 113
alto ... 15
ambitus ... 66
ambitus engraver ... 66
anacrusis ... 23
appendices ... 9
Application Usage ... 8, 9
appoggiatura ... 23
articulation ... 20
Articulations and ornamentations ... 21
articulations and slurs ... 105
assigning variables ... 33
autoBeamOff ... 22, 52
autoBeamOn ... 22
automated engraving ... 3
Automatic accidentals ... 19
automatic beams ... 22
Automatic beams ... 23

B

balance ... 2
Bar and bar number checks ... 131
bar numbers, tweaking placement ... 104
BarLine, example of overriding ... 91, 92, 93, 94
bass ... 15
beam ... 14, 22
Beam, example of overriding ... 115
beaming ... 22
beaming and lyrics ... 52
beams, automatic ... 22
beams, by hand ... 22
beams, controlling manually ... 114
beams, manual ... 22
blackness ... 2
block comment ... 16, 17
book ... 37
book ... 38, 57
book block, implicit ... 38
book, example of using ... 57
bound-details property, example ... 104, 105
brace ... 26
braces, curly ... 16
bracket types ... 43
bracket, triplet ... 84
bracket, tuplet ... 84
brackets, enclosing vs. marking ... 43
brackets, nesting ... 43
break-visibility property ... 92
break-visibility property, example ... 92

C

case sensitive ... 11, 16
center ... 100
Changing context default settings ... 66, 68
Changing defaults ... 9
changing size of objects ... 95
characters allowed in variables ... 33
charts, reference ... 9
Cheat sheet ... 9
choir staff ... 26, 52
ChoirStaff ... 26, 52
chord ... 27
ChordNames ... 25
chords ... 27
chords vs. voices ... 44
clef ... 15
Clef ... 16
Clef, example of overriding ... 94, 95, 96, 97, 98
clickable examples ... 17
Collision resolution ... 120
collisions, notes ... 51
color property ... 93
color property, example ... 82, 83, 84, 93, 94
color property, setting to Scheme procedure ... 129
color, rgb ... 94
color, X11 ... 93
combining expressions in parallel ... 25
comma ... 13
comment, line ... 16
comments ... 16, 17
compiling ... 11
compound music expression ... 24, 39

concurrent music ... 44
consists ... 66
constructing files, tips ... 17
content vs. layout ... 18
contents of a score block ... 39
context ... 25
context ... 65
context properties ... 62
context properties, modifying ... 62
context properties, setting with \context ... 65
context properties, setting with \with ... 65
context, finding ... 86
context, identifying correct ... 86
context, notation ... 25
context, specifying in lyric mode ... 89
context, Voice ... 44
contexts ... 5
Contexts ... 6, 63, 66
contexts explained ... 58
Contexts explained ... 59
contexts, creating ... 59
contexts, implicit ... 37
contexts, naming ... 61
controlling tuplets, slurs, phrasing slurs, and beams manually ... 114
convert-ly ... 34
Converting from other formats ... 9
creating contexts ... 59
Creating contexts ... 61
Creating MIDI files ... 38
Creating titles ... 35
crescendo ... 21
curly braces ... 16

D

decrescendo ... 21
default properties, reverting to ... 87
Difficult tweaks ... 83
direction property, example ... 84, 100, 101
Displaying staves ... 27
distances ... 98
dotted note ... 14
double flat ... 18
double sharp ... 18
down ... 100
duration ... 14
durations ... 14
DynamicLineSpanner, example of overriding ... 113
dynamics ... 21
Dynamics ... 21
dynamics, tweaking placement ... 107
DynamicText, example of overriding ... 108, 113

E

engraver ... 4
engravers ... 61
Engravers and Performers ... 62
engravers, adding ... 66
engravers, removing ... 66
engraving ... 2, 4
engraving multiple voices ... 5
engraving, automated ... 3
es ... 18
eses ... 18
evaluating Scheme ... 173
example of writing a score ... 76
example, first ... 11
examples, clickable ... 17
examples, simple ... 7
expression, music ... 24
expressions ... 16
expressions, parallel ... 25
extender line ... 29
extra-offset property ... 110
extra-offset property, example ... 114
extra-spacing-width ... 108
extra-spacing-width property ... 110
extra-spacing-width property, example ... 108, 113

F

FDL, GNU Free Documentation License ... 176
fermata, implementing in MIDI ... 125
file structure ... 37
File structure ... 37
File structure ... 39
files, tips for constructing ... 17
files, updating ... 34
fingering ... 21
fingering example ... 101, 102
Fingering instructions ... 21
fingering, chords ... 101
Fingering, example of overriding ... 101, 114
fingering, placement ... 101
fingeringOrientations property, example ... 102
first example ... 11
fixing overlapping notation ... 111
flat ... 18
flat, double ... 18
font ... 2
font-series property, example ... 126
font-shape property, example ... 89, 126
font-size property, example ... 83
fontSize property, example ... 97, 98
fontSize, default and setting ... 65
force-hshift property ... 110
force-hshift property, example ... 116, 122
foreign languages ... 8
formatting a score ... 4
formatting rules ... 4
future upgrades ... 34

G

General input and output ... 9
grace ... 23
grace notes ... 23
Grace notes ... 23
grand staff ... 26
GrandStaff ... 26
grob ... 80
grob sizing ... 108
grobs, moving colliding ... 109
grobs, positioning ... 114
grobs, properties of ... 84
GUILE ... 173

H

half note ... 14
`header` ... 34, 38
header block ... 34
headers ... 34
hiding objects ... 124
how to read the manual ... 17
hymn structure ... 53
hyphens ... 29

I

identifiers ... 33
idiom ... 8
implicit book block ... 38
implicit contexts ... 37
input format ... 37
Install ... 9
integrating LilyPond with other programs ... 9
interface ... 80, 88
interface properties ... 88
Interfaces for programmers ... 9, 174
Internals Reference ... 8, 10, 84
Internals Reference manual ... 84
Internals Reference, example of using ... 85
interval ... 13
invisible objects ... 124
`is` ... 18
`isis` ... 18
italic, example ... 89

J

jargon ... 8

K

`key` ... 18
key signature ... 18
Key signature ... 19
key signature, setting ... 18
Keyboard and other multi-staff instruments ... 27

L

language ... 8
layers ... 44
`layout` ... 38, 67
layout block, effect of location ... 38
layout object ... 80
layout objects, properties of ... 84
layout vs. content ... 18
Learning Manual ... 8
left-padding property ... 109, 112
length ... 98
LilyPond command index ... 9
LilyPond index ... 9
LilyPond Snippet Repository ... 9
LilyPond-book ... 9, 170
line comment ... 16, 17
LISP ... 173
List of colors ... 93, 94
Literature list ... 9
LSR ... 9
lyric mode, specifying context ... 89
`lyricmode` ... 52
lyrics ... 28
`Lyrics` ... 25, 52
lyrics and beaming ... 52
lyrics and multiple staves ... 32
Lyrics context, creating ... 52
lyrics, aligning ... 29
lyrics, linking to voice ... 52
lyrics, multi-syllable words ... 29
`lyricsto` ... 52
LyricText, example of overriding ... 89, 126

M

MacOS X, running LilyPond ... 11
macros ... 33
magstep ... 98
magstep function, example of using ... 98
major ... 18
manual beams ... 22
Manual beams ... 23
manual, reading ... 17
manually controlling tuplets, slurs, phrasing slurs, and beams ... 114
`markup` ... 22
markup example ... 99
markup text, allowing collisions ... 106
melisma ... 29
metronome mark, tweaking placement ... 104
MetronomeMark, example of overriding ... 111, 125
middle C ... 13
`midi` ... 38
minor ... 18
Modifying context plug-ins ... 68
modifying context properties ... 62
Modifying properties ... 112
modifying templates ... 69
moving colliding grobs ... 109
moving colliding objects ... 109
moving overlapping objects ... 109
MultiMeasureRest, example of overriding ... 113
Multiple scores in a book ... 38
multiple staves ... 25
multiple staves and lyrics ... 32
multiple vocal verses ... 53
multiple voices ... 28, 44
Multiple voices ... 48, 52
music engraving ... 2
music expression ... 24
music expression, compound ... 24, 39
Music Glossary ... 8
music typography ... 2
music, concurrent ... 44
music, simultaneous ... 44
Musical notation ... 9, 36
musical symbols ... 2

N

naming contexts ... 61
naming conventions for objects ... 81
naming conventions for properties ... 81
natural ... 18

nesting music expressions 50
nesting simultaneous constructs 50
nesting voices 50
neutral 100
`new` 25, 59
new contexts 59
`new Staff` 25
notating durations 14
notating rests 15
notation context 25
Notation manual tables 9
Notation Reference 8, 9
notation, simple 12
note collisions 51
note column 51
note durations 14
note durations in chords 27
note names 35
Note names in other languages 18, 19
note names, absolute 35
note value 23
NoteColumn, example of overriding 116, 122
NoteHead, example of overriding 82, 83, 94, 129
notes, spreading out with text 106

O

object 80
object collision within a staff 113
object properties 80
object, layout 80
objects, aligning on a baseline 113
objects, changing size of 95
objects, hiding 124
objects, invisible 124
objects, making invisible 124
objects, moving colliding 109
objects, naming conventions 81
objects, outside-staff 99
objects, positioning 114
objects, removing 124
objects, size of 95
objects, within-staff 99
octave 13
Octave checks 131
`once` 82, 87
once override 87
`oneVoice` 48
optical spacing 2
ossia 41, 95
Ossia staves 42
ossias 41
ottava bracket 104
outside-staff objects 99
outside-staff-priority property, example 106
overlapping notation 111
`override` 81
override command 81
override example 85
override syntax 81
`overrideProperty` 82
overrideProperty command 82
overriding once only 87

P

padding 109, 111
padding property 109
padding property, example 111
Painting objects white 93
parallel expressions 25
parallel expressions and relative notes 25
`partial` 23
partial measure 23
PDF file 11
phrasing 20
phrasing slur 19
Phrasing slurs 20
phrasing slurs, controlling manually 114
PhrasingSlur, example of overriding 114
piano staff 26
`PianoStaff` 26
pickup 23
pitch 13, 18
Pitch names 19
pitches 13
pitches, absolute values 35
plate engraving 2
plug-in 4
polyphony 5
polyphony 25
polyphony 28, 44
polyphony and relative note entry 46
positioning grobs 114
positioning objects 114
positions property 111
positions property, example 114, 115
properties in interfaces 88
properties of grobs 84
properties of layout objects 84
properties operating in contexts 62
properties vs. variables 174
properties, naming conventions 81
properties, object 80
property types 90

Q

quarter note 14
quote, single 13
quoting in Scheme 174

R

reading the manual 17
recursive structures 6
reference charts 9
regular rhythms 3
regular spacing 3
rehearsal marks, tweaking placement 104
`relative` 13
relative mode 13
relative mode, and accidentals 13
relative note entry and polyphony 46
relative notes and parallel expressions 25
relative notes and simultaneous music 25
`remove` 66
removing engravers 66
removing objects 124

rest ... 15
rest, spacer ... 28
`revert` ... 82, 87
revert command ... 82
reverting to a single voice ... 49
rgb colors ... 94
`rgb-color` ... 94
rhythms ... 14
right-padding property ... 109, 112
right-padding property, example ... 112
Running LilyPond ... 9
running LilyPond under MacOS X ... 11
running LilyPond under Unix ... 11
running LilyPond under Windows ... 11

S

SATB structure ... 53
SATB template ... 72
scale ... 13
Scheme ... 173
Scheme programming language ... 4
Scheme, in-line code ... 173
`score` ... 37
`score` ... 39
`Score` ... 25
score block, contents of ... 39
score formatting ... 4
Score layout ... 38
score, example of writing ... 76
scores, multiple ... 38
Script, example of overriding ... 111
self-alignment-X property ... 110
self-alignment-X property, example ... 113
`set` ... 62
setting properties within contexts ... 62
Setup ... 9
sharp ... 18
sharp, double ... 18
shift commands ... 51
`shiftOff` ... 51
`shiftOn` ... 51
`shiftOnn` ... 51
`shiftOnnn` ... 51
simple examples ... 7
simple notation ... 12
simultaneous music ... 44
simultaneous music and relative notes ... 25
Simultaneous notes ... 28
single staff polyphony ... 28
size of objects ... 95
size, changing ... 98
sizing grobs ... 108
Skipping corrected music ... 132
slur ... 19
slur ... 20
Slur example of overriding ... 86
Slur, example of overriding ... 87, 88
slur, phrasing ... 19
Slurs ... 20
slurs and articulations ... 105
slurs and outside-staff-priority ... 105
slurs crossing brackets ... 45
slurs versus ties ... 20
slurs, controlling manually ... 114
Snippet List ... 8, 9
snippets ... 9
songs ... 28
spacer rest ... 28
Spacing issues ... 9
spacing notes ... 51
spacing, regular ... 3
spanner ... 80
Specialist notation ... 9
staccato ... 20
`Staff` ... 25
staff group ... 26
staff line spacing, changing ... 98
staff, choir ... 26
staff, grand ... 26
staff, piano ... 26
staff, positioning ... 42
staff-padding property ... 109
staff-padding property, example ... 113
staff-position property ... 110
staff-position property, example ... 113, 121
staff-space property, example ... 98
StaffSymbol, example of overriding ... 94, 98
`startTextSpan` ... 104
staves, multiple ... 25
staves, temporary ... 41
stem directions and voices ... 48
stem down ... 48
stem length, changing ... 98
stem up ... 48
Stem, example of overriding ... 94, 100, 122, 124
stencil property ... 91
stencil property, example ... 91, 93, 95, 98, 112, 125
stencil property, use of ... 125
`stopTextSpan` ... 104
StringNumber, example of overriding ... 113
Structure of a score ... 41
syntax ... 6

T

template, modifying ... 69
template, SATB ... 72
template, writing your own ... 76
templates ... 17
temporary staves ... 41
tenor ... 15
terminology ... 8
Text editor support ... 11
text editors ... 11
text property, example ... 84, 112
text spanner ... 104
Text spanners ... 104
text, adding ... 22
`textLengthOff` ... 106
`textLengthOn` ... 106
TextScript, example of overriding ... 106
TextSpanner, example of overriding ... 104, 105
The tweak command ... 84
thickness ... 98
thickness property, example ... 86, 87, 88
tie ... 19, 20
Tie, example of overriding ... 121

Ties 20
ties crossing brackest 45
`time` 15
time signature 15
Time signature 16
`times` 23
TimeSignature, example of overriding 92, 93, 94, 95, 96, 97, 98
tips for constructing files 17
title 34
Top 8, 9, 10
transparency 92
transparent property 92
transparent property, example .. 84, 92, 122, 124, 125
transparent property, use of 124
transposition 18
treble 15
triplet 23
triplet bracket 84
triplets 23
triplets, nested 84
Tunable context properties 63, 66
tuplet beams, controlling manually 114
tuplet bracket 84
tuplet-number function, example 84
`TupletBracket` 84
TupletNumber, example of overriding 84
tuplets 23
Tuplets 23
tuplets, nested 84
`tweak` 83
tweak command 83
tweaking bar number placement 104
tweaking dynamics placement 107
tweaking methods 81
tweaking metronome mark placement 104
tweaking rehearsal mark placement 104
tweaks, using variables for 126
tying notes across voices 124
typography 4
typography, music 2

U

underscore 29
Unix, running LilyPond 11
`unset` 62
up 100
Upbeats 23
updating files 34
Updating files with convert-ly 34, 138
upgrades 34
using variables 33
using variables for tweaks 126

V

variables 33, 38, 133
variables vs. properties 174
variables, characters allowed in 33
variables, defining 33
variables, using 33
variables, using for tweaks 126
verse and refrain 54
verses, multiple vocal 53
`version` 34
version number 34
versioning 34
viewing music 11
Vocal music 32, 33, 58
vocal score structure 52
vocal scores with multiple verses 53
`Voice` 25
Voice context 44
voice contexts, creating 48
`voiceFour` 48
`voiceOne` 48
voices and stem directions 48
voices crossing brackets 45
voices vs. chords 44
voices, more on one staff 28
voices, multiple 44
voices, naming 45
voices, nesting 50
voices, reverting to single 49
voices, temporary 50
`voiceThree` 48
`voiceTwo` 48

W

whitespace insensitive 16
whole note 14
Windows, running LilyPond 11
`with` 65
within-staff objects 99
words with multiple syllables in lyrics 29
writing a score, example 76
Writing pitches 16
Writing rests 16
Writing rhythms 16
Writing text 22

X

X11 colors 93
`x11-color` 93
x11-color function, example of using 129
x11-color, example of using 94