

The Ticklish Guide To LilyPond

by Joshua Nichols

Contents

The Ticklish Guide	1
Introduction	1
The Alternative	3
Why Another Guide?	4
What is LilyPond?	4
What to Expect	5
Let's Begin!	5
Syntax	5
Skeletal Structures	5
The Language of Music	6
Plausible Structure	7
Other Assumptions	8
Finding Agreement	8
Putting Muscle on Skeleton	8
The \score Block	9
Rules of Thumb	10
Other Blocks Essential to the File	10
A Brief Philosophizing	11
The \relative Command	11
(More) Advanced Notation	13
Keys	13
Clef & Clef Changes	14
Time Signatures	14
Tempo Markings	14
Rhythm, Rests, and the Like	15
Rhythm	15
Rests	16
Accidentals	16
The Augmentation Dot	17
Articulations	18
Expression	19
Slurs and Ties	19
Hairpins	20
Spanners	20
Dynamics	20

Miscellaneous Notation	20
Beaming	21
Cadenza Notation	21
Bars (Barlines and Bar Checks)	21
Fingerings	22
Markups	22
Tuplets	23
Chords	23
Partial Measures	24
Bar Numbers	24
Conclusion	24
Post Script	25
 Appendix	 27
Different Views of Content Authoring	27
Main Examples	27
Basic Examples	27
Advanced Examples	29
Answers To Main Examples	29
Basic Examples	30
Advanced Examples	30
Real Life Example	31

Abstract

In the world of computers and programs related to creating printed materials, the way that most of the world operates is in a program (or paradigm) that is identified as a w^ysⁱw^yg.¹ This kind of program operates like this: you enter the information you wish to have displayed on the page into the computer, and then before printing it, you format the page (via a computer screen) to reflect what you want to have exactly. What you see on the screen is what you will get on the printed page. Though most programs on a computer deal almost exclusively this way, this is by no means the only way you can work in a computer. The alternative, the w^ysⁱw^ym,² addresses things completely differently.

Instead of “what you see is what you get,” you start by creating a file called an input file or source file (usually plain text) that contains commands and information that do not appear on the printed page. What actually happens next is an automatic compiling engine interprets that file for you and produces a new file that is the file you print. This secondary file depicts what you really *mean* to see on the printed page. That is why they call it “what you see is what you mean,” because you actually don’t see the finished product in the input file, but in a secondary file with everything formatted the way you intend it to be formatted. w^ysⁱw^ygs actually have a source file that you never see; you communicate to it and essentially “write” it by using the GUI.³ LilyPond is a w^ysⁱw^ym program that produces high-quality typeset music. It is also an open-source program rather than a proprietary program.⁴

The Ticklish Guide

Introduction

LilyPond is an incredible and immensely powerful tool for composers and engravers. It originally began as a joint venture between two musicians who honestly felt mainline scoring applications were lowering the standards of music notation. They believed in the fundamental principles of readability and notation done right the first time.

Unfortunately, in an age where the proliferation of w^ysⁱw^yg editors and instant-gratification computing are widespread and the normative standard of doing things, LilyPond has a slight disadvantage. The basic syntax of this typesetting engine is plain text.

This is one of the first complaints by those who are switching to LilyPond – and rightly so! The click-and-drag feature of countless computing programs has made the computer accessible to the regular user. Companies have been built upon the idea that you can get instant results.⁵ This raises some important reasons for this type of program. The first is that a program that uses a w^ysⁱw^yg GUI gives you instant feedback, whereby you can make any number of minute changes in the layout, coloring, or sensibilities of the document or file. The other main reason that people cite in favor of this approach is the ability to check for errors on-the-fly, whereby the user is constantly able to preempt an error in the file.

To be truthful, w^ysⁱw^yg is the only viable option for graphics-related projects (such as posters or illustration) because those things are by definition required to be adjusted at a moment by moment basis. Unfortunately, there are many reasons why w^ysⁱw^yg editors are inefficient and time consuming. To list a few:

1. When it comes to workflow, a w^ysⁱw^yg actually distracts the user from being creative or focused. This is because everything the user does is being minutely changed on a moment-by-moment basis

¹“What You See Is What You Get”

²“What You See Is What You Mean”

³The “Graphical User Interface” is the menus, icons, and “point-and-click” features of a program.

⁴Proprietary programs are closed-source, meaning that you cannot modify or manipulate the program’s coding. This is necessary for for-profit businesses who make money off of their programs; this allows them to maintain secrecy and to attempt to have control over the market. Open source is a paradigm which releases its code to be free and open to the public for editing and distribution, and it is inherently non-profit.

⁵Say, for example, Adobe’s infamous Creative Suite, Microsoft’s Office Suite, or MakeMusic’s Finale notation software

within the program, and it is always calling the attention of the user to respond (even when he or she is not looking to do so). This stunts workflow and ultimately makes the creative process more difficult to achieve.⁶

2. Proprietary software (that is, license-based user software) does not allow you to modify the source file or the program itself; it only allows you to *use* both. You don't actually purchase the program, you merely pay a right or license to use it. This means that, if you happen to think of a feature that you would like to have, then you don't have any viable means of getting it into the program without violating software agreements or spending a lot of money to pay for the right to modify the program.
3. While proprietary software usually includes same-version enhancements and bug fixes,⁷ the user is usually constrained to paying new licensing fees for major version changes.⁸
4. In the case of proprietary software, if the company goes out of business, or takes a major shift in priorities, you are still bound by the whim of the company. This means that, practically, you are lost in the dark if and when a company "goes under."
5. WYSIWYG almost always utilizes proprietary data formats in the source file. What this basically means is that the file itself is protected by a language which only the program can understand, and it takes quite some time to decode/debug these files so that other programs can read them. This practically relates to the real world: if I wish to send a document format that can only be read by, let's say, a program called Microsoft Office Desk, and if the person I am sending it to does not have that program or a highly proprietary means of decoding that source file, I am out of luck.
6. Proprietary data formats are also more easily corruptible and more difficult to recover in the event of a computer crash. This means that if for any reason your computer crashes⁹ or the software GUI fails for whatever reason, then unless the auto-save features (if any) are working in backing up files at the crucial moment where that "bit of data you need is so important," then it renders the file useless, and you have to recover ground. This is quite annoying for a regular text document, but imagine losing all of that information in a Finale or Sibelius file, where you might have made some serious changes to the music that, unfortunately, cannot be recovered.
7. Proprietary data formats are also strenuous on file sizes. A several-hundred-page score of a work with dense notation could absorb as much as several-hundred megabytes. This is also impractical because most emailing systems only accept an average of 8–12 megabyte file sizes.
8. In the case of workflow, version control¹⁰ is nearly impossible, since the details of a score or file are hidden within the proprietary formatting.
9. Proprietary data formats also make it nearly impossible to maintain source transparency. What is meant by this? This means that it's hard to retrieve statistical data and meta-data about the file

⁶For instance, in Finale, the software automatically fills in the remaining unused beats of a measure for you with rests. This hinders composition because your mind has been "called" to focus on the remaining beats of the measure, rather than focusing on the composition itself. This perhaps is more apparent in word processors like Microsoft Word which deal directly with formatting as you type. What you see is your formatting, which is always calling your attention in the moment of said formatting. If you are typing a paragraph, for instance, and you notice that your word spacing is "a little off" (because the program's default behavior shows you what it would look like on print), then you become distracted from actually writing, which is the intention of using the program. You can see this effect easily: try typing in a text editor (such as WordPad on Windows) and type like that for a couple of weeks without using your favorite WYSIWYG word processing program. You will surely find productivity and content increase.

⁷A bug is anything from a major to a minor glitch that affects the usability and stability of the program. Every program goes through many changes to keep bugs to a minimum.

⁸For instance, if you had Sibelius 6, you would be entitled to all the bug fixes and enhancements for that version, but you would have to buy an upgraded license in order to use Sibelius 7.

⁹Though, it is far less likely in today's computer, but still a problem, nonetheless.

¹⁰...or managing on a transparent level the changes a file goes through...

in any real and understandable way.¹¹ The formatting and scoring of the program's proprietary data format make it nearly impossible to extract critical data of and pertaining to the "stuff" of the file; this is most easily seen in viewing native .doc and .docx files in an open-source word processor. Generally speaking, the formatting and page styles are skewed in some way, making it difficult to understand what the intentions of the author were. This also applies to editing across several versions of the same program. Microsoft 2007 does not readily and easily transcribe the file formats of a Microsoft 97 document file.

10. Finally, wysiwyg applications are rarely fully featured. This is because the proprietors have to make decisions about what to include and exclude from the menus and shortcuts of the GUI in order to make it learnable for the most amount of potential clientele. Programs that are fully featured are rarely easy to learn, and the systems are usually archaic and thus only beneficial to those who have been part of the software experience for a long time.

Most people implicitly accept these shortcomings (with exception to file risks and proprietary risks), and walk a tight line between being "fed up" and "satisfied" with their user experience. It could also generally apply that 80% of the users use only 20% of the features, and vice versa.¹²

The Alternative

When it comes to alternatives, there is the wysiwyM system of formatting. This simplifies workflow and allows the user to make adjustments only when the end product doesn't meet their expectations. wysiwyM also adopts the practice of handling the formatting, layout, and stylings of your file (which can be manipulated). Products that offer a plethora of formatting options do not help the user to make professional and informed decisions. wysiwyM offers a user experience that simplifies workflow and gives the user the opportunity to breathe their creativity, while correctly formatting their content behind the scenes to adopted standards within the scope of the program.

In addition, there is the plain text format. This method of content storage stores all the information of a file in a plain text file (which can be edited by any text editor), and it is translated by the compiler or program in charge of transcribing the data. There are many advantages to organizing data this way, a few being:

1. Plain text files are more easily recovered in the event of a crash. Because the file is not dependent on proprietary formatting, you only lose what you didn't save, or if it is somehow corrupted, you can at least extract information leading up to the time of the crash.
2. File sizes are significantly smaller. A proprietary file format is anywhere between 10-1000 times larger than a plain text file of the same information.
3. One hundred percent of the input file is meta-data.¹³ Styling of the information is transparent in the input file. Even if someone does not have the compiler to produce the finished product of the

¹¹Proprietary data formats are basically encrypted files, files that use a secret language to communicate with the native GUI. This is another reason why it is difficult to understand proprietary formats with programs that cannot natively understand it.

¹²Truly, this can be a touchy subject. Many people who talk about switching to another program often lament because, frankly, they don't believe that they have the time to learn it correctly. They feel as though they've done as much as they can with their current program and have gained a thorough understanding of it. Frankly, however, this is a bit of a farce, considering most people don't actually have a functioning knowledge of programs like Finale and Sibelius; they *think* that they have a functioning knowledge of the program. This is because the menus and options give the illusion that you have that knowledge, that you are in control. The truth, in reality, is that we don't know the program as much as we think. This has been discussed at length in internet forums and professional papers when speaking about the differences between Microsoft Word and L^AT_EX and other related typesetting engines (T_EX, XeL_AT_EX, and L_X).

¹³Meta-data is the part of an input file that corresponds directly to something that shows up on the printed page. For example, if you look at the musical note "middle-c" on a printed page, that note corresponds directly to a place in the input file where "middle-c" appears as a command. In LilyPond, everything in the input file is meta-data. However, there is a sort of fluff that accompanies most proprietary input files. It is information that is neither necessary nor directly important to the printed page; it is not meta-data itself but exists in addition to the meta-data. It is usually garbled and jumbled data bits (not plain text) which can only be read and

file you sent them, they can still read and “parse” the information on a meta-level. Also, the constant need to create the desired format and style is significantly reduced, if not totally eradicated.

4. Plain text allows the corruptible elements found patently in WYSIWYGs to be nearly absolved, the most critical being workflow. When one is not consumed by the constant nagging of a WYSIWYG’s constant updating of style and format, the user can focus on data production, which leads to greater and more responsive creativity.

Why Another Guide?

LilyPond utilizes the plain text workflow and WYSIWYM approach to input. Though the discussion of why one should switch to plain text workflow and WYSIWYM editors should be clear, the challenge of switching (especially if one is a hardcore WYSIWYG user) is still daunting and fuzzy.

LilyPond has some of the most incredible and comprehensive documentation I have ever seen when it comes to open-source programs. Indeed, one should not see this “even gentler” introduction as a replacement for the documentation; on the contrary, this should be a “baby language tutorial” for those who see a document like the *Learning Manual* on LilyPond’s website as a little too much too quickly.

This guide should also serve as a “fundamentals” interaction. The documentation, though comprehensive and far better suited for learning, does miss the mark on explaining some of the reasons why certain things are done.

Nevertheless, if you have found your way to this document, then you are probably searching for a way to transition without overloading your brain. My intentions (as a former hardcore WYSIWYG user) are for you to get what I didn’t get several years ago.

What is LilyPond?

Like I have said before in the Abstract, LilyPond does not have a typical GUI, and it is unlike any other program you may be familiar with. Microsoft Word formats the text on the screen to reflect the printed page exactly. LilyPond is the exact opposite: you write plain text information into a file that the program understands, and instead of what you see on the screen being what prints out, the compiler converts that information in the plain text file into a new file that depicts exactly what will print. LilyPond is an engine which translates bytes of data into formatted music. The output of the music is to PDF (via PostScript) which is the universally recognized file format for the transferring and archiving of documents and other things. Most all computers can read and print PDF files. The software for viewing PDFs is readily available (most people already have it on their computer) and takes up an extremely small amount of computer hard drive space.

In addition, LilyPond, though not a proprietary formatting system, does utilize “jargon” and various “scripts” to simplify the output process. The compiler itself contains millions of lines of code, all corresponding to processes that, like a cell in the human body, only respond when called for by the source file. This simplifies the workload on the computer and allows the engine to utilize minimal RAM and computing space. This also makes the program lightweight, with the actual bulk of the program being contained in files which correspond to fonts and graphical drawing systems. The rest of the code are the pathways which initialize when called upon by the source file through the compiler.

This is important for the understanding of the program because there are certain boundaries around what one can and cannot do with an input file. One of the things that the input file must contain for LilyPond is a version statement. The version statement is required because it helps the compiler to realize what form of the input language you are using.

The input language is also very precise. There are, however, some helpful tips about the program:

interpreted by the particular program to which that the file corresponds. (It usually contains information such as bookmarks in the file, etc.) See “Different Views of Content Authoring” in the Appendix.

1. It is not a “space-sensitive” file format. It does not require that a certain *amount* of spaces be present in the syntax in order to produce the same output as another example.¹⁴
2. Errors are generated in a terminal or separate compiling screen which indicate where a fatal error has occurred within the file, or where cautionary messages advise on the status of the compiler.
3. Syntax should be seen as intuitive

The syntax *at first* will seem... cerebral. This is to be expected.

What to Expect

The rest of this guide will teach you how to write your music in the form of the input language. As you begin to use LilyPond, Expect:

1. ...the syntax to rewire your brain. We have trained our brains to understand menus, options, and instant gratification. This is completely inefficient and wasteful of time in the long run. However, when you come to understand what you really need, and understand how to communicate effectively, your brain begins to transform into a fiend of a processing machine.
2. ...to be frustrated from time to time. This is especially true in the beginning. Patience when learning something new is always important.
3. ...to feel empty and unsatisfied while writing input code at first and from time to time. Being worshippers of instant gratification, we feel the compulsive need to see results as we are inputting information. This will shift as time moves along.
4. ...to feel an enormous sense of accomplishment and completion when you finish and compile your scores. You will feel this every time you create a new score!

Let's Begin!

Syntax

LilyPond is organized as a syntactical language. This means that it has a specific way of identifying common functions and musical text. Though the language is not space-sensitive, it is case-sensitive. This is incredibly important: the most common mistakes made by beginners (in my opinion) are made at the basic syntactical levels of the program, especially ones pertaining to lower-case, upper-case, and the location of musical expressions.

Skeletal Structures

LilyPond has a frame, a skeletal structure, by which the basic requirements of the syntax are addressed. Here are those structures:

1. The backslash (\) - The backslash introduces a linguistic function of the syntax directly into the program's compiler. Think of it as a signal to the compiler that what comes after it contains musical information and/or data meant to be recognized in real time.
2. The percent sign (%) - The percent sign signals a comment, or a thing that is ignored in real time while the compiler is parsing the information. A single percent sign is line-sensitive. This means that wherever the percent sign resides, everything to the right of it on the the same line is ignored. The percent sign can also be used with curly braces which can “comment out” an entire block of text contained within the braces (e.g. `%{ blah %}`). This is explained in more detail later.

¹⁴For instance, the compiler does not see a difference between 1 space and any larger number of spaces.

3. The curly brace ({}) - The curly brace encloses musical expressions or functions. This is perhaps the single most important skeletal structure to understanding LilyPond. Whenever using curly braces, you must surround them in spaces (this will become obvious in further sections).
4. The equal sign (=) - The equal sign does not initialize any particular pathway in the compiler, but simply is used as a “definer” or “signaler” of sorts to bring a definition to any particular variable (e.g. `composer = "J. S. Bach"`).
5. The double quote (") - This usually defines things that are to be quoted directly as written into the score. It also is used as a container for scripts and other such things (to be explained more later). This container should not be confused with two single quote marks (' ').¹⁵

There are other fundamental structures related to music expression which will be addressed later. Most importantly, there are no substitutes for the above structures; they are input-sensitive. For example, a backslash cannot be replaced by a forward slash (/). Curly braces cannot be replaced by square brackets or parenthesis ([], ()). Moreover, if one forgets to complete these structures or give them proper meaning (definition), the compiler will warn you of the error you are making.

The Language of Music

LilyPond has developed a unique but intuitive way of communicating basic musical expressions. In addition, LilyPond has made many things easy:

1. The program has “presumptions” or a set of assumptions it makes when certain information is not provided in the source file. This allows music to be quickly input and economically handled.
2. The program is a WYSIWYM, which means that instead of you having to worry about the layout all the time, you only have to worry about the layout when you are looking at a draft product. You do not have to worry about how the music is layed out and thus can focus on the input of notes.
3. The program makes many intuitive decisions about the size and characteristics (or house styles) of the score, which means that 90% of the time the compiler gets the size of the score “correct.”

All of these things allow you as the user to focus on the typing of information, only needing to worry about how it looks once everything is said and done.

So, what does LilyPond recognize as musical information? The default language of LilyPond recognizes all of the musical alphabet as plain English:

a b c d e f g

Not so difficult is it? In the case of flats and sharps, however, it can be tricky. The default syntactical language of LilyPond is Dutch, which means:

flat = `es`; sharp = `is`;

This can be changed by default, however. If you wish to use an English spelling, changing the language to English allows you to enter flats as “f” and sharps as “s”. Double-sharps and double-flats are repeated syntactically in a row (e.g. e-double-flat is “`eff`” and f-double-sharp can be “`fss`” or “`fx`”).¹⁶

It also recognizes rhythms by their common mathematical break-down:

¹⁵Do note that the double quote is not the same as stylized curly quotes. Most text editors do not create stylized double quotes, but many word processors do. Be careful that if you edit a file in a program like LibreOffice, Microsoft Office, or OpenOffice to turn off the curly quote feature. If you use a program like Frescobaldi or LilyPond’s native text editor, this should not be a problem.

¹⁶To change the input file to reflect English spellings, insert `\language "english"` if you need to change the language of the input to english, insert:

`\language "english"`

into the file right before or right after the version number (which will be explained shortly).

1 2 4 8 16 32 64 128

Not so difficult, eh?

In what order do we express characteristics of notes? The order of syntax is as follows:

1. Pitch
2. Accidental
3. Rhythm
4. Expression (addressed shortly)

So, a quarter-note e-flat is expressed as: `ees4`. Make sense? Just remember, **PARE** your information down.

There are also other intuitive structures LilyPond understands. Musical expressions can be created by remembering common “textual equivalents,” such as the following:

< - crescendo; > - decrescendo; . - staccato; - -tenuto; ^ - “house-top accent”

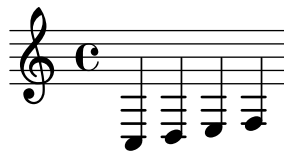
Slurs are written as parenthesis (()) and beams are expressed by square brackets ([]).

Plausible Structure

How do all of the idiosyncrasies combine to make musical expressions? Simple: you express the musical thoughts like sentences. You write expressions separated by spaces, like with normal words. Here’s an example:¹⁷

`c4 d4 e4 f4`

would produce:



Notice how the compiler made some assumptions:

1. The default clef is the treble clef
2. The default time signature is common-time.
3. Barlines are automatically placed when the default time-signature is satisfied within the measure.
4. The compiler only places as many bars of music necessary to complete the musical information given in the source file.

The compiler makes these assumptions unless you specify otherwise, and thus it significantly reduces the amount of specific information you have to put into a source file. This comes in handy when learning LilyPond.

¹⁷Any examples from here on out will be related to the concept being demonstrated. The examples may contain other elements, but in any case the focal point of the code is on the concept being demonstrated. It does not necessarily mean that the code is actually outputting the other elements of the example.

Other Assumptions

The compiler also makes some keen assumptions about the information that you input. One of them is rhythm. As in the example above, though one can specify the specific rhythmic value of each note (which could become cumbersome in extended scores), it really isn't necessary. The compiler won't change the rhythmic value until you tell it to. One can define the notes as:

```
c4 d e f
```

...and still come out with the same output as before. If it is at the very beginning of the score, one could even define the phrase as:

```
c d e f
```

...and even still be able to get the same output. This is because the default rhythmic value is a quarter note until you tell it otherwise.

Finding Agreement

I'm sure lots of people find some of the above assumptions and syntactical jargon as strange or even backwards. But, to be honest, these assumptions and intuitions are more intuitive than popular WYSIWYGs of the day.

It can be argued that we read music pitch-level first then rhythm, even if they seem simultaneous. It could also be argued that, much like simple-entry modes in common WYSIWYGs, rhythm ought not be explicitly expressed unless a change occurs. This simple line of assumptions can be your greatest ally...or enemy. Either way, the process of inputting code may feel slow, but because of the powerful compiler and engraving engines within the program, your brain will actually have done far less work than with any WYSIWYG.

How about some fun? Let's create some examples to explore the syntax.

Putting Muscle on Skeleton

Every musical expression, or muscle, must be connected by some kind of tendon; in this case, it is the curly brace (`{}`). First, start by typing in the version statement:

```
\version "2.18.2" %Always check to make sure your version statement matches  
the version that you have installed on your computer.%}
```

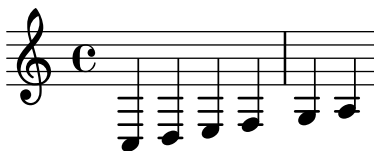
And then some curly braces:

```
{ }
```

And now let's begin the musical expression. Try this:

```
{ c d e f g a } %those are the curly braces from directly above%}
```

Save your file, and then compile it.¹⁸ What should result is an output file that looks something like this:



¹⁸If you are in Windows, click-and-drag the saved file into the shortcut icon for LilyPond. In Mac, just click the menu button and click "compile".

Notice how there is no barline at the end of the partial measure, nor a place where the program “filled-in” the space with rests because it doesn’t equal a measure; the compiler just did what you told it to do. If you wanted a barline at the end of that measure, you could do that. Theoretically, that is no problem at all.

If you haven’t begun to see it yet, don’t worry; it will make sense in a short while. What is beginning to happen, however, is your brain is rewiring to think of things pragmatically, rather than programatically. Most wysiwygs deal with input programatically: you have to manipulate the program’s default output to achieve the goal. This is very practical for very simple input. However, once you go beyond this it becomes a matter of tracing your steps through the menus to get to the desired outcome. For some processes, it’s simple. For others, however, wysiwyg is very cumbersome. However, in a pragmatic sense of input, you need only be familiar enough with some of the basic linguistic characteristics of the input code in order to accomplish what you need. Let’s do some more input, shall we?

The \score Block

One of the main assumptions that the above examples make is that of the \score block. When you use curly braces by themselves, you are actually telling the compiler, “Look, I have a simple score, and the scheme of this score is all default.” This is great for getting your feet off the ground, but you will definitely be wanting to do more complicated tasks than that. So, let’s start introducing the \score block:

```
\version "2.18.2" %{\always remember to check your version number%}  
\score { c d e f g a }
```

Save and compile. What do you get? Probably something like this:

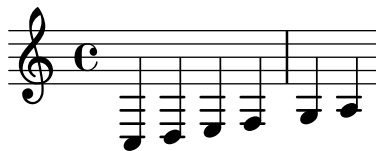
```
error: syntax error, unexpected NOTENAME_PITCH...
```

...with “Exited with return code 1.” as the final message. Why did we get an error here? We got an error because we have not properly defined our \score block. This brings up a significant point: when you start to use functions like \score, you must then start explicitly defining your parameters within these functions. The typesetting engine will cease to make assumptions once you begin to define your own variances and contexts.

Now that the compiler knows you are beginning to define your own variables, you must now tell it a crucial piece of information: what *exactly* the information is for. What is missing from the above example? How about a staff:

```
\version "2.18.2" %{\always remember to check your version number%}19  
\score { \new Staff { c d e f g a } }
```

And this produces:



Ah, we are back in business! Now we have a score! You may be wondering, “What did I just do?” Simple: you invoked a \score block, which contains single musical expressions, and defined that you wanted a staff by telling it to create a “new” (\new) one! It’s that simple!

Here’s something really important: try not to panic if you get an error message. When I first did, I panicked, and I wrongfully didn’t work through it. When you are calm and trying to parse what you are seeing, usually the answer is clear right from the get go.

¹⁹From this point on, the *Ticklish Guide* assumes that you have entered the appropriate version number into your file.

Rules of Thumb

We need to establish some things. The first thing to talk about is the open curly braces. When it comes to open braces, if they are opened, they need to be closed.²⁰ It is really better to habitually draw open ({) and closed (}) braces *at the same time* and then enter your text between them. This habit will prevent the most frustrating of errors which most amateurs tend to make when first using LilyPond.

Other Blocks Essential to the File

We want to produce more than just musical information, though! We would like to be able to produce titles and composer markings to help with the identification of the music. For this, we will use a new command:

```
\header{...}
```

Isn't it intuitive? Any title and related information can be found here. You can define practically any element of the page. Here are some basic variables we can use:

```
%within the header block...
dedication = ""
title = ""
subtitle = ""
composer = ""
poet = ""
arranger = ""
```

Just type the information you want between the sets of double quotes. If these ever appear in the `\header{}`, they will appear in predefined places according to common notational standards. Here's an example:

	dedication	
	title	
	subtitle	
poet		composer
		arranger

This can be modified, as described in the section of this guide entitled "Markups." Of course, there are plenty of other variables, but as a "ticklish" introduction, this should suffice in order to go on further. Another block that is essential is the paper block:

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

In this block, we can define all of the arbitrary variables related to the styling of the paper and its contents. Wondering what those variables look like?

```
%within the paper block...
#(set-default-paper-size "letter")21
right-margin = 1\in
left-margin = 1\in
bottom-margin = 1\in
top-margin = 1\in
```

²⁰Another useful way to think of braces and other related shapes are as "containers." Containers need to open and close. For example, you never want to leave a container of cookies open.

²¹Notice the double quotes as a container.

There are numerous advantages to this block, and those will become abundantly clear later on. The most glaring advantage is the ability to define a paper size, as well as the margins of the printable page! In Europe and basically the rest of the world except for the United States, the default paper size of printers is the “a4” size, and so if you wish to change this, you will have to explicitly change it using the script `#{set-default-paper-size ...}`.

Most everything you can see so far is intuitive – and that’s the point. There are a lot of things that will seem backwards at first, but 75-90% of all the syntax of LilyPond has a common thread to a realistic view of its command.

The last block worth mentioning is the `\layout{}` block. It can be put in any part of the file, but it is most commonly found nested within the `\score{}` block.²²

A Brief Philosophizing

Many have misunderstood LilyPond to be a language which is a lot like programming language. Languages like C++ and Python come to mind to those who see a program’s syntax like this. However, any closer detail will tell you that it is far from it.

The structures are extremely lean and simplified²³ and have mostly been made rid of programming idiosyncrasies. For example, the program doesn’t require you to create explicit formal arguments in order to call certain features of the program. In fact, LilyPond is several million lines of code that have been transformed into pathways that are “turned on” when one uses LilyPond’s simplified syntax. So, what the user actually has to learn is far from coding language; in order to function efficiently with C++ or Python one must be intimate with *most* of the syntax. With LilyPond, one must only be familiar with the syntactical equivalencies and basic rules in order to use it rather proficiently.²⁴

The `\relative` Command

Think of the last WYSIWYG you used. How did you input the music? If you are in a quick hurry, or easily frustrated by pointer accuracy of the click-and-drag, you used some form of simple-entry or quick-entry. How did you switch octaves? Did you use a `shift+↑` (or `shift+↓`) shortcut? How did you know when to change octaves? Chances are you knew because your program, by default, actuates the octave closest to the previous note you entered. In this case, the octave is determined by the relative note that precedes the current note. This is common, so that you do not have weird octaves popping up in simple entry. The distance of that interval is usually whatever is less than or equal to a 4th. If it is above a 4th (any kind of 4th, mind you) then it takes the shortcut command `shift+↑` (or `shift+↓`) to move to a different octave than that of the note preceding it.

LilyPond has the same function; instead of writing in absolute pitch (which is useful for some things), you can write in `\relative` pitch, which bases the current note relative to the previous note. In order to use this command, you must initiate a starting pitch of some kind, and it can be anything:

```
\relative c' {...music...}
```

In the above case, the notes contained within the curly braces are relative to the absolute pitch of `c'`, which is middle c. So, let’s try it out:²⁵

```
\relative c' { %Line breaking and indentation is useful here because it
keeps the score clean and easy to follow. Instead of having all of your
music on one line, you can displace it 1-2 measures per line, or however
you prefer.%}
```

²²The `\layout` block can define indents for specific scores, define lyric sizes, and a whole host of other things, such as score layout and even staff-to-staff spacing.

²³Especially since version 2.18.x

²⁴Of course, to fine-tune the score, one must be more familiar with the details of the syntax and the internal language, but at that point you are becoming your own expert.

²⁵Please note that it is a single quote after the c, not a double quote.

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

This would generate the following output:



Notice how each note follows the next in a scale, and that is because each note was less than a 4th away from the previous note! What if we did this without `\relative` command? `{c d e f g a b c}` would produce:



Notice how it went back to some different notes. This is because you were writing in absolute pitch, which is default in LilyPond.²⁶ How then, does one change octaves? Simple! Use a “'” to raise an octave (relative to the pitch before) or a “,” to lower an octave (relative to the pitch before). Let’s try a broken chord:

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

Produces the following output:



But, if you take out the single quote (or tick mark), you get the following output:



Very different, no? This is perhaps one of the more frequent mistakes of new LilyPond users: forgetting to change the octave when it needed to be.²⁷ Try inputting the following examples:



If you get the output right the first time, congratulations! If not, don’t panic! Retrace your steps, and see where you went wrong. If you aren’t quite sure what to use, experiment, and see what the output gives you. Remember, LilyPond is a logical typesetting system, and it bases its output on the instructions and logical output of what you write! So if there is something wrong in your file, then it has to do with what you wrote.

²⁶This format of writing pitches is extremely useful in experimental and post-modern notation. This is very useful if one does not want to have any doubt that they have written the proper pitch. With tonal music and most other music, however, `\relative` is a suitable way of describing pitches.

²⁷One of the first projects for which I used LilyPond was transcribing a flute solo for an audition at a major university. I had typed everything in, compiled, and my flute line ended up four octaves lower than it should have been! I found that I had forgotten an octave mark at one point, and once I found it, I corrected the problem, and the output fixed itself!

(More) Advanced Notation

Now that we have some of the groundwork established, let's consider getting on to more practical, real life situations.

The rest of the *Ticklish Guide* will show you the basics of all the aspects you would need to create a simple score; with all the explanations and such, it should be a rather smooth transition into the *Learning Manual* on the LilyPond website. The developers of the program even recommend experienced users covering the *Learning Manual* once in a while because there are many things that are not covered in as great of detail in other manuals as in the *Learning Manual*.²⁸ Furthermore, the *Learning Manual* is far more in-depth and explanatory in its nature.

If at any point something doesn't make sense, reread it and try it again, or go to the corresponding *Learning Manual* chapter that covers it. Most of the time, however, your mistakes will be self-inflicted (meaning it's not something the manuals or I did wrong), so exercising patience is the absolute key to maintaining composure during the early stages.

It is also important to note that there are many ways of getting help. One of them is the LilyPond mailing list. It's an email update sent several times a day with questions and responses to questions that may not be covered in the manuals.²⁹

The initial stage of learning LilyPond should be self-guided. However, I would *highly* recommend downloading and using the "gentle-giant" Frescobaldi³⁰ program. It is perhaps the greatest front-end program for interacting with LilyPond. It is extremely light-weight but heavy duty (like a carbon-fiber sports car) and it contains auto-completes which, when used correctly, speed up the process of syntax.

Following this guide will be an appendix of graduated examples with the answers of those examples, for your practice. None of the examples in the appendix will go beyond the scope of this guide. It will also contain various problems of notation for you to solve, to get comfortable and familiar with how errors contribute to problem solving in scores. With patience, practice, and a calm mind, you can become very proficient with the basic functionality of LilyPond.

Keys

The concept of keys in LilyPond is rather straight forward.³¹ You must first invoke `\key` which is then followed by two arguments: a key center and a key mode. You can use any key center you can think of, and there are a host of modes. Of course, the most common are major and minor. Here's what a key looks like:

```
\key ees \minor % { invoking minor or major requires a backslash % }
```

And it produces the following output:



²⁸Once you have finished this guide, it would be safe to drop in the 2.18.2 *Learning Manual* starting at section 2.2.1. There will be some overlap at that point, but it will be minimal.

²⁹When it comes to the email mailing list, it is easy enough to setup your email client to automatically archive these emails to a folder dedicated to LilyPond. Then, when you have questions about using LilyPond, you can search these emails with keywords about what you might need, and you can also send questions to the list itself.

³⁰It can be found and downloaded from www.frescobaldi.org/

³¹You will still have to spell out every note (for example, if you are in the key of "ees" you still have to write "ees/aes/bes" every time you use those notes). This is a so-called "problem" with wsiwYG users trying to switch over to LilyPond, citing the common complaint, "I only should have to type the note name and the key will take care of it for me." This thinking, in reality, is slightly fallacious, because it does not actually make "sense." The note "e-flat" in the key of e-flat is still...get ready for it...e-flat. That's because a key is not a mystical device used to define arbitrary notes, but a consistent reminder of a set of accidentals to be applied throughout a work or piece of music.

The *Notation Reference*, a guide for the advanced use of LilyPond, describes custom key signatures. This is very useful for obscure or 20-21st century music.

Clef & Clef Changes

LilyPond assumes that you will be using a treble clef. If you want to start with a different clef, or if you want to change clefs in the middle of the music, you invoke the command `\clef` and the name of the clef you wish to use. When it comes to clef changes, LilyPond can place them wherever you want inside of a measure.

LilyPond by default supports many different clefs, the most common being the treble, bass, and alto clefs:



So for this example, `\clef bass`, `\clef alto`, and `\clef treble` were used.

Time Signatures

LilyPond handles time signatures with a backslash, but there is only a small twist: the actual time signature is described using a forward slash (/). To enter the time signature $\frac{3}{4}$, you must type:

```
\time 3/4
```

And the compilation will give you:



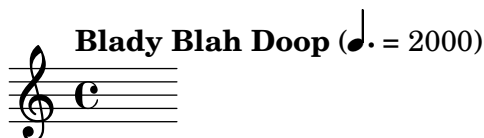
Everything following that change is affected by the command. This goes for most other things in an input file.

Tempo Markings

There are two very flexible ways of dealing with tempo markings, which always occur starting on the beat following this command. To create a new marking, you need to invoke `\tempo` and then one of two things: a text container and/or a note value definer. They happen to look like this:

```
\tempo "Blady Blah Doop" 4. = 200032
```

Which would output:



³²The dot after the "4" represents an augmentation dot. That will be explained in the rhythm portion below.

Fascinating isn't it? On wysiwygs you would have to create new tempo markings for each time you wanted to use something other than a traditional Italian term, but that is no fun, right? In LilyPond you can also just use one or the other feature. For example, if you wanted to just use a text tempo marking, preformatted the way it was supposed to be, you could use `\tempo "Insert Tempo Here"`. If you just wanted a tempo marking indicating a metronome marking, you only need to use `\tempo 2 %{\whatever beat you want%} = 100 %{\or whatever metronome marking%}`. The number on the left side of the equals sign corresponds with the beat emphasis mentioned above. The number on the right can be set to any amount you can think.

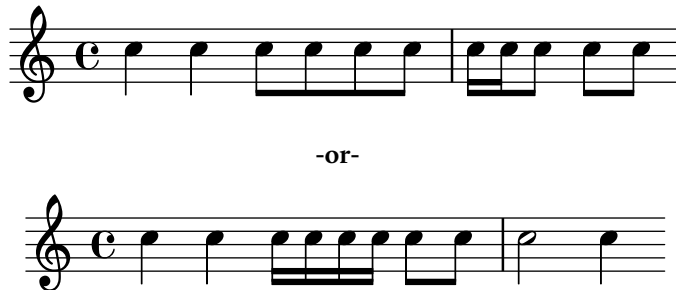
Rhythm, Rests, and the Like

Rhythm

Though explained briefly earlier, this section will describe in detail the manner in which one should think of rhythm. In common wysiwygs, rhythm is expressed by either clicking a value and dragging it across the screen into the appropriate place on the staff or it is done by short-hand (using a keypad and choosing an arbitrary number which corresponds with the note type). In LilyPond, this is now, for the time being, more of a mental exercise. When it comes to rhythm, LilyPond makes several assumptions:

1. The note value at the beginning of a piece is by default the quarter note;
2. The note value changes when one invokes such a rhythm;
3. The note value remains in effect until another value is invoked.³³

With this process, let's try a few examples:



Try replicating these processes before looking below at the answers. What are the two ways that they could have been written? One way:

1. The first example: `\relative c' '{ c4 c4 c8 c8 c8 c8 c16 c16 c8 c8 c8 c8 }`
2. The second example: `\relative c' '{ c4 c4 c16 c16 c16 c16 c8 c8 c2 c4 }`

This however, would have been tedious. There is good news, though, because the notation could have certainly been simplified. If we know that LilyPond assumes that the current note takes the previous rhythm unless changed at that moment, then how can these two examples have been cleaned up?

1. The first example: `\relative c' '{ c c c8 c c c c16 c c8 c c }34`
2. The second example: `\relative c' '{ c c c16 c c c c8 c c2 c4 }`

³³This is much like the law of motion which says an object stays in motion until another force acts upon it.

³⁴The value must always come right after the note name without a space.

Do you understand what is happening here? The compiler does not need to know the value of every note; it just needs to know when the value is changing. This is a radical assumption that is mimicked in WYSIWYGs. Some known issues when switching to this method of entering note values is primarily *forgetting* to enter the value you are thinking. This was a common problem of mine before I got really good at it. Occasionally, I still make this mistake.

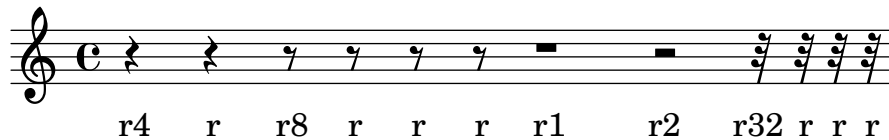
What about the augmentation dot? The augmentation dot comes right after the rhythm. Each note is a syntactical unit, so it must come in immediate succession to the note's value before a space is added. For example:³⁵



Below the example is what input had to occur in order to create the output. See how each note is a unit, a grammatical piece of this musical puzzle? Just like in forming sentences, certain things must be present in order to make sense.

Rests

When it comes to rests, they are written exactly like notes. There are two types of rests that can be used. One is a visible and the other an invisible (or spacer). To invoke a visible rest, use “r” and the length of note you wish for it to represent, like so:



When it comes to spacer rests, just use the letter “s” in place of “r” and you will create a space the length of the rhythmic value you give it.

Accidentals

LilyPond supports a wide variety of accidentals and note-naming properties. But of course the most common in music are the flat, double-flat, sharp, double-sharp, and natural:



These are invoked using simple phrases at the end of the note: `es` for flat, `is` for sharp, `eses` for double-flat, `isis` for double-sharp. Naturals are invoked by not using any phrase. Therefore, LilyPond does *not* assume that a flat or sharp carries until otherwise notated; that would be a disaster.

The *Notation Reference* describes several ways in which one can change the visual appearance of accidentals *in the output*. The default appearance, however, is one in which an accidental carries through the measure unless changed. This does not mean, however, that *in the input* you can write something like “ees” and just write “e” throughout that measure and the flat will continue through it. Every time

³⁵The augmentation dot can be double-dotted as well. In fact, you can add as many dots as you like!

you want an e-flat, whether you already wrote it before or not, you must write it as e-flat.³⁶ Here is an example:



The notation for this example is as follows: `\relative c' { ces4 bes a4. gis8 fisis gis ais bis cis2 }`. Notice how each note is its own unit. How would I get a natural to appear on the first “a” in the example? I would write, after the syntax, an “!” right before the beat value:

`\relative c' { ces4 bes a!4. gis8 fisis gis ais bis cis2 }`

And the resulting output would be:



Imagine the exclamation mark as a jolting reminder (!), an explicit command in the note name. The same reasonably goes for parenthesized accidentals; you only need to use a “?” after the note name. The resulting output would be:



So, in reality, both of these bits of syntax make complete sense when imagining how to “code” your input file. Both of these useful syntactical units can be used anywhere in the musical text.³⁷

Another important thing to remember (it will be brought about later on) is the fact that accidentals are part of the identity of the note; this comes before the placement of that note in terms of the octave when writing the input in the source file.

The Augmentation Dot

As mentioned before, this tool is used to augment the length of a note. When writing the input, it simply comes immediately after the rhythmic value of the note, and it cannot be used in absence of a rhythmic value. It must also come before any other articulations or expressions. The augmentation dot can be used *ad libitum* to any arbitrary end. For example, you can use one dot or as many dots as you would like:



The program calculates the degree of that augmentation dot internally and very naturally. Normally, most programs do not allow you to use more than three augmentation dots.³⁸

³⁶This has bothered people in the various forums of the internet, and to some degree some people find it tedious. Consider this, however: whenever you read a piece of music with a key signature, especially with accidentals in it, you don’t think of the other notes in the measure as being natural, do you? This is a “rewiring” thing I talked about in the Introduction: we as a culture of computer users with very interactive GUIs have grown accustomed to seeing menus and not needing to think about the “details” of a score; the program watches all of that for us! In exchange for this luxury, however, you are getting something far better: a very clean and tidy score that in most cases only needs moderate to no tweaking to get right and beautiful.

³⁷LilyPond goes above and beyond just these two methods of writing accidentals. You can adjust, change, and tweak any property you want, even to include microtonal linguistics, and even change how “!” or “?” behave by default.

³⁸Most users will say, however, that this is a pointless feature. I would suggest that a program’s ability to be expanded easily and economically is where the true power and authority comes from. Being able to draw an augmentation dot six times may never

Articulations

When it comes to articulations, LilyPond soars. It has one of the most complete sets of articulations and recognizable sets of articulations I have ever come across. And, as you may begin to realize, LilyPond is never limited by this fact. You can add to the list of articulations, define your own, and design, from scratch, anything you can imagine.

The compiler has a preconceived list of commands that the user may use to communicate with it. The most common articulations are:

- Marcato defined as `^`
- Accent defined as `>`
- Tenuto defined as `-`
- Portato defined as `_`
- Espressivo defined as `\espressivo`
- Staccato defined as `.`

The fermata is invoked with a backslash followed by the name: `\fermata`. Notice how these all have an “intuitive feel” to them, and they make “sense.” The caret key looks like a marcato accent; the greater-than-sign looks like an accent; a dash looks like a tenuto; a period looks like a staccato. In order to invoke these articulations (except for the fermata and the *espressivo*), one must use a direction definer *first*. This can be one of three things: a caret key “`^`” which indicates up; a dash “`-`” which indicates a neutral positions (the direction of which is ultimately defined by the compiler); and an underscore “`_`” which defines the articulation below the note. Here’s an example of all of them in play:



For this example, I used the neutral definer. But, if you were to use the caret key (pointing up) or the underscore (below), you would be telling to the compiler to override what it thinks the music should do and it would take your explicit command to do either instead.

The other great thing about LilyPond is the ability to “stack” articulations and other things given to the note. For example, one can put two different articulations to one specific note:³⁹



The code for that was `\relative c' { c2-^-- c^>_ . }`. Notice how on the second “c,” there were articulations given above and below the note. You may be wondering, “How does the compiler know when I’m doing an articulation?” Because of the unique and simplified setup of LilyPond, the program intuitively handles many of these things; you only need concern yourself with putting what you want where you want it.

LilyPond also has historic articulations, Kevian notation, and ancient music articulations. The *Notation Reference* lists how to do all of these.

really be necessary in real life, but the principle that you should not be limited by a program’s abilities is the key to this concept in LilyPond’s programming.

³⁹You can place as many articulations stacked onto one note as you would like.

Expression

LilyPond handles other expressions differently. Imagine being a computer program; you know that slurs and ties, hairpins, and the like do not look the same twice. They are not like mordents or portatos; they are defined on a moment by moment basis, spanning many different notes. For this reason, LilyPond separates and compiles expression commands separately from regular articulations, and has a separate engine to compile those requests in the source file.

Slurs and Ties

Slurs can define many different things, but LilyPond only recognizes three types of “slurs”: slur, phrasing slur, and tie. There are necessary and different components to create each event.

In the case of both the slur and the phrasing slur, LilyPond treats them like the curly brace or a square brace: if they are opened, they must be closed. LilyPond will generate a warning saying you have not begun or ended a slur if it is missing from your input file.

To invoke a slur, attach “()” to the first and last notes you wish to slur. Here’s an example:



If you use an articulation, the slurs are after that. Think of a slur (or any other thing related to a curve) as connecting whole units of music. The example above can be written as: `\relative c' { c4.(b8 a g f e d4 c b2) }`.

“But what if I want to nest slurs?” To do this, you need to use a phrasing slur. The difference between a phrasing slur and a slur syntactically is the use of the backslash. For example, `\relative c' { c4.\((b8 a g) f(e d4 c) b2\)}` produces:



When it comes to ties, you use the tilde “~” between the notes you want to tie. For example, { cis'2 ~ cis'4 c' } produces:



Do note that if you try to tie two different notes together, you will get an error message saying “...unterminated tie...” and you will be told where that unterminated tie is. The tie will also not display. You must also make sure that the notes match in terms of what octave they are in and whether or not they have an accidental.

Here is a combined example using all three types of slurs (ties, slurs, and phrasing slurs):



The code for this example is as follows: `\relative c' { \time 3/4 ces4(~ ces8 b16 ais~ ais4 b a g16(fis gis ais) b4 ais b)\fermata }`

You will find that LilyPond shapes slurs and other expression with grace and beauty. There will be few times that you will ever need to change the shape, and the initial output of slurs by default is far more readable than other notation programs.

Hairpins

Hairpins, or objects that describe an active change in dynamics, are invoked with a backslash. The three relevant syntactical units are the lesser-than symbol “<”, greater-than symbol “>”, and the exclamation point “!”. Both the lesser-than and greater-than symbols indicate what they look like: opening and closing hairpins. The exclamation point delineates the ending point of either.⁴⁰ For example:



The corresponding code would be: `\relative c' { c4< c c c c\! c> c c c c\mf }`.

Spanners

There are many corresponding text indications of gradual volume changes. These can be found in the *Learning Manual* and in the *Notation Reference*.

Dynamics

Dynamics are possibly the easiest to remember. They are invoked by a backslash, and LilyPond automatically supports the following: `\ppppp`, `\pppp`, `\ppp`, `\pp`, `\p`, `\mp`, `\mf`, `\f`, `\ff`, `\fff`, `\ffff`, `\fffff`, `\fp`, `\sf`, `\sff`, `\sp`, `\spp`, `\sfz`, and `\rfz`. They can be directly attached to a note, or the direction of the dynamics can be dictated by using the caret key (above) or underscore key (below) immediately before invoking the dynamic. Here's an example:



The corresponding code would be `{ c'4\mf c'~\fff c'~\sfz c'~\fp }`. Like before, LilyPond is capable of creating custom dynamics. The details of how to accomplish that are found in the *Notation Reference*. Remember that the dynamic is part of the identity of the individual note, so it therefore comes directly after the note (with expression) without a space.

Miscellaneous Notation

There are many (more) advanced notational things that LilyPond handles that are more common with beginners. Herein are some of those things.

⁴⁰Hairpins can also be stopped by dynamics, as shown in the example.

Beaming

LilyPond handles most of the beaming in the syntax by itself. There might, however, come a point when you desire to change the beaming of a group of notes, especially just a handful of times. LilyPond accomplishes this by use of the square bracket “[]”. The bracket occurs after the augmentation dot but can happen before any articulations. This example, `\relative c'' { cis8.[c16 c8] c[c] c[c c c c c c] }`, outputs the following:



Cadenza Notation

Cadenza notation is quite useful when marking chant, basic cadenza notation, or when you want to use a different length of measure without changing the time signature itself. It is invoked with the argument `\cadenzaOn` and it is turned off with `\cadenzaOff`. Do note that you will be responsible for notating the beaming structure of that measure (if need be), and you will also have to notate a bar line (explained in the next section).

Bars (Barlines and Bar Checks)

Bars are communicated in an intuitive way. If you desire to have something other than a traditional single barline, you will have to invoke `\bar` followed by an argument contained in double quotes (e.g. `\bar " . "`). Here, LilyPond goes above and beyond, and gives a plethora of options. Consider the following table:

command	output
<code>\bar " "</code>	
<code>\bar " . "</code>	
<code>\bar " "</code>	
<code>\bar " . "</code>	
<code>\bar " . . "</code>	
<code>\bar " . "</code>	
<code>\bar " . "</code>	
<code>\bar " ; "</code>	
<code>\bar " ! "</code>	

As it should come as no surprise, LilyPond doesn't stop here. There are endless possibilities with LilyPond. These barlines, however, cover the gamut of most usage.

Another fundamental concept in detecting errors in LilyPond is the use of a bar check. It is used at the end of any supposed measure in the input, and is notated by a single verticle line separator (|). This will cause the compiler to check to make sure that all of the beats add up appropriately for that measure.

Fingerings

Fingerings are dictated in the same manner as articulations. You are required to use a directional identifier (caret key or underscore key) or a dash (which allows the compiler to choose the direction). Fingerings can also be stacked, and they are delineated with numbers (imagine that!). The notation `{ c' '-1_2 c' '-3 c' '-2-4^5 }` would produce:



The *Notation Reference* describes many different ways of doing advanced fingerings.

Markups

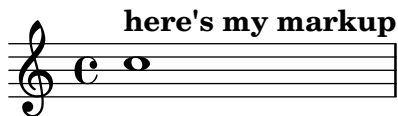
The `\markup{}` block is a system of entering other information, whether text-related or other-related, to be printed in the body of the score, that is not directly addressed by the musical information. It can be used outside of the `\score{}` block or within any elements in a score. Think of markup as a means by which you can enter other printed information other than music.⁴¹

Markups are perhaps the most versatile means of affecting text-based output. They are also incredibly powerful, because they can also integrate text and music. The *Notation Reference* and the *Learning Manual* both cover markups more extensively and comprehensively than I ever could, but here are some common uses.

The first use is modifying the default output of a header block. A good example of this is within the use of sacred music; whether a tune name is involved in a hymn or a song, you will generally want to mark it in small caps. LilyPond utilizes this very well, and it is invoked using the following: `\markup { \smallCaps "Text Goes Here" }`. There are many different things at play here, the first being that I am invoking “smallCaps” within the curly braces, and it is modifying the double quote container. Here is a complex series of markups:

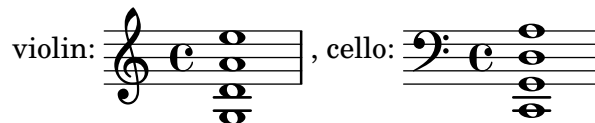
NSA *Fart*

This particular example invoked `\smallCaps`, `\italic`, and `\huge`. Multiple markup commands can be used on a single container of information. In addition, markups can be used on notes and music. In order to do that, you must explicitly indicate on which side of the note you wish the markup to occur. Here's an example:



The input for the above example is as follows: `\relative c' { c1^\markup { \bold "here's my markup" } }`

When using markup in a short file, however, don't forget to close all necessary open braces. LilyPond can also embed music within a markup:



The abilities are endless. To see a full list of markup commands, see section A.10 of the *Notation Reference* in the manuals.

⁴¹Though, it can be used to create musical examples, as you will see shortly.

Tuplets

Tuplets are essential to mixed meter and other such notation practices. From my own experience, writing tuplets in other WYSIWYG programs was a hassle and a nightmare. Luckily, LilyPond deals most effortlessly with tuplets.

To begin a tuplet, one must invoke the argument `\tuplet`, which is followed by two things: the ratio of said tuplet and an enclosed expression of that tuplet. That practically looks like this:

```
\tuplet %{\invoked%} 3/2 %{\first # equals the number of notes, second #
equals the number of notes it displaces %} { c8 d e }
```

Which will give you the following output:



Furthermore, you can nest triplets, and use them *ad libitum* to any length, time, or proportion you can imagine. Take, for example, this extraordinary demonstration (which even uses juxtaposed meter and irrational time signatures):

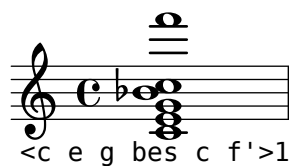
A complex musical score consisting of three staves. The first staff features a series of nested triplets with various time signatures: 4/20, 3/4, 1/5, and 1/4. The second staff is a simple 4/4 time signature. The third staff features a complex time signature of 4 + 3 + 2 over 12, followed by 9/6, 3/2, 9/6, 7/5, 2/7 + 3/14, and 7/8. The notation includes many accidentals and complex rhythmic groupings.

Chords

A more common need in music notation is to create chords. This notation is accomplished in LilyPond by using the less-than and greater-than symbols as a container (`<>`). If you are inside of an absolute pitch environment (the default of LilyPond), then the octave in which you define the chord appears:

A musical staff in treble clef with a common time signature 'C'. A chord is shown with a flat symbol above it. The chord consists of the notes C4, E4, B3, D4, and E3. Below the staff, the LilyPond code is shown: `<c' e' bes' d'' ees''>1`.

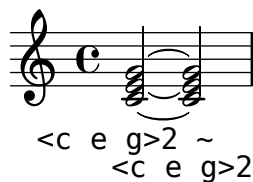
But, if you are in relative pitch, it depends on the note directly preceding it, and whether or not it is equal-to or less-than a fourth away:



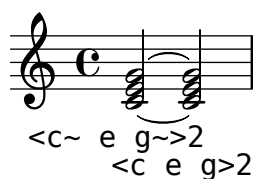
When entering in and exiting out of chords in `\relative` mode, the next note notated is always based on the first note of the chord given:



Chords can be tied; they can be note specific or whole chord specific. You can treat, in the input, a chord as an entire unit:



Or you can specify individual notes within the chord to be tied:



Either way, it is important to remember that when it comes to ties, the octave and note name need to match, or else you will get an error (it won't be a fatal error, it just will not show in the output).

Partial Measures

Partial measures are achieved by inserting at the beginning of the score `\partial` followed by what kind of note you want partially. For example, the input `\partial 8` followed by your score would achieve an eighth note pickup. Note that when you use partial measures, they must be at the beginning of the score, and not somewhere else in the score.

Bar Numbers

LilyPond creates bar numbers automatically at the beginning of each system starting on the second system. There are ways of toggling this command and changing its default behavior, but those methods are addressed in the *Notation Reference*.

Conclusion

Within this guide we have explored many of the intricate but interrelated and logical facets of LilyPond's extraordinary typesetting system. I have also showed you that though the syntax of LilyPond is a code of

sorts, it is certainly far from the complex languages and grammatical structures of actual programming languages. Inasmuch that you have read this guide, you should have a basic, functioning idea of how LilyPond works and how to enter simple notation. The Appendix contains many exercises and corresponding answers to hone your basic abilities with LilyPond. They should challenge you, but not beyond the scope of this guide. If you feel confident in your abilities and do not need to practice the exercises in the Appendix, then I would encourage you to start the *Learning Manual* found on LilyPond's website, and walk through it. Much of it you will already know, but I would still encourage you to try all the things it mentions and get a successful compilation of all the examples before continuing onward through it. Your understanding and habits with this program are contingent upon you understanding, comprehending, and processing this language. Should you become successful with it, it is only the beginning of a lifelong romance between you and the beautified scores you've created.

IMSLP.ORG has hundreds of thousands of scores waiting to be re-engraved for the public domain. I would encourage you to go there and find a couple of scores to re-engrave, and use them as real-life examples for honing your skills. Each score presents its own challenges, and there will be many moments where you will actually be googling for tweaks and instructions more than you will be typing code. Just make sure as you get used to the syntax and the various tweaks of the program to be aware of the version number and that the tweak is relevant to what you are trying to accomplish. Happy Pondering!

Post Script

Here's one massive example that makes use of all the concepts covered here in this guide. You are welcome to replicate it, or study it with the input directly below it.



```
\relative c { \clef bass \key f \major \time 4/4
bes4.\( c8 d( e f ges)
\cadenzaOn aes2\)\~ aes4. bes!8 aes[ g? f ees] \cadenzaOff \bar "."
<c e>2^_4_2 <c d>^\espressivo_5_1
bes16-. a-. bes-- ceses-- ees d ees f <d g->2
\time 3/4 <c g'>2 \clef treble bes''16^~ bes c d-^
\key ees \major f[( ees d c bes aes g f ees d c bes-.)]
\tuplet 7/6 { ees8[ ees] aes[ c] d[ ees f]\fermata } \bar "||"
R1*3/4\fermataMarkup \bar ""
\time 16/8 \clef bass bes,,,8[ d] g[ f g] d[ c bes] d4 ees aes, c16([ d ees f])
```

```
\cadenzaOn r4 r8 bes[ bes bes bes bes] r2\fermata \cadenzaOff \bar "|.|"
}
```

Appendix

Different Views of Content Authoring

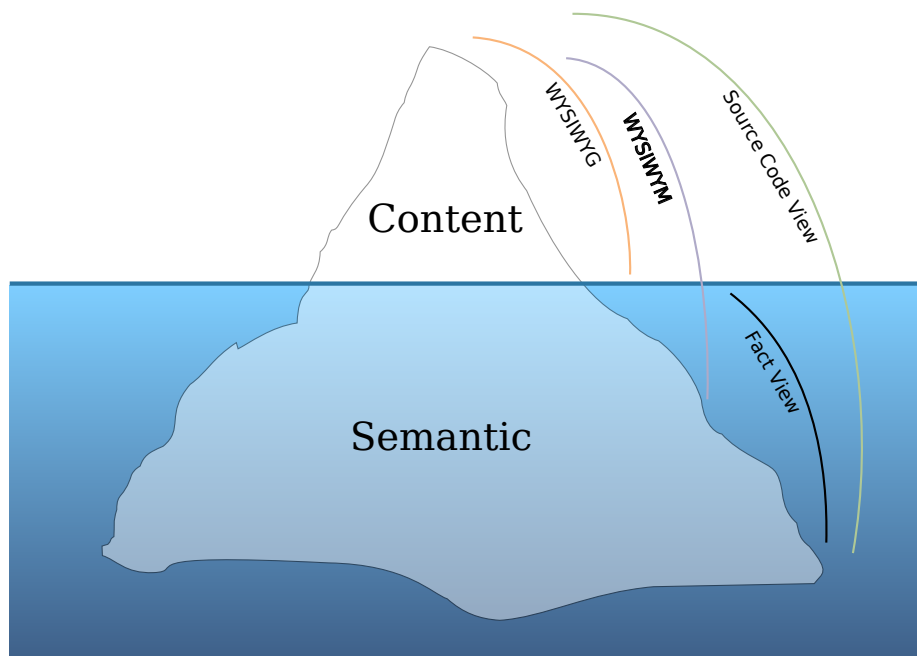


Figure 1: <http://en.wikipedia.org/wiki/WYSIWYM>

Main Examples

Replicate the following examples. The answers to these examples are found in the next section.

Basic Examples

- 1.
- 2.
- 3.

4. 

5. 

6. 

7. 

8. 

9. 

Advanced Examples

1. **Slow** ($\text{♩} = 1$)

5

hint: this piece contains no incomplete measures.

2.

3. $\text{♩} = 500$

5

4. **Fast**

3

6

Answers To Main Examples

The answers are formatted to consume the least amount of space, but within the answers themselves there are indents and spacing to help facilitate the reading of the file.

Basic Examples

1. `\relative c'' { \clef treble \time 4/4 f,4 g a b c8 b a g f4 e d fis a g f r r2 \bar "|." }`
2. `\relative c'' { \clef treble \time 3/4 b4 a g f bes a g f ees8 f g4 bes8 c d4 ees16 d c8 b a g4 \bar "|." }`
3. `\relative c'' { \time 6/4 \clef treble cis2 d1 ees2 d4 cis a2 a8 g fis e b' cis d2. fis2 fis, bes \bar "|." }`
4. `\relative c'' { \time 4/4 \clef treble d,16 fis a cis d fis d b a8. g16 fis e8 fis16 g fis g a \tuplet 3/2 { bes8 a bes } cis2 \bar ".." }`
5. `\relative c'' { \time 3/4 \key bes \major \clef treble f,4 bes f ees2. d4 bes' c8 d ees2 d8 c bes2 a8 g f4. ees8 f g aes2. \bar "||" }`
6. `\relative c'' { \key b \major \clef treble \time 5/4 b1~ b4 bes2 a4 gis f \tuplet 3/2 { ees8 f g~ } g8 b \tuplet 3/2 { ais8 b cis~ } cis8 fis, \tuplet 3/2 { cis'8 dis e~ } \time 4/4 e8 dis cis b a4. fis8 b1\fermata \bar "||" }`
7. `\relative c { \time 4/4 \clef bass \key bes \major \partial 4 f,4 bes8 c d ees f4 f8 ees f ees d c ees'4 d8 c bes a g f ees4 r d8 r r4 f r g16 f r8 ees16 d c r a' c, f, r \tuplet 3/2 { a8 ees' ces' } bes8 r d, r r f,4 r8 c2. \bar "!" }`
8. `\relative c'' { \key f \minor \clef treble \time 7/8 f,8[-. aes-. b]-. c4.-> e,8-. des!2-_ ees4.-^ e'4\fermata ees4-> des-! c8 bes8-. r bes-- r bes4. \key ees \major bes16([aes bes c d c f, aes b des c bes c) ees,-.] r4\fermata c8 e f g16 f ees d \bar "||" }`
9. `\relative c { \time 5/1 \clef bass c1(e g a) g,(d' c a) e(f b d') c(a g b,) b2 b b b b1 r \bar "|." }`

Advanced Examples

1. `\relative c { \time 3/8 \clef bass \key ges \major \tempo "Slow" 4. = 1 \cadenzaOn bes2\fermata \bar "||" \cadenzaOff bes4. aes8 ges aes \tuplet 3/2 { bes16([aes bes]) } \tuplet 3/2 { ces(bes ces) } \tuplet 3/2 { des([ees des]) } bes32([aes ges f ges aes bes) f'!]-. ges[-. ges-. ges-. ges]-. \time 4/8 ges,8-> ges-> ges-> ges-> ees'[-. ees-. ees-. bes16(ces)]~ ces8 ees f, c f[r ges'] r bes[ces, c aes] \bar "|." }`
2. `\relative c'' { \key f \major \clef treble \time 3/4 r4~\markup { \italic "hint: this piece contains no incomplete measures." } r r8 r s4 r r r8 r r s r r r16 r r2\fermata r8 s r s r s r\fermata \bar "||" }`
3. `\relative c'' { \key g \major \clef treble \time 5/4 \tempo 4 = 500 b4(\(e, dis2) e4(fis c b2) c'4(e dis e g)\) a,8(\(eis fis4 a bes2) a4(fis c b2)\) ais4(\(fis'16(g e' d) b(gis e8) dis16(fis a8) g4\) c,\fermata b(fis' b bes d) g2 g, d4 \bar ".." }`
4. `\relative c'' { \key des \major \time 4/4 \clef treble \tempo "Fast" bes4-. aes-- ges-. des-- ees2.-_ des8(ees f ges aes bes) c([aes f des c bes des des'-.)] c([aes f des c f c' e]) f(c f, ges) g(d cis ees) f des aes' bes-. ces4 bes aes2 \bar "|." }`

Real Life Example

After you have gone away and walked through the *Learning Manual*, and you feel confident in your “tweaking” skills or your “Google-search-for-tweak” skills, try out this example. It combines a host of different techniques to bring the piece together. It is also safe to use predefined templates either through Frescobaldi (or any other front-end program of your choice) or via the internet.

3 Moods

For Low Reed Instrument

Joshua Nichols

I Adagio lugubre (♩ = 50)

pp < sf > p

7 pp sub. f sub. p

11 mf f

16 ffff

21 ff sub. pp

28 pp

33 p p

II **Vivo** (♩ = 150)

8

13

19

26

32

38

ff

41

p

47

sub. ff

53

f *poco decr.*

59

sfz

64

f *mp* *p* *pp* *f* *ff*

III Moderato (♩ = 87 - 95)

p *mf* *p* *f* *mp*

6

f *ff* *p* *f* *pp*

11

p *f* *p*

16

f *sub. pp* *f*

20

p *sub. f* *sub. p*

24 (♩ = ♩)

mf *pp* *poco cresc. e accel.*

29

ff