LilyPond

The music typesetter

Contributor's Guide

The LilyPond development team

This manual documents contributing to LilyPond version 2.14.1. It discusses technical issues and policies that contributors should follow.

This manual is not intended to be read sequentially; new contributors should only read the sections which are relevant to them. For more information about different jobs, see Section "Help us" in *General Information*.

For more information about how this manual fits with the other documentation, or to read this manual in other formats, see Section "Manuals" in *General Information*.

If you are missing any manuals, the complete documentation can be found at `http://www.lilypond.org/`.

Copyright © 2007–2011 by the authors.

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

For LilyPond version 2.14.1

Table of Contents

1 Introduction to contributing 1

1.1 Help us 1
1.2 Overview of work flow 2
1.3 Mentors 3

2 Quick start 5

2.1 Lilydev 5
2.1.1 Installing lilydev 5
2.1.2 Configuring lilydev in virtualbox 6
2.1.3 Using lilydev 6
2.2 Using lily-git 6
Install and configuration of `lily-git.tcl` 7
Daily use of `lily-git.tcl` 7
2.3 Compiling with lilydev 8
2.4 Now start work! 9

3 Working with source code 11

3.1 Manually installing lily-git.tcl 11
3.2 Starting with Git 11
3.2.1 Setting up 12
Installing Git 12
Initializing a repository 12
Configuring Git 12
3.2.2 Downloading remote branches 13
Organization of remote branches 13
LilyPond repository sources 13
Downloading individual branches 14
Downloading all remote branches 15
Other branches 15
3.3 Basic Git procedures 15
3.3.1 The Git contributor's cycle 15
3.3.2 Pulling and rebasing 15
3.3.3 Using local branches 16
Creating and removing branches 16
Listing branches and remotes 17
Checking out branches 17
Merging branches 17
3.3.4 Commits and patches 17
Understanding commits 17
Making commits 18
Commit messages 18
Making patches 19
Uploading a patch for review 19
3.4 Advanced Git procedures 21
3.4.1 Advanced Git concepts 21
3.4.2 Resolving conflicts 22
3.4.3 Reverting all local changes 22

3.4.4 Working with remote branches ... 22
3.4.5 Git log ... 23
3.4.6 Applying remote patches ... 23
3.4.7 Sending and receiving patches via email ... 23
3.4.8 Cleaning up multiple patches ... 24
3.4.9 Commit access ... 24
3.5 Git on Windows ... 27
3.5.1 Background to nomenclature ... 27
3.5.2 Installing git ... 27
3.5.3 Initialising Git ... 27
3.5.4 Git GUI ... 28
3.5.5 Personalising your local git repository ... 28
3.5.6 Checking out a branch ... 29
3.5.7 Updating files from 'remote/origin/master' ... 29
3.5.8 Editing files ... 29
3.5.9 Sending changes to 'remotes/origin/master' ... 30
3.5.10 Resolving merge conflicts ... 30
3.5.11 Other actions ... 30
3.6 Repository directory structure ... 31
3.7 Other Git documentation ... 33

4 Compiling ... 34

4.1 Overview of compiling ... 34
4.2 Requirements ... 34
4.2.1 Requirements for running LilyPond ... 34
4.2.2 Requirements for compiling LilyPond ... 34
4.2.3 Requirements for building documentation ... 35
4.3 Getting the source code ... 36
4.4 Configuring `make` ... 36
4.4.1 Running `./autogen.sh` ... 36
4.4.2 Running `../configure` ... 37
Configuration options ... 37
Checking build dependencies ... 37
Configuring target directories ... 37
4.5 Compiling LilyPond ... 38
4.5.1 Using `make` ... 38
4.5.2 Saving time with the '`-j`' option ... 38
4.5.3 Compiling for multiple platforms ... 38
4.5.4 Useful `make` variables ... 39
4.6 Post-compilation options ... 39
4.6.1 Installing LilyPond from a local build ... 39
4.6.2 Generating documentation ... 39
Documentation editor's edit/compile cycle ... 39
Building documentation ... 39
Saving time with `CPU_COUNT` ... 40
AJAX search ... 40
Installing documentation ... 41
Building documentation without compiling ... 41
4.6.3 Testing LilyPond binary ... 42
4.7 Problems ... 42
Bison 1.875 ... 42
Compiling on MacOS X ... 42
Solaris ... 43
FreeBSD ... 43

International fonts ... 43
Using lilypond python libraries ... 44
4.8 Concurrent stable and development versions ... 44
4.9 Build system ... 44

5 Documentation work ... 45

5.1 Introduction to documentation work ... 45
5.2 Documentation suggestions ... 45
5.3 Texinfo introduction and usage policy ... 46
5.3.1 Texinfo introduction ... 46
5.3.2 Documentation files ... 47
5.3.3 Sectioning commands ... 47
5.3.4 LilyPond formatting ... 48
5.3.5 Text formatting ... 50
5.3.6 Syntax survey ... 50
Comments ... 50
Cross references ... 50
External links ... 51
Fixed-width font ... 51
Indexing ... 52
Lists ... 52
Special characters ... 53
Miscellany ... 53
5.3.7 Other text concerns ... 54
5.4 Documentation policy ... 54
5.4.1 Books ... 54
5.4.2 Section organization ... 55
5.4.3 Checking cross-references ... 56
5.4.4 General writing ... 56
5.4.5 Technical writing style ... 57
5.5 Tips for writing docs ... 57
5.6 Scripts to ease doc work ... 58
5.7 Docstrings in scheme ... 59
5.8 Translating the documentation ... 59
5.8.1 Getting started with documentation translation ... 59
Translation requirements ... 60
Which documentation can be translated ... 60
Starting translation in a new language ... 60
5.8.2 Documentation translation details ... 60
Files to be translated ... 61
Translating the Web site and other Texinfo documentation ... 62
Adding a Texinfo manual ... 64
5.8.3 Documentation translation maintenance ... 65
Check state of translation ... 65
Updating documentation translation ... 65
Updating translation committishes ... 66
5.8.4 Translations management policies ... 67
Maintaining without updating translations ... 67
Managing documentation translation with Git ... 69
5.8.5 Technical background ... 69

6 Website work ... 71
- 6.1 Introduction to website work ... 71
- 6.2 Uploading and security ... 71
- 6.3 Debugging website and docs locally ... 73
- 6.4 Translating the website ... 73

7 LSR work ... 74
- 7.1 Introduction to LSR ... 74
- 7.2 Adding and editing snippets ... 74
- 7.3 Approving snippets ... 75
- 7.4 LSR to Git ... 75
- 7.5 Fixing snippets in LilyPond sources ... 75
- 7.6 Renaming a snippet ... 76
- 7.7 Updating LSR to a new version ... 76

8 Issues ... 78
- 8.1 Introduction to issues ... 78
- 8.2 Bug Squad setup ... 78
- 8.3 Bug Squad checklists ... 79
- 8.4 Issue classification ... 81
- 8.5 Adding issues to the tracker ... 83
- 8.6 Patch handling ... 83
- 8.7 Summary of project status ... 84

9 Regression tests ... 86
- 9.1 Introduction to regression tests ... 86
- 9.2 Precompiled regression tests ... 86
- 9.3 Compiling regression tests ... 87
- 9.4 Regtest comparison ... 87
- 9.5 Finding the cause of a regression ... 88
- 9.6 Memory and coverage tests ... 89
- 9.7 MusicXML tests ... 90

10 Programming work ... 91
- 10.1 Overview of LilyPond architecture ... 91
- 10.2 LilyPond programming languages ... 93
  - 10.2.1 C++ ... 93
  - 10.2.2 Flex ... 93
  - 10.2.3 GNU Bison ... 93
  - 10.2.4 GNU Make ... 93
  - 10.2.5 GUILE or Scheme ... 93
  - 10.2.6 MetaFont ... 93
  - 10.2.7 PostScript ... 93
  - 10.2.8 Python ... 93
- 10.3 Programming without compiling ... 93
  - 10.3.1 Modifying distribution files ... 94
  - 10.3.2 Desired file formatting ... 94
- 10.4 Finding functions ... 94
  - 10.4.1 Using the ROADMAP ... 94
  - 10.4.2 Using grep to search ... 94
  - 10.4.3 Using git grep to search ... 95
  - 10.4.4 Searching on the git repository at Savannah ... 95

10.5 Code style ... 95
10.5.1 Languages ... 95
10.5.2 Filenames ... 95
10.5.3 Indentation ... 95
10.5.4 Naming Conventions ... 97
10.5.5 Broken code ... 98
10.5.6 Code comments ... 98
10.5.7 Handling errors ... 98
10.5.8 Localization ... 98
10.6 Debugging LilyPond ... 100
10.6.1 Debugging overview ... 100
10.6.2 Debugging C++ code ... 100
10.6.3 Debugging Scheme code ... 101
10.7 Tracing object relationships ... 103
10.8 Adding or modifying features ... 104
10.8.1 Write the code ... 104
10.8.2 Write regression tests ... 104
10.8.3 Write convert-ly rule ... 105
10.8.4 Automatically update documentation ... 105
10.8.5 Manually update documentation ... 105
10.8.6 Edit changes.tely ... 106
10.8.7 Verify successful build ... 106
10.8.8 Verify regression tests ... 106
10.8.9 Post patch for comments ... 107
10.8.10 Push patch ... 107
10.8.11 Closing the issues ... 107
10.9 Iterator tutorial ... 107
10.10 Engraver tutorial ... 107
10.10.1 Useful methods for information processing ... 107
10.10.2 Translation process ... 108
10.10.3 Preventing garbage collection for SCM member variables ... 108
10.10.4 Listening to music events ... 108
10.10.5 Acknowledging grobs ... 108
10.10.6 Engraver declaration/documentation ... 109
10.11 Callback tutorial ... 109
10.12 LilyPond scoping ... 109
10.13 LilyPond miscellany ... 110
10.13.1 Spacing algorithms ... 110
10.13.2 Info from Han-Wen email ... 110
10.13.3 Music functions and GUILE debugging ... 114

11 Release work ... 115
11.1 Development phases ... 115
11.2 Minor release checklist ... 115
11.3 Major release checklist ... 116
11.4 Release extra notes ... 118

12 Build system notes 120
12.1 Build system overview 120
12.2 Tips for working on the build system 120
12.3 General build system notes 120
12.3.1 How stepmake works 121
12.4 Doc build 124
12.4.1 Building a bibliography 124
12.5 Website build 125
12.6 Building an Ubuntu distro 127

13 Administrative policies 128
13.1 Meta-policy for this document 128
13.2 Meisters 128
13.3 Administrative mailing list 129
13.4 Grand Organization Project (GOP) 129
13.4.1 Motivation 129
13.4.2 Ongoing jobs 130
13.4.3 Policy decisions 130
13.5 Grand LilyPond Input Syntax Standardization (GLISS) 133
13.5.1 Specific GLISS issues 134
13.6 Unsorted policies 136

Appendix A GNU Free Documentation License 138

1 Introduction to contributing

This chapter presents a quick overview of ways that people can help LilyPond.

1.1 Help us

We need you!

The LilyPond development team is quite small; we really want to get more people involved. Please consider helping your fellow LilyPond users by contributing!

Even working on small tasks can have a big impact: taking care of them allows experienced developers work on advanced tasks, instead of spending time on those simple tasks.

Simple tasks

No source code or compiling required!

- Mailing list support: answer questions from fellow users.
- Bug reporting: help users create proper Section "Bug reports" in *General Information*, and/or join the Bug Squad to organize Section "Issues" in *Contributor's Guide*.
- Documentation: small changes can be proposed by following the guidelines for Section "Documentation suggestions" in *Contributor's Guide*.
- LilyPond Snippet Repository (LSR): create and fix snippets following the guidelines in Section "Adding and editing snippets" in *Contributor's Guide*.
- Discussions, reviews, and testing: the developers often ask for feedback about new documentation, potential syntax changes, and testing new features. Please contribute to these discussions!

Advanced tasks

These jobs generally require that you have the source code and can compile LilyPond.

> **Note:** We suggest that contributors using Windows or MacOS X do **not** attempt to set up their own development environment; instead, use Lilydev as discussed in Section "Quick start" in *Contributor's Guide*.

Contributors using Linux or FreeBSD may also use Lilydev, but if they prefer their own development environment, they should read Section "Working with source code" in *Contributor's Guide*, and Section "Compiling" in *Contributor's Guide*.

- Documentation: for large changes, see Section "Documentation work" in *Contributor's Guide*.
- Website: the website is built from the normal documentation source. See the info about documentation, and also Section "Website work" in *Contributor's Guide*.
- Translations: see Section "Translating the documentation" in *Contributor's Guide*, and Section "Translating the website" in *Contributor's Guide*.
- Bugfixes or new features: the best way to begin is to join the Frogs, and read Section "Programming work" in *Contributor's Guide*.

Projects

Frogs

Website and mailing list:

http://frogs.lilynet.net

The Frogs are ordinary LilyPond users who have chosen to get involved in their favorite software's development. Fixing bugs, implementing new features, documenting the source code: there's a lot to be done, but most importantly: this is a chance for everyone to learn more about LilyPond, about Free Software, about programming... and to have fun. If you're curious about any of it, then the word is: *Join the Frogs!*

Grand LilyPond Input Syntax Standardization

Website: Section "Grand LilyPond Input Syntax Standardization (GLISS)" in *Contributor's Guide*.

GLISS will stabilize the (non-tweak) input syntax for the upcoming LilyPond 3.0. After updating to 3.0, the input syntax for untweaked music will remain stable for the foreseeable future.

We will have an extensive discussion period to determine the final input specification.

> **Note:** GLISS will start shortly after 2.14 is released.

Grand Organizing Project

Website: Section "Grand Organization Project (GOP)" in *Contributor's Guide*.

GOP will be our big recruiting drive for new contributors. We desperately need to spread the development duties (including "simple tasks" which require no programming or interaction with source code!) over more people. We also need to document knowledge from existing developers so that it does not get lost.

Unlike most "Grand Projects", GOP is not about adding huge new features or completely redesigning things. Rather, it is aimed at giving us a much more stable foundation so that we can move ahead with larger tasks in the future.

> **Note:** GOP will start shortly after the 2.14 release.

1.2 Overview of work flow

> **Ultra-short summary for Unix developers**: source code is at `git://git.sv.gnu.org/lilypond.git`. Documentation is built with Texinfo, after pre-processing with `lilypond-book`. Send well-formed patches to `lilypond-devel@gnu.org`.

Git is a *version control system* that tracks the history of a program's source code. The LilyPond source code is maintained as a Git repository, which contains:

- all of the source files needed to build LilyPond, and
- a record of the entire history of every change made to every file since the program was born.

The 'official' LilyPond Git repository is hosted by the GNU Savannah software forge at http://git.sv.gnu.org. Although, since Git uses a *distributed* model, technically there is no central repository. Instead, each contributor keeps a complete copy of the entire repository (about 116M).

Changes made within one contributor's copy of the repository can be shared with other contributors using *patches*. A patch is a simple text file generated by the `git` program that

indicates what changes have been made (using a special format). If a contributor's patch is approved for inclusion (usually through the mailing list), someone on the current development team will *push* the patch to the official repository.

The Savannah software forge provides two separate interfaces for viewing the LilyPond Git repository online: *cgit* and *gitweb*. The cgit interface should work faster than gitweb in most situations, but only gitweb allows you to search through the source code using `grep`, which you may find useful. The cgit interface is at http://git.sv.gnu.org/cgit/lilypond.git/ and the gitweb interface is at http://git.sv.gnu.org/gitweb/?p=lilypond.git.

Git is a complex and powerful tool, but tends to be confusing at first, particularly for users not familiar with the command line and/or version control systems. We have created the `lily-git` graphical user interface to ease this difficulty.

Compiling ('building') LilyPond allows developers to see how changes to the source code affect the program itself. Compiling is also needed to package the program for specific operating systems or distributions. LilyPond can be compiled from a local Git repository (for developers), or from a downloaded tarball (for packagers). Compiling LilyPond is a rather involved process, and most contributor tasks do not require it.

Contributors can contact the developers through the 'lilypond-devel' mailing list. The mailing list archive is located at http://lists.gnu.org/archive/html/lilypond-devel/. If you have a question for the developers, search the archives first to see if the issue has already been discussed. Otherwise, send an email to lilypond-devel@gnu.org. You can subscribe to the developers' mailing list here: http://lists.gnu.org/mailman/listinfo/lilypond-devel.

> **Note:** Contributors on Windows or MacOS X wishing to compile code or documentation are strongly advised to use our Ubuntu LilyPond Developer Remix, as discussed in Chapter 2 [Quick start], page 5.

1.3 Mentors

We have a semi-formal system of mentorship, similar to the medieval "journeyman/master" training system. New contributors will have a dedicated mentor to help them "learn the ropes".

> **Note:** This is subject to the availability of mentors; certain jobs have more potential mentors than others.

Contributor responsibilities

1. Ask your mentor which sections of the CG you should read.
2. If you get stuck for longer than 10 minutes, ask your mentor. They might not be able to help you with all problems, but we find that new contributors often get stuck with something that could be solved/explained with 2 or 3 sentences from a mentor.
3. If you have been working on a task much longer than was originally estimated, stop and ask your mentor. There may have been a miscommunication, or there may be some time-saving tips that could vastly simply your task.
4. Send patches to your mentor for initial comments.
5. Inform your mentor if you're going to be away for a month, or if you leave entirely. Contributing to lilypond isn't for everybody; just let your mentor know so that we can reassign that work to somebody else.
6. Inform your mentor if you're willing to do more work – we always have way more work than we have helpers available. We try to avoid overwhelming new contributors, so you'll be given less work than we think you can handle.

Mentor responsibilities

1. Respond to questions from your contributor(s) promptly, even if the response is just "sorry, I don't know" or "sorry, I'm very busy for the next 3 days; I'll get back to you then". Make sure they feel valued.
2. Inform your contributor(s) about the expected turnaround for your emails – do you work on lilypond every day, or every weekend, or what? Also, if you'll be unavailable for longer than usual (say, if you normally reply within 24 hours, but you'll be at a conference for a week), let your contributors know. Again, make sure they feel valued, and that your silence (if they ask a question during that period) isn't their fault.
3. Inform your contributor(s) if they need to do anything unusual for the builds, such as doing a "make clean / doc-clean" or switching git branches (not expected, but just in case...)
4. You don't need to be able to completely approve patches. Make sure the patch meets whatever you know of the guidelines (for doc style, code indentation, whatever), and then send it on to the frog list or -devel for more comments. If you feel confident about the patch, you can push it directly (this is mainly intended for docs and translations; code patches should almost always go to -devel before being pushed).
5. Keep track of patches from your contributor. If you've sent a patch to -devel, it's your responsibility to pester people to get comments for it, or at very least add it to the google tracker.

2 Quick start

Want to submit a patch for LilyPond? Great! This chapter is designed to let you do this as quickly and easily as possible.

It is not possible to compile LilyPond on Windows, and extremely difficulty to compile it on MacOS X. We have therefore made a 'remix' of Ubuntu which includes all necessary dependencies to compile both LilyPond and the documentation. This can be run inside a virtual machine without disturbing your main operating system. The full name is "Ubuntu LilyPond Developer Remix", but we refer to it as "lilydev" for short.

> **Advanced note:** experienced developers may prefer to use their own development environment. It may be instructive to skim over these instructions, but be aware that this chapter is intended for helpful users who may have never created a patch before.

2.1 Lilydev

This section discusses how to install and use the Ubuntu LilyPond Development Remix.

2.1.1 Installing lilydev

1. Install some virtualization software.

   Any virtualization tool can be used, but we recommend VirtualBox:

   `http://www.virtualbox.org/wiki/Downloads`

   In virtualization terminology, your main operating system is the "host", while lilydev is the "guest".

2. Download the Ubuntu LilyPond Developer Remix disk image: (approximately 1 GB)

   `http://files.lilynet.net/ubuntu-lilydev-remix-1.1.iso`

   Advanced note: Some users might want these files, but if you don't recognize what they are, then you don't want them:
   `http://files.lilynet.net/ubuntu-lilydev-remix-1.1.iso.md5`

3. Create a music new "virtual machine" inside your virtualization software.

   If possible, use at least 700 MB of RAM (1GB would be better) for the virtual machine, and use "dynamically expanding storage" for the virtual hard drive. A complete compile of everything (code, docs, regression tests) can reach 10 GB.

4. Install '`ubuntu-lilydev-remix-1.1.iso`' as the "guest" operating system on your virtualized system.

   1. When '`ubuntu-lilydev-remix-1.1.iso`' boots, it shows an ISOLINUX `boot:` prompt. Type:

      ```
      install
      ```

   2. At the "Prepare disk space" stage, do not be afraid to select "Erase and use the entire disk", since this refers to your *virtual disk*, not your machine's actual hard drive.

   3. When prompted to remove the installation CD, go to Devices → CD/DVD Devices and de-select '`ubuntu-lilydev-remix-1.1.iso`'.

      Advanced note: The latest version of lilydev is based on Ubuntu 10.04.1; if you encounter any difficulties installing it, search for one of the many tutorials for installing that particular version of Ubuntu as a guest operating system.

5. Do any extra configuration for your virtualization software.

   There are additional instructions for VirtualBox in Section 2.1.2 [Configuring lilydev in virtualbox], page 6.

If you use other virtualization software, then follow the normal procedures for your virtualization software with Ubuntu as the client.

Known issues and warnings

Not all hardware is supported in all virtualization tools. In particular, some contributors have reported problems with USB devices. If you would like to investigate further, then look for help for your virtualization tool using your normal OS as the "host" and Ubuntu as the "client".

2.1.2 Configuring lilydev in virtualbox

VirtualBox has extra "guest additions" which can make the virtualization easier to use (full-screen, easy file sharing between host and guest operating systems, shared clipboards, etc).

1. In *VirtualBox*, select Devices → Install Guest Additions....
2. In *Ubuntu*, select Places → VBOXADDITIONS_. A file-system window will open.
3. Double-click on the '`autorun.sh`' file, then select "Run in Terminal", and enter your password when prompted.
4. Once the script is finished, "eject" the virtual CD, and then go to Devices → CD/DVD Devices and de-select '`VBoxGuestAdditions.iso`'.
5. Restart Ubuntu to complete the installation.

   Advanced note: If you do any kernel upgrades, you may need to re-run these VBOXADDITIONS instructions.

Some other steps may be helpful:

- In the settings for the virtual machine, set the network to Bridged mode to allow you to access shared folders on your Windows host.
- Set up any additional features, such as 'Shared Folders' between your main operating system and ubuntu. This is distinct from the networked share folders in Windows. Consult external documentation for this step.

  Some longtime contributors have reported that 'shared folders' are rarely useful and not worth the fuss, particularly since files can be shared over a network instead.

2.1.3 Using lilydev

If you are not familiar with Linux, it may be beneficial to read a couple of "introduction to Ubuntu" webpages.

- One particular change from Windows and MacOS X is that most software should be installed with your "package manager"; this vastly simplifies the process of installing and configuring software. Go to Applications → Ubuntu Software Center.
- The rest of this manual assumes that you are using the command-line; double-click on the 'Terminal' icon on the desktop.
- Pasting into a terminal is done with `Ctrl+Shift+v`.
- The "Places" top-menu has shortcuts to a graphical "navigator" like Windows Explorer or the MacOS X Finder.
- Right-click allows you to edit a file with gedit. We recommend using gedit.
- Some contributors have recommended: (pdf available for free)

  `http://www.ubuntupocketguide.com/`

2.2 Using lily-git

`lily-git.tcl` is a graphical tool to help you access and share changes to the lilypond source code.

Install and configuration of `lily-git.tcl`

> **Note:** The rest of this manual assumes that you are using the command-line; double-click on the 'Terminal' icon on the desktop.

1. Type (or copy&paste) into the Terminal:

   ```
   lily-git.tcl
   ```

2. Click on the "Get source" button.

   This will create a directory called '`lilypond-git/`' within your home directory, and will download the source code into that directory (around 55Mb). When the process is finished, the "Command output" window will display "Done", and the button label will change to say "Update source".

   > **Note:** Some contributors have reported that nothing happens at this step. If this occurs, then try again in a few minutes – we suspect that this is an intermittant network problem. If the problem persists, please ask for help.

3. Navigate to the '`lilypond-git/`' directory to view the source files.

You should now progress to Section 2.3 [Compiling with lilydev], page 8.

> **Note:** Throughout the rest of this manual, most command-line input should be entered from '`~/lilypond-git/`'. This is referred to as the *top source directory*.

Advanced note: the "Get source" button does not fetch the entire history of the git repository, so utilities like `gitk` will only be able to display the most recent additions. As you continue to work with `lily-git.tcl`, the "Update source" button will take any new additions and add it to whatever is currently in your repository's history.

Daily use of `lily-git.tcl`

> **Note:** Only work on one set of changes at once. Do not start work on any new changes until your first set has been accepted.

1. Update source

At the beginning of each session of lilypond work, you should click the "Update source" button to get the latest changes to the source code.

> **Note:** In some rare and unfortunate circumstances, this will result in a *merge conflict*. If this occurs, follow the instructions for "Abort changes", below. Your work will not be lost.

2a. New local commit

A single commit typically represents one logical set of related changes (such as a bug-fix), and may incorporate changes to multiple files at the same time.

When you're finished making the changes for a commit, click the "New local commit" button. This will open the "Git Commit Message" window. The message header is required, and the message body is optional.

After entering a commit message, click "OK" to finalize the commit.

> **Advanced note:** for more information regarding commits and commit messages, see Section 3.3.4 [Commits and patches], page 17.

2b. Amend previous commit

You can go back and make changes to the most recent commit with the "Amend previous commit" button. This is useful if a mistake is found after you have clicked the "New local commit" button.

To amend the most recent commit, re-edit the source files as needed and then click the "Amend previous commit" button. The earlier version of the commit is not saved, but is replaced by the new one.

> **Note:** This does not update the patch **files**; if you have a patch file from an earlier version of the commit, you will need to make another patch set when using this feature. The old patch file will not be saved, but will be replaced by the new one after you click on "Make patch set".

3. Make patch set

Before making a patch set from any commits, you should click the "Update source" button to make sure the commits are based on the most recent remote snapshot.

When you click the "Make patch set" button, `lily-git.tcl` will produce patch files for any new commits, saving them to the current directory. The command output will display the name of the new patch files near the end of the output:

```
0001-CG-add-lily-git-instructions.patch
Done.
```

Send patch files to the appropriate place:

- If you have a mentor, send it to them via email.
- New contributors should send the patch attached to an email to `frogs@lilynet.net`. Please add "[PATCH]" to the subject line.
- Translators should send patches to `translations@lilynet.net`.
- More experienced contributors should upload the patch for web-based review. This requires additional software and use of the command-line; see [Uploading a patch for review], page 19.

The "Abort changes – Reset to origin" button

> **Note:** Only use this if your local commit history gets hopelessly confused!

The button labeled "Abort changes – Reset to origin" will copy all changed files to a subdirectory of '`lilypond-git/`' named '`aborted_edits/`', and will reset the repository to the current state of the remote repository (at `git.sv.gnu.org`).

2.3 Compiling with lilydev

Lilydev is our 'remix' of Ubuntu which contains all the necessary dependencies to do lilypond development; for more information, see Section "Lilydev" in *Contributor's Guide*.

Preparing the build

To prepare the build directory, enter (or copy&paste) the below text. This should take less than a minute.

```
cd ~/lilypond-git/
sh autogen.sh --noconfigure
mkdir -p build/
cd build/
../configure
```

Building `lilypond`

Compiling lilypond will likely take between 5 and 60 minutes, depending on your computer's speed and available RAM. We recommend that you minimize the terminal window while it is building; this can have a non-negligible effect on compilation speed.

```
cd ~/lilypond-git/build/
make
```

You may run the compiled `lilypond` with:

```
cd ~/lilypond-git/build/
out/bin/lilypond my-file.ly
```

Building the documentation

Compiling the documentation is a much more involved process, and will likely take 2 to 10 hours.

```
cd ~/lilypond-git/build/
make doc
```

The documentation is put in '`out-www/offline-root/`'. You may view the html files by entering the below text; we recommend that you bookmark the resulting page:

```
firefox ~/lilypond-git/build/out-www/offline-root/index.html
```

Installing

Don't. There is no reason to install lilypond within lilydev. All development work can (and should) stay within the '`$HOME/lilypond-git/`' directory, and any personal composition or typesetting work should be done with an official GUB release.

Problems and other options

To select different build options, or isolate certain parts of the build, or to use multiple CPUs while building, read Chapter 4 [Compiling], page 34.

In particular, contributors working on the documentation should be aware of some bugs in the build system, and should read the workarounds in Section 4.6.2 [Generating documentation], page 39.

2.4 Now start work!

Lilydev users may now skip to the chapter which is aimed at their intended contributions:

- Chapter 5 [Documentation work], page 45
- Section 5.8 [Translating the documentation], page 59
- Chapter 6 [Website work], page 71
- Chapter 9 [Regression tests], page 86
- Chapter 10 [Programming work], page 91

These chapters are mainly intended for people not using LilyDev, but they contain extra information about the "behind-the-scenes" activities. We recommend that you read these at your leisure, a few weeks after beginning work with LilyDev.

- Chapter 3 [Working with source code], page 11
- Chapter 4 [Compiling], page 34

3 Working with source code

> **Note:** New contributors should read Chapter 2 [Quick start], page 5, and in particular Section 2.2 [Using lily-git], page 6, instead of this chapter.

Advanced contributors will find this material quite useful, particularly if they are working on major new features.

3.1 Manually installing lily-git.tcl

We have created an easy-to-use GUI to simplify git for new contributors. If you are comfortable with the command-line, then skip ahead to Section 3.2 [Starting with Git], page 11.

> **Note:** These instructions are only for people who are *not* using Section 2.1 [Lilydev], page 5.

1. If you haven't already, download and install Git.
   - Windows users: download the `.exe` file labeled "Full installer for official Git" from:
     `http://code.google.com/p/msysgit/downloads/list`
   - Other operating systems: either install `git` with your package manager, or download it from the "Binaries" section of:
     `http://git-scm.com/download`
2. Download the `lily-git.tcl` script from:
   `http://git.sv.gnu.org/cgit/lilypond.git/plain/scripts/auxiliar/lily-git.tcl`
3. To run the program from the command line, navigate to the directory containing `lily-git.tcl` and enter:
   ```
   wish lily-git.tcl
   ```
4. Click on the "Get source" button.
   This will create a directory called '`lilypond-git/`' within your home directory, and will download the source code into that directory (around 55Mb). When the process is finished, the "Command output" window will display "Done", and the button label will change to say "Update source".
5. Navigate to the '`lilypond-git/`' directory to view the source files.

> **Note:** Throughout the rest of this manual, most command-line input should be entered from '`~/lilypond-git/`'. This is referred to as the *top source directory*.

Further instructions are in [Daily use of lily-git.tcl], page 7.

Advanced note: the "Get source" button does not fetch the entire history of the git repository, so utilities like `gitk` will only be able to display the most recent additions. As you continue to work with `lily-git.tcl`, the "Update source" button will take any new additions and add it to whatever is currently in your repository's history.

3.2 Starting with Git

Using the Git program directly (as opposed to using the `lily-git.tcl` GUI) allows you to have much greater control over the contributing process. You should consider using Git if you want to work on complex projects, or if you want to work on multiple projects concurrently.

3.2.1 Setting up

TODO: Remove this note if incorporating Windows instructions throughout this section:

> **Note:** These instructions assume that you are using the command-line version of Git 1.5 or higher. Windows users should skip to Section 3.5 [Git on Windows], page 27.

Installing Git

If you are using a Unix-based machine, the easiest way to download and install Git is through a package manager such as `rpm` or `apt-get`—the installation is generally automatic. The only required package is (usually) called `git-core`, although some of the auxiliary `git*` packages are also useful (such as `gitk`).

Alternatively, you can visit the Git website (`http://git-scm.com/`) for downloadable binaries and tarballs.

TODO: add Windows installation instructions (or `@ref{Git on Windows}`).

Initializing a repository

Once Git is installed, you'll need to create a new directory where your initial repository will be stored (the example below uses '`~/lilypond-git/`', where `~` represents your home directory). Run `git init` from within the new directory to initialize an empty repository:

```
mkdir ~/lilypond-git/; cd ~/lilypond-git/
git init
```

Technical details

This creates (within the '`~/lilypond-git/`' directory) a subdirectory called '`.git/`', which Git uses to keep track of changes to the repository, among other things. Normally you don't need to access it, but it's good to know it's there.

Configuring Git

> **Note:** Throughout the rest of this manual, all command-line input should be entered from the top directory of the Git repository being discussed (eg. '`~/lilypond-git/`'). This is referred to as the *top source directory*.

Before downloading a copy of the main LilyPond repository, you should configure some basic settings with the `git config` command. Git allows you to set both global and repository-specific options.

To configure settings that affect all repositories, use the `--global` command line option. For example, the first two options that you should always set are your *name* and *email*, since Git needs these to keep track of commit authors:

```
git config --global user.name "John Smith"
git config --global user.email john@example.com
```

To configure Git to use colored output where possible, use:

```
git config --global color.ui auto
```

The text editor that opens when using `git commit` can also be changed. If none of your editor-related environment variables are set ($GIT_EDITOR, $VISUAL, or $EDITOR), the default editor is usually `vi` or `vim`. If you're not familiar with either of these, you should

probably change the default to an editor that you know how to use. For example, to change the default editor to `nano`, enter:

```
git config --global core.editor nano
```

TODO: Add instructions for changing the editor on Windows, which is a little different, I think. -mp

Technical details

Git stores the information entered with `git config --global` in the file '`.gitconfig`', located in your home directory. This file can also be modified directly, without using `git config`. The '`.gitconfig`' file generated by the above commands would look like this:

```
[user]
        name = John Smith
        email = john@example.com
[color]
        ui = auto
[core]
        editor = nano
```

Using the `git config` command *without* the `--global` option configures repository-specific settings, which are stored in the file '`.git/config`'. This file is created when a repository is initialized (using `git init`), and by default contains these lines:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
```

However, since different repository-specific options are recommended for different development tasks, it is best to avoid setting any now. Specific recommendations will be mentioned later in this manual.

3.2.2 Downloading remote branches

Organization of remote branches

The main LilyPond repository is organized into *branches* to facilitate development. These are often called *remote* branches to distinguish them from *local* branches you might create yourself (see Section 3.3.3 [Using local branches], page 16).

The `master` branch contains all the source files used to build LilyPond, which includes the program itself (both stable and development releases), the documentation (and its translations), and the website. Generally, the `master` branch is expected to compile successfully.

The `lilypond/translation` branch is a side branch that allows translators to work without needing to worry about compilation problems. Periodically, the Translation Meister (after verifying that it doesn't break compilation), will *merge* this branch back into `master` to incorporate recent translations. Similarly, the `master` branch is usually merged into the `lilypond/translation` branch after significant changes to the English documentation. See Section 5.8 [Translating the documentation], page 59 for details.

LilyPond repository sources

The recommended source for downloading a copy of the main repository is:

```
git://git.sv.gnu.org/lilypond.git
```

However, if your internet router filters out connections using the GIT protocol, or if you experience difficulty connecting via GIT, you can try these other sources:

```
ssh://git.sv.gnu.org/srv/git/lilypond.git
http://git.sv.gnu.org/r/lilypond.git
```

The SSH protocol can only be used if your system is properly set up to use it. Also, the HTTP protocol is slowest, so it should only be used as a last resort.

Downloading individual branches

Once you have initialized an empty Git repository on your system (see [Initializing a repository], page 12), you can download a remote branch into it. Make sure you know which branch you want to start with.

To download the `master` branch, enter the following:

```
git remote add -ft master -m master \
  origin git://git.sv.gnu.org/lilypond.git/
```

To download the `lilypond/translation` branch, enter:

```
git remote add -ft lilypond/translation -m \
  lilypond/translation origin git://git.sv.gnu.org/lilypond.git/
```

The `git remote add` process could take up to ten minutes, depending on the speed of your connection. The output will be something like this:

```
Updating origin
remote: Counting objects: 235967, done.
remote: Compressing objects: 100% (42721/42721), done.
remote: Total 235967 (delta 195098), reused 233311 (delta 192772)
Receiving objects: 100% (235967/235967), 68.37 MiB | 479 KiB/s, done.
Resolving deltas: 100% (195098/195098), done.
From git://git.sv.gnu.org/lilypond
 * [new branch]      master     -> origin/master
From git://git.sv.gnu.org/lilypond
 * [new tag]         flower/1.0.1 -> flower/1.0.1
 * [new tag]         flower/1.0.10 -> flower/1.0.10

 * [new tag]         release/2.9.6 -> release/2.9.6
 * [new tag]         release/2.9.7 -> release/2.9.7
```

When `git remote add` is finished, the remote branch should be downloaded into your repository—though not yet in a form that you can use. In order to browse the source code files, you need to *create* and *checkout* your own local branch. In this case, however, it is easier to have Git create the branch automatically by using the `checkout` command on a non-existent branch. Enter the following:

```
git checkout -b branch origin/branch
```

where *`branch`* is the name of your tracking branch, either `master` or `lilypond/translation`.

Git will issue some warnings; this is normal:

```
warning: You appear to be on a branch yet to be born.
warning: Forcing checkout of origin/master.
Branch master set up to track remote branch master from origin.
Already on 'master'
```

By now the source files should be accessible—you should be able to edit any files in the '`lilypond-git/`' directory using a text editor of your choice. But don't start just yet! Before editing any source files, learn how to keep your changes organized and prevent problems later—read Section 3.3 [Basic Git procedures], page 15.

Technical Details

The `git remote add` command should add some lines to your local repository's '`.git/config`' file:

```
[remote "origin"]
        url = git://git.sv.gnu.org/lilypond.git/
        fetch = +refs/heads/master:refs/remotes/origin/master
```

Downloading all remote branches

To download all remote branches at once, you can `clone` the entire repository:

```
git clone git://git.sv.gnu.org/lilypond.git
```

Other branches

Most contributors will never need to touch the other branches. If you wish to do so, you will need more familiarity with Git; please see Section 3.7 [Other Git documentation], page 33.

- `dev/XYZ`: These branches are for individual developers. They store code which is not yet stable enough to be added to the `master` branch.
- `stable/XYZ`: The branches are kept for archival reasons.

Another item of interest might be the Grand Unified Builder, our cross-platform building tool. Since it is used by projects as well, it is not stored in our gub repository. For more info, see `http://lilypond.org/gub`. The git location is `http://github.com/janneke/gub`.

3.3 Basic Git procedures

3.3.1 The Git contributor's cycle

Here is a simplified view of the contribution process on Git:

1. Update your local repository by *pulling* the most recent updates from the remote repository.
2. Edit source files within your local repository's *working directory*.
3. *Commit* the changes you've made to a local *branch*.
4. Generate a *patch* to share your changes with the developers.

3.3.2 Pulling and rebasing

When developers push new patches to the `git.sv.gnu.org` repository, your local repository is **not** automatically updated. It is important to keep your repository up-to-date by periodically *pulling* the most recent *commits* from the remote branch. Developers expect patches to be as current as possible, since outdated patches require extra work before they can be used.

Occasionally you may need to rework some of your own modifications to match changes made to the remote branch (see Section 3.4.2 [Resolving conflicts], page 22), and it's considerably easier to rework things incrementally. If you don't update your repository along the way, you may have to spend a lot of time resolving branch conflicts and reconfiguring much of the work you've already done.

Fortunately, Git is able to resolve certain types of branch conflicts automatically with a process called *rebasing*. When rebasing, Git tries to modify your old commits so they appear as new commits (based on the latest updates). For a more involved explanation, see the `git-rebase` man page.

To pull without rebasing (recommended for translators), use the following command:

```
git pull    # recommended for translators
```

If you're tracking the remote `master` branch, you should add the `-r` option (short for `--rebase`) to keep commits on your local branch current:

```
git pull -r # use with caution when translating
```

If you don't edit translated documentation and don't want to type `-r` every time, configure the master branch to rebase by default with this command:

```
git config branch.master.rebase true
```

If pull fails because of a message like

```
error: Your local changes to 'Documentation/learning/tutorial.itely'
would be overwritten by merge.  Aborting.
```

or

```
Documentation/learning/tutorial.itely: needs update
refusing to pull with rebase: your working tree is not up-to-date
```

it means that you have modified some files in you working tree without committing changes (see Section 3.3.4 [Commits and patches], page 17); you can use the `git stash` command to work around this:

```
git stash      # save uncommitted changes
git pull -r    # pull using rebase (translators omit "-r")
git stash pop  # reapply previously saved changes
```

Note that `git stash pop` will try to apply a patch, and this may create a conflict. If this happens, see Section 3.4.2 [Resolving conflicts], page 22.

TODO: I think the next paragraph is confusing. Perhaps prepare the reader for new terms 'committish' and 'head'? -mp

> **Note:** translators and documentation editors, if you have changed committishes in the head of translated files using commits you have not yet pushed to `git.sv.gnu.org`, please do not rebase. If you want to avoid wondering whether you should rebase each time you pull, please always use committishes from master and/or lilypond/translation branch on `git.sv.gnu.org`, which in particular implies that you must push your changes to documentation except committishes updates (possibly after having rebased), then update the committishes and push them.

TODO: when committishes automatic conditional update have been tested and documented, append the following to the warning above: Note that using update-committishes make target generally touches committishes.

Technical details

The `git config` command mentioned above adds the line `rebase = true` to the master branch in your local repository's '`.git/config`' file:

```
[branch "master"]
        remote = origin
        merge = refs/heads/master
        rebase = true
```

3.3.3 Using local branches

Creating and removing branches

Local branches are useful when you're working on several different projects concurrently. To create a new branch, enter:

```
git branch name
```

To delete a branch, enter:

```
git branch -d name
```

Git will ask you for confirmation if it sees that data would be lost by deleting the branch. Use `-D` instead of `-d` to bypass this. Note that you cannot delete a branch if it is currently checked out.

Listing branches and remotes

You can get the exact path or URL of all remote branches by running:

```
git remote -v
```

To list Git branches on your local repositories, run

```
git branch     # list local branches only
git branch -r  # list remote branches
git branch -a  # list all branches
```

Checking out branches

To know the currently checked out branch, i.e. the branch whose source files are present in your working tree, read the first line of the output of

```
git status
```

The currently checked out branch is also marked with an asterisk in the output of `git branch`.

You can check out another branch *other_branch*, i.e. check out *other_branch* to the working tree, by running

```
git checkout other_branch
```

Note that it is possible to check out another branch while having uncommitted changes, but it is not recommended unless you know what you are doing; it is recommended to run `git status` to check this kind of issue before checking out another branch.

Merging branches

To merge branch *foo* into branch *bar*, i.e. to "add" all changes made in branch *foo* to branch *bar*, run

```
git checkout bar
git merge foo
```

If any conflict happens, see Section 3.4.2 [Resolving conflicts], page 22.

There are common usage cases for merging: as a translator, you will often want to merge `master` into `lilypond/translation`; on the other hand, the Translations meister wants to merge `lilypond/translation` into `master` whenever he has checked that `lilypond/translation` builds successfully.

3.3.4 Commits and patches

Understanding commits

Technically, a *commit* is a single point in the history of a branch, but most developers use the term to mean a *commit object*, which stores information about a particular revision. A single commit can record changes to multiple source files, and typically represents one logical set of related changes (such as a bug-fix). You can list the ten most recent commits in your current branch with this command:

```
git log -10 --oneline
```

If you're using an older version of Git and get an 'unrecognized argument' error, use this instead:

```
git log -10 --pretty=oneline --abbrev-commit
```

More interactive lists of the commits on the remote `master` branch are available at http://git.sv.gnu.org/gitweb/?p=lilypond.git;a=shortlog and http://git.sv.gnu.org/cgit/lilypond.git/log/.

Making commits

Once you have modified some source files in your working directory, you can make a commit with the following procedure:

1. Make sure you've configured Git properly (see [Configuring Git], page 12). Check that your changes meet the requirements described in Section 10.5 [Code style], page 95 and/or Section 5.4 [Documentation policy], page 54. For advanced edits, you may also want to verify that the changes don't break the compilation process.
2. Run the following command:

   ```
   git status
   ```

   to make sure you're on the right branch, and to see which files have been modified, added or removed, etc. You may need to tell Git about any files you've added by running one of these:

   ```
   git add file  # add untracked file individually
   git add .     # add all untracked files in current directory
   ```

   After `git add`, run `git status` again to make sure you got everything. You may also need to modify '`GNUmakefile`'.
3. Preview the changes about to be committed (to make sure everything looks right) with:

   ```
   git diff HEAD
   ```

   The `HEAD` argument refers to the most recent commit on the currently checked-out branch.
4. Generate the commit with:

   ```
   git commit -a
   ```

   The `-a` is short for `--all` which includes modified and deleted files, but only those newly created files that have previously been added.

Commit messages

When you run the `git commit -a` command, Git automatically opens the default text editor so you can enter a *commit message*. If you find yourself in a foreign editing environment, you're probably in `vi` or `vim`. If you want to switch to an editor you're more familiar with, quit by typing `:q!` and pressing `<Enter>`. See [Configuring Git], page 12 for instructions on changing the default editor.

In any case, Git will open a text file for your commit message that looks like this:

```
# Please enter the commit message for your changes.  Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   working.itexi
#
```

Your commit message should begin with a one-line summary describing the change (no more than 50 characters long), and if necessary a blank line followed by several lines giving the details:

```
Doc: add Baerenreiter and Henle solo cello suites

Added comparison of solo cello suite engravings to new essay with
high-res images, fixed cropping on Finale example.
```

Commit messages often start with a short prefix describing the general location of the changes. If a commit affects the documentation in English (or in several languages simultaneously) the commit message should be prefixed with "Doc: ". If the commit affects only one of the translations, the commit message should be prefixed with "Doc-**: ", where ** is the two-letter language code. Commits that affect the website should use "Web: " for English, and "Web-**: " for the other languages. Also, changes to a single file are often prefixed with the name of the file involved. Visit the links listed in [Understanding commits], page 17 for examples.

Making patches

If you want to share your changes with other contributors and developers, you need to generate *patches* from your commits. We prefer it if you follow the instructions in [Uploading a patch for review], page 19. However, we present an alternate method here.

You should always run `git pull -r` (translators should leave off the `-r`) before doing this to ensure that your patches are as current as possible.

Once you have made one or more commits in your local repository, and pulled the most recent commits from the remote branch, you can generate patches from your local commits with the command:

```
git format-patch origin
```

The `origin` argument refers to the remote tracking branch at `git.sv.gnu.org`. This command generates a separate patch for each commit that's in the current branch but not in the remote branch. Patches are placed in the current working directory and will have names that look something like this:

```
0001-Doc-Fix-typos.patch
0002-Web-Remove-dead-links.patch
```

Send an email (must be less than 64 KB) to `lilypond-devel@gnu.org` briefly explaining your work, with the patch files attached. Translators should send patches to `translations@lilynet.net`. After your patches are reviewed, the developers may push one or more of them to the main repository or discuss them with you.

Uploading a patch for review

Any non-trivial change should be uploaded to our "Rietveld" code review website:

`http://codereview.appspot.com/`

`git-cl` install

LilyDev users should skip over these 'install' instructions.

1. Install `git-cl` by entering:

   ```
   git clone git://neugierig.org/git-cl.git
   ```

2. Add the '`git-cl/`' directory to your PATH, or create a symbolic link to the `git-cl` and `upload.py` scripts in one of your PATH directories (such as '`$HOME/bin`').

`git-cl` configuration

LilyDev users should perform these 'configuration' instructions.

1. You must have a gmail account; please create one if you do not have one already.

2. Move into the top source directory and then configure `git cl` with the following commands. If you do not understand any question, just answer with a newline (CR).

   ```
   cd $HOME/lilypond-git/
   git cl config
   ```

   The "CC list" question should be answered with:

   ```
   lilypond-devel@gnu.org
   ```

Uploading patch set

> **Note:** Unless you are familiar with branches, only work on one set of changes at once.

There are two methods, depending on your git setup.

- **Master branch**: (easy option, and used in `lily-git.tcl`)

  If you added your patch to `master`, then:

  ```
  git pull -r
  git cl upload origin/master
  ```

  If you have git push ability, make sure that you *remove* your patch (with `git rebase` or `git reset`) before pushing other stuff.

- **Separate branch**: (complicated option)

  Ensure your changes are committed in a separate branch, which should differ from the reference branch to be used by just the changes to be uploaded. If the reference branch is to be origin/master, ensure this is up-to-date. If necessary, use git rebase to rebase the branch containing the changes to the head of origin/master. Finally, check out branch with the changes and enter the command:

  ```
  git cl upload <reference SHA1 ID>
  ```

  where <reference SHA1 ID> is the SHA1 ID of the commit to be used as a reference source for the patch. Generally, this will be the SHA1 ID of origin/master, and in that case the command:

  ```
  git cl upload origin/master
  ```

  can be used.

After prompting for your Google email address and password, the patch set will be posted to Rietveld, and you will be given a URL for your patch.

> **Note:** Some installations of git-cl fail when uploading a patch set that includes a .scm file. When this happens, it can generally be fixed by editing the file '`/etc/mime.types`'. Add a line to this file containing `text/x-script.scheme scm`.

Announcing your patch set

You should then announce the patch by logging into the code review issue webpage and using "Publish + Mail Comments" to add a (mostly bogus) comment to your issue. The text of your comment will be sent to our developer mailing list.

> **Note:** There is no automatic notification of a new patch; you must add a comment yourself.

Revisions

As revisions are made in response to comments, successive patch sets for the same issue can be uploaded by reissuing the git-cl command with the modified branch checked out.

Sometimes in response to comments on revisions, the best way to work may require creation of a new branch in git. In order to associate the new branch with an existing Rietveld issue, the following command can be used:

```
git cl issue issue-number
```

where `issue-number` is the number of the existing Rietveld issue.

Resetting git cl

If `git cl` becomes confused, you can "reset" it by running:

```
git cl issue 0
```

3.4 Advanced Git procedures

> **Note:** This section is not necessary for normal contributors; these commands are presented for information for people interested in learning more about git.

It is possible to work with several branches on the same local Git repository; this is especially useful for translators who may have to deal with both `lilypond/translation` and a stable branch, e.g. `stable/2.12`.

Some Git commands are introduced first, then a workflow with several Git branches of LilyPond source code is presented.

3.4.1 Advanced Git concepts

A bit of Git vocabulary will be explained below. The following is only introductory; for a better understanding of Git concepts, you may wish to read Section 3.7 [Other Git documentation], page 33.

The `git pull origin` command above is just a shortcut for this command:

```
git pull git://git.sv.gnu.org/lilypond.git/ branch:origin/branch
```

where *`branch`* is typically `master` or `lilypond/translation`; if you do not know or remember, see Section 3.2.2 [Downloading remote branches], page 13 to remember which commands you issued or which source code you wanted to get.

A *commit* is a set of changes made to the sources; it also includes the committish of the parent commit, the name and e-mail of the *author* (the person who wrote the changes), the name and e-mail of the *committer* (the person who brings these changes into the Git repository), and a commit message.

A *committish* is the SHA1 checksum of a commit, a number made of 40 hexadecimal digits, which acts as the internal unique identifier for this commit. To refer to a particular revision, don't use vague references like the (approximative) date, simply copy and paste the committish.

A *branch* is nothing more than a pointer to a particular commit, which is called the *head* of the branch; when referring to a branch, one often actually thinks about its head and the ancestor commits of the head.

Now we will explain the two last commands you used to get the source code from Git—see [Downloading individual branches], page 14.

```
git remote add -ft branch -m branch \
  origin git://git.sv.gnu.org/lilypond.git/
```

```
git checkout -b branch origin/branch
```

The `git remote` has created a branch called `origin/branch` in your local Git repository. As this branch is a copy of the remote branch web from git.sv.gnu.org LilyPond repository, it is called a *remote branch*, and is meant to track the changes on the branch from git.sv.gnu.org: it will be updated every time you run `git pull origin` or `git fetch origin`.

The `git checkout` command has created a branch named `branch`. At the beginning, this branch is identical to `origin/branch`, but it will differ as soon as you make changes, e.g. adding newly translated pages or editing some documentation or code source file. Whenever you pull, you merge the changes from `origin/branch` and `branch` since the last pulling. If you do not have push (i.e. "write") access on git.sv.gnu.org, your `branch` will always differ from `origin/branch`. In this case, remember that other people working like you with the remote branch `branch` of git://git.sv.gnu.org/lilypond.git/ (called `origin/branch` on your local repository) know nothing about your own `branch`: this means that whenever you use a committish or make a patch, others expect you to take the latest commit of `origin/branch` as a reference.

Finally, please remember to read the man page of every Git command you will find in this manual in case you want to discover alternate methods or just understand how it works.

3.4.2 Resolving conflicts

Occasionally an update may result in conflicts – this happens when you and somebody else have modified the same part of the same file and git cannot figure out how to merge the two versions together. When this happens, you must manually merge the two versions.

If you need some documentation to understand and resolve conflicts, see paragraphs *How conflicts are presented* and *How to resolve conflicts* in `git merge` man page.

If all else fails, you can follow the instructions in Section 3.4.3 [Reverting all local changes], page 22. Be aware that this eliminates any changes you have made!

3.4.3 Reverting all local changes

Sometimes git will become hopelessly confused, and you just want to get back to a known, stable state. This command destroys any local changes you have made, but at least you get back to the current online version:

```
git reset --hard origin/master
```

3.4.4 Working with remote branches

Fetching new branches from git.sv.gnu.org

To fetch and check out a new branch named `branch` on git.sv.gnu.org, run from top of the Git repository

```
git config --add remote.origin.fetch \
  +refs/heads/branch:refs/remotes/origin/branch

git checkout --track -b branch origin/branch
```

After this, you can pull `branch` from git.sv.gnu.org with:

```
git pull
```

Note that this command generally fetches all branches you added with `git remote add` (when you initialized the repository) or `git config --add`, i.e. it updates all remote branches from remote `origin`, then it merges the remote branch tracked by the current branch into the current branch. For example, if your current branch is `master`, `origin/master` will be merged into `master`.

Local clones, or having several working trees

If you play with several Git branches, e.g. `master`, `lilypond/translation`, `stable/2.12`), you may want to have one source and build tree for each branch; this is possible with subdirectories of your local Git repository, used as local cloned subrepositories. To create a local clone for the branch named *`branch`*, run

```
git checkout branch
git clone -lsn . subdir
cd subdir
git reset --hard
```

Note that *`subdir`* must be a directory name which does not already exist. In *`subdir`*, you can use all Git commands to browse revisions history, commit and uncommit changes; to update the cloned subrepository with changes made on the main repository, cd into *`subdir`* and run `git pull`; to send changes made on the subrepository back to the main repository, run `git push` from *`subdir`*. Note that only one branch (the currently checked out branch) is created in the subrepository by default; it is possible to have several branches in a subrepository and do usual operations (checkout, merge, create, delete...) on these branches, but this possibility is not detailed here.

When you push *`branch`* from *`subdir`* to the main repository, and *`branch`* is checked out in the main repository, you must save uncommitted changes (see `git stash`) and do `git reset --hard` in the main repository in order to apply pushed changes in the working tree of the main repository.

3.4.5 Git log

The commands above don't only bring you the latest version of the sources, but also the full history of revisions (revisions, also called commits, are changes made to the sources), stored in the '`.git`' directory. You can browse this history with

```
git log     # only shows the logs (author, committish and commit message)
git log -p  # also shows diffs
gitk        # shows history graphically
```

> **Note:** The `gitk` command may require a separate `gitk` package, available in the appropriate distribution's repositories.

3.4.6 Applying remote patches

TODO: Explain how to determine if a patch was created with `git format-patch`.

Well-formed git patches created with `git format-patch` should be committed with the following command:

```
git am patch
```

Patches created without `git format-patch` can be applied in two steps. The first step is to apply the patch to the working tree:

```
git apply patch
```

The second step is to commit the changes and give credit to the author of the patch. This can be done with the following command:

```
git commit -a --author="John Smith <john@example.com>"
```

3.4.7 Sending and receiving patches via email

The default `x-diff` MIME type associated with patch files (i.e., files whose name ends in `.patch`) means that the encoding of line endings may be changed from UNIX to DOS format when they

are sent as attachments. Attempting to apply such an inadvertently altered patch will cause git to fail with a message about 'whitespace errors'.

The solution to such problems is surprisingly simple—just change the default file extension of patches generated by git to end in `.txt`, for example:

```
git config format.suffix '.patch.txt'
```

This should cause email programs to apply the correct base64 encoding to attached patches.

If you receive a patch with DOS instead of UNIX line-endings, it can be converted back using the `dos2unix` utility.

Lots of useful information on email complications with patches is provided on the Wine wiki at http://wiki.winehq.org/GitWine.

3.4.8 Cleaning up multiple patches

If you have been developing on your own branch for a while, you may have more commmits than is really sensible. To revise your work and condense commits, use:

```
git rebase origin/master
git rebase -i origin/master
```

> **Note:** Be a bit cautious – if you completely remove commits during the interactive session, you will... err... completely remove those commits.

3.4.9 Commit access

Most contributors are not able to commit patches directly to the main repository—only members of the LilyPond development team have *commit access*. If you are a contributor and are interested in joining the development team, contact the Project Manager through the mailing list (lilypond-devel@gnu.org). Generally, only contributors who have already provided a number of patches which have been pushed to the main repository will be considered for membership.

If you have been approved by the Project Manager, use the following procedure to obtain commit access:

1. If you don't already have one, set up a Savannah user account at https://savannah.gnu.org/account/register.php. If your web browser responds with an "untrusted connection" message when you visit the link, follow the steps for including the CAcert root certificate in your browser, given at http://savannah.gnu.org/tls/tutorial/.

   > **Note:** Savannah will silently put your username in lower-case – do not try to use capital letters.

2. After registering, if you are not logged in automatically, login at https://savannah.gnu.org/account/login.php—this should take you to your "my" page (https://savannah.gnu.org/my/).
3. Click on the "My Groups" link to access the "My Group Membership" page. From there, find the "Request for Inclusion" box and search for "LilyPond". Among the search results, check the box labeled "GNU LilyPond Music Typesetter" and write a brief (required) message for the Project Manager ("Hey it's me!" should be fine).

   Note that you will not have commit access until the Project Manager activates your membership. Once your membership is activated, LilyPond should appear under the heading "Groups I'm Contributor of" on your "My Group Membership" page.
4. Generate an SSH 'dsa' key pair. Enter the following at the command prompt:

```
ssh-keygen -t dsa
```

When prompted for a location to save the key, press <ENTER> to accept the default location ('`~/.ssh/id_dsa`').

Next you are asked to enter an optional passphrase. On most systems, if you use a passphrase, you will likely be prompted for it every time you use `git push` or `git pull`. You may prefer this since it can protect you from your own mistakes (like pushing when you mean to pull), though you may find it tedious to keep re-entering it.

You can change/enable/disable your passphrase at any time with:

```
ssh-keygen -f ~/.ssh/id_dsa -p
```

Note that the GNOME desktop has a feature which stores your passphrase for you for an entire GNOME session. If you use a passphrase to "protect you from yourself", you will want to disable this feature, since you'll only be prompted once. Run the following command, then logout of GNOME and log back in:

```
gconftool-2 --set -t bool \
  /apps/gnome-keyring/daemon-components/ssh false
```

After setting up your passphrase, your private key is saved as '`~/.ssh/id_dsa`' and your public key is saved as '`~/.ssh/id_dsa.pub`'.

5. Register your public SSH 'dsa' key with Savannah. From the "My Account Configuration" page, click on "Edit SSH Keys", then paste the contents of your '`~/.ssh/id_dsa.pub`' file into one of the "Authorized keys" text fields, and click "Update".

   Savannah should respond with something like:

   ```
   Success: Key #1 seen Keys registered
   ```

6. Configure Git to use the SSH protocol (instead of the GIT protocol). From your local Git repository, enter:

   ```
   git config remote.origin.url \
     ssh://user@git.sv.gnu.org/srv/git/lilypond.git
   ```

   where *user* is your username on Savannah.

7. After your membership has been activated and you've configured Git to use SSH, test the connection with:

   ```
   git pull --verbose
   ```

   SSH should issue the following warning:

   ```
   The authenticity of host 'git.sv.gnu.org (140.186.70.72)' can't
   be established.
   RSA key fingerprint is
   80:5a:b0:0c:ec:93:66:29:49:7e:04:2b:fd:ba:2c:d5.
   Are you sure you want to continue connecting (yes/no)?
   ```

   Make sure the RSA key fingerprint displayed matches the one above. If it doesn't, respond "no" and check that you configured Git properly in the previous step. If it does match, respond "yes". SSH should then issue another warning:

   ```
   Warning: Permanently added 'git.sv.gnu.org,140.186.70.72' (RSA) to
   the list of known hosts.
   ```

   The list of known hosts is stored in the file '`~/.ssh/known_hosts`'.

   At this point, you are prompted for your passphrase if you have one, then Git will attempt a pull.

   If `git pull --verbose` fails, you should see error messages like these:

   ```
   Permission denied (publickey).
   fatal: The remote end hung up unexpectedly
   ```

If you get the above error, you may have made a mistake when registering your SSH key at Savannah. If the key is properly registered, you probably just need to wait for the Savannah server to activate it. It usually takes a few minutes for the key to be active after registering it, but if it still doesn't work after an hour, ask for help on the mailing list.

If `git pull --verbose` succeeds, the output will include a 'From' line that shows 'ssh' as the protocol:

```
From ssh://user@git.sv.gnu.org/srv/git/lilypond
```

If the protocol shown is not 'ssh', check that you configured Git properly in the previous step.

8. Test your commit access with a dry run:

```
git push --dry-run --verbose
```

Note that recent versions of Git (Git 1.6.3 or later) will issue a big warning if the above command is used. The simplest solution is to tell Git to push all matching branches by default:

```
git config push.default matching
```

Then `git push` should work as before. For more details, consult the `git push` man page.

Technical details

- On Firefox, to view or remove the CAcert root certificate, go to: Edit > Preferences > Advanced > Encryption > View Certificates > Authorities > Certificate Name > Root CA > CA Cert Signing Authority.
- The `git config` commands above should modify your local repository's '`.git/config`' file. These lines:

```
[remote "origin"]
        url = git://git.sv.gnu.org/lilypond.git/
```

should now be changed to:

```
[remote "origin"]
        url = ssh://user@git.sv.gnu.org/srv/git/lilypond.git
```

where *user* is your login name on Savannah.

- Similarly, the `git config push.default matching` command should add these lines to '`.git/config`':

```
[push]
        default = matching
```

Known issues and warnings

Encryption protocols, including ssh, generally do not permit packet fragmentation to avoid introducing a point of insecurity. This means that the maximum packet size must not exceed the smallest MTU (Maximum Transmission Unit) set in the routers along the path. This smallest MTU is determined by a procedure during call set-up which relies on the transmission over the path of ICMP packets. If any of the routers in the path block ICMP packets this mechanism fails, resulting in the possibility of packets being transmitted which exceed the MTU of one of the routers. If this happens the packet is discarded, causing the ssh session to hang, timeout or terminate with the error message

```
ssh: connect to host <host ip addr> port 22: Bad file number
fatal: The remote end hung up unexpectedly
```

depending on precisely when in the proceedings the first large packet is transmitted. Most routers on the internet have MTU set to 1500, but routers installed in homes to connect via broadband may use a slightly smaller MTU for efficient transmission over ATM. If this problem is encountered a possible work-around is to set the MTU in the local router to 1500.

3.5 Git on Windows

TODO: Decide what to do with this... Pare it down? Move paragraphs next to analogous Unix instructions? -mp

3.5.1 Background to nomenclature

Git is a system for tracking the changes made to source files by a distributed set of editors. It is designed to work without a master repository, but we have chosen to have a master repository for LilyPond files. Editors hold a local copy of the master repository together with any changes they have made locally. Local changes are held in a local 'branch', of which there may be several, but these instructions assume you are using just one. The files visible in the local repository always correspond to those on the currently 'checked out' local branch.

Files are edited on a local branch, and in that state the changes are said to be 'unstaged'. When editing is complete, the changes are moved to being 'staged for commit', and finally the changes are 'committed' to the local branch. Once committed, the changes (called a 'commit') are given a unique 40-digit hexadecimal reference number called the 'Committish' or 'SHA1 ID' which identifies the commit to Git. Such committed changes can be sent to the master repository by 'pushing' them (if you have write permission) or by sending them by email to someone who has, either as a complete file or as a 'diff' or 'patch' (which send just the differences from the master repository).

3.5.2 Installing git

Obtain Git from http://code.google.com/p/msysgit/downloads/list (note, not msysGit, which is for Git developers and not PortableGit, which is not a full git installation) and install it.

Note that most users will not need to install SSH. That is not required until you have been granted direct push permissions to the master git repository.

Start Git by clicking on the desktop icon. This will bring up a command line bash shell. This may be unfamiliar to Windows users. If so, follow these instructions carefully. Commands are entered at a $ prompt and are terminated by keying a newline.

3.5.3 Initialising Git

Decide where you wish to place your local Git repository, creating the folders in Windows as necessary. Here we call the folder to contain the repository `[path]/Git`, but if you intend using Git for other projects a directory name like `lilypond-git` might be better. You will need to have space for around 100Mbytes.

Start the Git bash shell by clicking on the desk-top icon installed with Git and type

```
cd [path]/Git
```

to position the shell at your new Git repository.

Note: if [path] contains folders with names containing spaces use

```
cd "[path]/Git"
```

Then type

```
git init
```

to initialize your Git repository.

Then type (all on one line; the shell will wrap automatically)

```
git remote add -ft master origin git://git.sv.gnu.org/lilypond.git
```

to download the lilypond master files.

> **Note:** Be patient! Even on a broadband connection this can take 10 minutes or more. Wait for lots of [new tag] messages and the $ prompt.

We now need to generate a local copy of the downloaded files in a new local branch. Your local branch needs to have a name. It is usual to call it 'master' and we shall do that here.

To do this, type

```
git checkout -b master origin/master
```

This creates a second branch called 'master'. You will see two warnings (ignore these), and a message advising you that your local branch 'master' has been set up to track the remote branch. You now have two branches, a local branch called 'master', and a tracking branch called 'origin/master', which is a shortened form of 'remotes/origin/master'.

Return to Windows Explorer and look in your Git repository. You should see lots of folders. For example, the LilyPond documentation can be found in [path]/Git/Documentation/.

The Git bash shell is terminated by typing `exit` or by clicking on the usual Windows close-window widget.

3.5.4 Git GUI

Almost all subsequent work will use the Git Graphical User Interface, which avoids having to type command line commands. To start Git GUI first start the Git bash shell by clicking on the desktop icon, and type

```
cd [path]/Git
git gui
```

The Git GUI will open in a new window. It contains four panels and 7 pull-down menus. At this stage do not use any of the commands under Branch, Commit, Merge or Remote. These will be explained later.

The top panel on the left contains the names of files which you are in the process of editing (Unstaged Changes), and the lower panel on the left contains the names of files you have finished editing and have staged ready for committing (Staged Changes). At present, these panels will be empty as you have not yet made any changes to any file. After a file has been edited and saved the top panel on the right will display the differences between the edited file selected in one of the panels on the left and the last version committed on the current branch.

The panel at bottom right is used to enter a descriptive message about the change before committing it.

The Git GUI is terminated by entering CNTL-Q while it is the active window or by clicking on the usual Windows close-window widget.

3.5.5 Personalising your local git repository

Open the Git GUI, click on

```
Edit -> Options
```

and enter your name and email address in the left-hand (Git Repository) panel. Leave everything else unchanged and save it.

Note that Windows users must leave the default setting for line endings unchanged. All files in a git repository must have lines terminated by just a LF, as this is required for Merge to work, but Windows files are terminated by CRLF by default. The git default setting causes the line endings of files in a Windows git repository to be flipped automatically between LF and CRLF as required. This enables files to be edited by any Windows editor without causing problems in the git repository.

3.5.6 Checking out a branch

At this stage you have two branches in your local repository, both identical. To see them click on

```
Branch -> Checkout
```

You should have one local branch called 'master' and one tracking branch called 'origin/master'. The latter is your local copy of the 'remotes/origin/master' branch in the master LilyPond repository. The local 'master' branch is where you will make your local changes.

When a particular branch is selected, i.e., checked out, the files visible in your repository are changed to reflect the state of the files on that branch.

3.5.7 Updating files from 'remote/origin/master'

Before starting the editing of a file, ensure your local repository contains the latest version of the files in the remote repository by first clicking

```
Remote -> Fetch from -> origin
```

in the Git GUI.

This will place the latest version of every file, including all the changes made by others, into the 'origin/master' branch of the tracking branches in your git repository. You can see these files by checking out this branch, but you must *never* edit any files while this branch is checked out. Check out your local 'master' branch again.

You then need to merge these fetched files into your local 'master' branch by clicking on

```
Merge -> Local Merge
```

and if necessary select the local 'master' branch.

Note that a merge cannot be completed if you have made any local changes which have not yet been committed.

This merge will update all the files in the 'master' branch to reflect the current state of the 'origin/master' branch. If any of the changes conflict with changes you have made yourself recently you will be notified of the conflict (see below).

3.5.8 Editing files

First ensure your 'master' branch is checked out, then simply edit the files in your local Git repository with your favourite editor and save them back there. If any file contains non-ASCII characters ensure you save it in UTF-8 format. Git will detect any changes whenever you restart Git GUI and the file names will then be listed in the Unstaged Changes panel. Or you can click the Rescan button to refresh the panel contents at any time. You may break off and resume editing any time.

The changes you have made may be displayed in diff form in the top right-hand panel of Git GUI by clicking on the file name shown in one of the left panels.

When your editing is complete, move the files from being Unstaged to Staged by clicking the document symbol to the left of each name. If you change your mind it can be moved back by clicking on the ticked box to the left of the name.

Finally the changes you have made may be committed to your 'master' branch by entering a brief message in the Commit Message box and clicking the Commit button.

If you wish to amend your changes after a commit has been made, the original version and the changes you made in that commit may be recovered by selecting

```
Commit -> Amend Last Commit
```

or by checking the Amend Last Commit radio button at bottom right. This will return the changes to the Staged state, so further editing made be carried out within that commit. This must only be done *before* the changes have been Pushed or sent to your mentor for Pushing - after that it is too late and corrections have to be made as a separate commit.

3.5.9 Sending changes to 'remotes/origin/master'

If you do not have write access to 'remotes/origin/master' you will need to send your changes by email to someone who does.

First you need to create a diff or patch file containing your changes. To create this, the file must first be committed. Then terminate the Git GUI. In the git bash shell first cd to your Git repository with

```
cd [path]/Git
```

if necessary, then produce the patch with

```
git format-patch origin
```

This will create a patch file for all the locally committed files which differ from 'origin/master'. The patch file can be found in [path]/Git and will have a name formed from the commit message.

3.5.10 Resolving merge conflicts

As soon as you have committed a changed file your local `master` branch has diverged from `origin/master`, and will remain diverged until your changes have been committed in `remotes/origin/master` and Fetched back into your `origin/master` branch. Similarly, if a new commit has been made to `remotes/origin/master` by someone else and Fetched, your local `master` branch is divergent. You can detect a divergent branch by clicking on

```
Repository -> Visualise all branch history
```

This opens up a very useful new window called 'gitk'. Use this to browse all the commits made by yourself and others.

If the diagram at top left of the resulting window does not show your `master` tag on the same node as the `remotes/origin/master` tag your branch has diverged from `origin/master`. This is quite normal if files you have modified yourself have not yet been Pushed to `remotes/origin/master` and Fetched, or if files modified and committed by others have been Fetched since you last Merged `origin/master` into your local `master` branch.

If a file being merged from `origin/master` differs from one you have modified in a way that cannot be resolved automatically by git, Merge will report a Conflict which you must resolve by editing the file to create the version you wish to keep.

This could happen if the person updating `remotes/origin/master` for you has added some changes of his own before committing your changes to `remotes/origin/master`, or if someone else has changed the same file since you last fetched the file from `remotes/origin/master`.

Open the file in your editor and look for sections which are delimited with ...

[to be completed when I next have a merge conflict to be sure I give the right instructions -td]

3.5.11 Other actions

The instructions above describe the simplest way of using git on Windows. Other git facilities which may usefully supplement these include

- Using multiple local branches (Create, Rename, Delete)
- Resetting branches
- Cherry-picking commits
- Pushing commits to remote/origin/master
- Using gitk to review history

Once familiarity with using git on Windows has been gained the standard git manuals can be used to learn about these.

3.6 Repository directory structure

```
Prebuilt Documentation and packages are available from:

    http://www.lilypond.org

LilyPond development is hosted at:

    http://savannah.gnu.org/projects/lilypond

Here is a simple explanation of the directory layout for
LilyPond's source files.


.                                                 Toplevel READMEs, ChangeLog,
|                          build bootstrapping, patches
|                          for third party programs
|
|-- Documentation/       Top sources for manuals
|   |
|   |
|   |   INDIVIDUAL CHAPTERS FOR EACH MANUAL:
|   |
|   |-- contributor/     Contributor's Guide
|   |-- essay/           Essay on automated music engraving
|   |-- extending/       Extending
|   |-- learning/        Learning Manual
|   |-- notation/        Notation Reference
|   |-- usage/           Usage
|   |-- web/             The website
|   |   `-- ly-examples/ .ly files for the "Examples" page
|   |
|   |
|   |   TRANSLATED MANUALS:
|   |     Each language's directory can contain...
|   |       1) translated versions of:
|   |          * top sources for manuals
|   |          * individual chapters for each manual
|   |       2) a texidocs/ directory for snippet translations
|   |
|   |-- de/              German
|   |-- es/              Spanish
|   |-- fr/              French
|   |-- hu/              Hungarian
|   |-- it/              Italian
|   |-- ja/              Japanese
|   |-- nl/              Dutch
|   |
|   |
|   |   MISCELLANEOUS DOC STUFF:
|   |
|   |-- css/             CSS files for HTML docs
```

```
|   |-- included/        .ly files used in the manuals
|   |-- logo/            Web logo and "note" icon
|   |-- misc/            Old announcements, ChangeLogs and NEWS
|   |-- pictures/        Images used (eps/jpg/png/svg)
|   |   `-- pdf/         (pdf)
|   |-- po/              Translated build/maintenance scripts
|   |-- snippets/        Auto-generated .ly snippets (from the LSR)
|   |   `-- new/         Snippets too new for the LSR
|   `-- topdocs/         AUTHORS, INSTALL, README
|
|
|   C++ SOURCES:
|
|-- flower/              A simple C++ library
|-- lily/                C++ sources for the LilyPond binary
|
|
|   LIBRARIES:
|
|-- ly/                  .ly \include files
|-- mf/                  MetaFont sources for Emmentaler fonts
|-- ps/                  PostScript library files
|-- scm/                 Scheme sources for LilyPond and subroutine files
|-- tex/                 TeX and texinfo library files
|
|
|   SCRIPTS:
|
|-- python/              Python modules, MIDI module
|   `-- auxiliar/        Python modules for build/maintenance
|-- scripts/             End-user scripts (--> lilypond/usr/bin/)
|   |-- auxiliar/        Maintenance and non-essential build scripts
|   `-- build/           Essential build scripts
|
|
|   BUILD PROCESS:
|   (also see SCRIPTS section above)
|
|-- make/                Specific make subroutine files
|-- stepmake/            Generic make subroutine files
|
|
|   REGRESSION TESTS:
|
|-- input/
|   `-- regression/      .ly regression tests
|       |-- abc2ly/      .abc regression tests
|       `-- musicxml/    .xml and .itexi regression tests
|
|
|   MISCELLANEOUS:
|
```

```
|-- elisp/                Emacs LilyPond mode and syntax coloring
|-- vim/                  Vi(M) LilyPond mode and syntax coloring
`-- po/                   Translations for binaries and end-user scripts
```

3.7 Other Git documentation

- Official git man pages: http://www.kernel.org/pub/software/scm/git/docs/
- More in-depth tutorials: http://git-scm.com/documentation
- Book about git: Pro Git

4 Compiling

This chapter describes the process of compiling the LilyPond program from source files.

4.1 Overview of compiling

Compiling LilyPond from source is an involved process, and is only recommended for developers and packagers. Typical program users are instead encouraged to obtain the program from a package manager (on Unix) or by downloading a precompiled binary configured for a specific operating system. Pre-compiled binaries are available on the Section "Download" in *General Information* page.

Compiling LilyPond from source is necessary if you want to build, install, or test your own version of the program.

A successful compile can also be used to generate and install the documentation, incorporating any changes you may have made. However, a successful compile is not a requirement for generating the documentation. The documentation can be built using a Git repository in conjunction with a locally installed copy of the program. For more information, see [Building documentation without compiling], page 41.

Attempts to compile LilyPond natively on Windows have been unsuccessful, though a workaround is available (see Section "Lilydev" in *Contributor's Guide*).

4.2 Requirements

4.2.1 Requirements for running LilyPond

Running LilyPond requires proper installation of the following software:

- DejaVu fonts (normally installed by default)
- FontConfig (2.4.0 or newer)
- Freetype (2.1.10 or newer)
- Ghostscript (8.60 or newer)
- Guile (1.8.2 or newer)
- Pango (1.12 or newer)
- Python (2.4 or newer)

International fonts are required to create music with international text or lyrics.

4.2.2 Requirements for compiling LilyPond

Below is a full list of packages needed to build LilyPond. However, for most common distributions there is an easy way of installing most all build dependencies in one go:

Distribution	**Command**
Debian, Ubuntu	`sudo apt-get build-dep lilypond`
Fedora, RHEL	`sudo yum-builddep lilypond`
openSUSE, SLED	`sudo zypper --build-deps-only source-install lilypond`

- Everything listed in Section 4.2.1 [Requirements for running LilyPond], page 34
- Development packages for the above items (which should include header files and libraries).

  Red Hat Fedora:

```
guile-devel-version
fontconfig-devel-version
freetype-devel-version
pango-devel-version
python-devel-version
```

Debian GNU/Linux:

```
guile-version-dev
libfontconfig1-dev
libfreetype6-dev
libpango1.0-dev
pythonversion-dev
```

- Flex
- FontForge (20060125 or newer; 20100501 or newer is recommended; must be compiled with '`--enable-double`'. Failure to do so can lead to poor intersection calculations and poorly-rendered glyphs.)
- GNU Bison
- GNU Compiler Collection (3.4 or newer, 4.*x* recommended)
- GNU gettext (0.17 or newer)
- GNU Make (3.78 or newer)
- MetaFont (mf-nowin, mf, mfw or mfont binaries), usually packaged with TEX.
- MetaPost (mpost binary), usually packaged with TEX.
- Perl
- Texinfo (4.11 or newer)
- Type 1 utilities (1.33 or newer recommended)

4.2.3 Requirements for building documentation

You can view the documentation online at `http://www.lilypond.org/doc/`, but you can also build it locally. This process requires some additional tools and packages:

- Everything listed in Section 4.2.2 [Requirements for compiling LilyPond], page 34
- ImageMagick
- Netpbm
- gzip
- rsync
- Texi2HTML (1.82)
- International fonts

  Red Hat Fedora:

```
fonts-arabic
fonts-hebrew
fonts-ja
fonts-xorg-truetype
taipeifonts
ttfonts-ja
ttfonts-zh_CN
```

  Debian GNU/Linux:

```
emacs-intl-fonts
ttf-kochi-gothic
```

```
ttf-kochi-mincho
xfonts-bolkhov-75dpi
xfonts-cronyx-75dpi
xfonts-cronyx-100dpi
xfonts-intl-.*
```

4.3 Getting the source code

Downloading the Git repository

In general, developers compile LilyPond from within a local Git repository. Setting up a local Git repository is explained in Section "Starting with Git" in *Contributor's Guide*.

Downloading a source tarball

Packagers are encouraged to use source tarballs for compiling.

The tarball for the latest stable release is available on the Section "Source" in *General Information* page.

The latest source code snapshot is also available as a tarball from the GNU Savannah Git server.

All tagged releases (including legacy stable versions and the most recent development release) are available here:

`http://download.linuxaudio.org/lilypond/source/`

Download the tarball to your '`~/src/`' directory, or some other appropriate place.

> **Note:** Be careful where you unpack the tarball! Any subdirectories of the current folder named '`lilypond/`' or '`lilypond-x.y.z/`' (where *x.y.z* is the release number) will be overwritten if there is a name clash with the tarball.

Unpack the tarball with this command:

```
tar -xzf lilypond-x.y.z.tar.gz
```

This creates a subdirectory within the current directory called '`lilypond-x.y.z/`'. Once unpacked, the source files occupy about 40 MB of disk space.

Windows users wanting to look at the source code may have to download and install the free-software 7zip archiver to extract the tarball.

4.4 Configuring `make`

4.4.1 Running `./autogen.sh`

After you unpack the tarball (or download the Git repository), the contents of your top source directory should be similar to the current source tree listed at `http://git.sv.gnu.org/gitweb/?p=lilypond.git;a=tree`.

Next, you need to create the generated files; enter the following command from your top source directory:

```
./autogen.sh --noconfigure
```

This will generate a number of files and directories to aid configuration, such as '`configure`', '`README.txt`', etc.

Next, create the build directory with:

```
mkdir build/
cd build/
```

We heavily recommend building lilypond inside a separate directory with this method.

4.4.2 Running `../configure`

Configuration options

> **Note:** make sure that you are in the '`build/`' subdirectory of your source tree.

The `../configure` command (generated by `./autogen.sh`) provides many options for configuring `make`. To see them all, run:

```
../configure --help
```

Checking build dependencies

> **Note:** make sure that you are in the '`build/`' subdirectory of your source tree.

When `../configure` is run without any arguments, it will check to make sure your system has everything required for compilation:

```
../configure
```

If any build dependency is missing, `../configure` will return with:

```
ERROR: Please install required programs:  foo
```

The following message is issued if you are missing programs that are only needed for building the documentation:

```
WARNING: Please consider installing optional programs:  bar
```

If you intend to build the documentation locally, you will need to install or update these programs accordingly.

> **Note:** `../configure` may fail to issue warnings for certain documentation build requirements that are not met. If you experience problems when building the documentation, you may need to do a manual check of Section 4.2.3 [Requirements for building documentation], page 35.

Configuring target directories

> **Note:** make sure that you are in the '`build/`' subdirectory of your source tree.

If you intend to use your local build to install a local copy of the program, you will probably want to configure the installation directory. Here are the relevant lines taken from the output of `../configure --help`:

> By default, '`make install`' will install all the files in '`/usr/local/bin`', '`/usr/local/lib`' etc. You can specify an installation prefix other than '`/usr/local`' using '`--prefix`', for instance '`--prefix=$HOME`'.

A typical installation prefix is '`$HOME/usr`':

```
../configure --prefix=$HOME/usr
```

Note that if you plan to install a local build on a system where you do not have root privileges, you will need to do something like this anyway—`make install` will only succeed if the installation prefix points to a directory where you have write permission (such as your home directory). The installation directory will be automatically created if necessary.

The location of the `lilypond` command installed by this process will be '*prefix*`/bin/lilypond`'; you may want to add '*prefix*`/bin/`' to your `$PATH` if it is not already included.

It is also possible to specify separate installation directories for different types of program files. See the full output of `../configure --help` for more information.

If you encounter any problems, please see Section 4.7 [Problems], page 42.

4.5 Compiling LilyPond

4.5.1 Using `make`

> **Note:** make sure that you are in the '`build/`' subdirectory of your source tree.

LilyPond is compiled with the `make` command. Assuming `make` is configured properly, you can simply run:

```
make
```

'`make`' is short for '`make all`'. To view a list of `make` targets, run:

```
make help
```

TODO: Describe what `make` actually does.

4.5.2 Saving time with the '`-j`' option

If your system has multiple CPUs, you can speed up compilation by adding '`-j`*X*' to the `make` command, where '*X*' is one more than the number of cores you have. For example, a typical Core2Duo machine would use:

```
make -j3
```

If you get errors using the '`-j`' option, and '`make`' succeeds without it, try lowering the *X* value.

Because multiple jobs run in parallel when '`-j`' is used, it can be difficult to determine the source of an error when one occurs. In that case, running '`make`' without the '`-j`' is advised.

4.5.3 Compiling for multiple platforms

If you want to build multiple versions of LilyPond with different configuration settings, you can use the `--enable-config=`*CONF* option of `configure`. You should use `make conf=`*CONF* to generate the output in '`out-`*CONF*'. For example, suppose you want to build with and without profiling, then use the following for the normal build

```
./configure --prefix=$HOME/usr/ --enable-checking
make
```

and for the profiling version, specify a different configuration

```
./configure --prefix=$HOME/usr/ --enable-profiling \
  --enable-config=prof --disable-checking
make conf=prof
```

If you wish to install a copy of the build with profiling, don't forget to use `conf=`*CONF* when issuing `make install`:

```
make conf=prof install
```

See also

Section 4.6.1 [Installing LilyPond from a local build], page 39

4.5.4 Useful `make` variables

If a less verbose build output if desired, the variable `QUIET_BUILD` may be set to `1` on `make` command line, or in '`local.make`' at top of the build tree.

4.6 Post-compilation options

4.6.1 Installing LilyPond from a local build

If you configured `make` to install your local build in a directory where you normally have write permission (such as your home directory), and you have compiled LilyPond by running `make`, you can install the program in your target directory by running:

```
make install
```

If instead, your installation directory is not one that you can normally write to (such as the default '`/usr/local/`', which typically is only writeable by the superuser), you will need to temporarily become the superuser when running `make install`:

```
sudo make install
```

or...

```
su -c 'make install'
```

If you don't have superuser privileges, then you need to configure the installation directory to one that you can write to, and then re-install. See [Configuring target directories], page 37.

4.6.2 Generating documentation

Documentation editor's edit/compile cycle

- Initial documentation build:

```
make [-jX]
make [-jX CPU_COUNT=X] doc  ## can take an hour or more
```

- Edit/compile cycle:

```
## edit source files, then...

make [-jX]                  ## needed if editing outside
                            ##   Documentation/, but useful anyway
                            ##   for finding Texinfo errors.
touch Documentation/*te??   ## bug workaround
make [-jX CPU_COUNT=X] doc  ## usually faster than initial build.
```

- Reset:

  In some cases, it is possible to clean the compiled documentation with '`make doc-clean`', but this method is not guaranteed to fix everything. Instead, we recommend that you delete your '`build/`' directory, and begin compiling from scratch. Since the documentation compile takes much longer than the non-documentation compile, this does not increase the overall time by a great deal.

Building documentation

After a successful compile (using `make`), the documentation can be built by issuing:

```
make doc
```

The first time you run `make doc`, the process can easily take an hour or more. After that, `make doc` only makes changes to the pre-built documentation where needed, so it may only take a minute or two to test changes if the documentation is already built.

If `make doc` succeeds, the HTML documentation tree is available in '`out-www/offline-root/`', and can be browsed locally. Various portions of the documentation can be found by looking in '`out/`' and '`out-www`' subdirectories in other places in the source tree, but these are only *portions* of the docs. Please do not complain about anything which is broken in those places; the only complete set of documentation is in '`out-www/offline-root/`' from the top of the source tree.

Compilation of documentation in Info format with images can be done separately by issuing:

```
make info
```

Known issues and warnings

If source files have changed since the last documentation build, output files that need to be rebuilt are normally rebuilt, even if you do not run `make doc-clean` first. However, build dependencies in the documentation are so complex that some newly-edited files may not be rebuilt as they should be; a workaround is to `touch` the top source file for any manual you've edited. For example, if you make changes to a file in '`notation/`', do:

```
touch Documentation/notation.tely
```

The top sources possibly affected by this are:

```
Documentation/extend.texi
Documentation/changes.tely
Documentation/contributor.texi
Documentation/essay.tely
Documentation/extending.tely
Documentation/learning.tely
Documentation/notation.tely
Documentation/snippets.tely
Documentation/usage.tely
Documentation/web.texi
```

You can `touch` all of them at once with:

```
touch Documentation/*te??
```

However, this will rebuild all of the manuals indiscriminately—it is more efficient to `touch` only the affected files.

Saving time with `CPU_COUNT`

The most time consuming task for building the documentation is running LilyPond to build images of music, and there cannot be several simultaneously running `lilypond-book` instances, so the '`-j`' `make` option does not significantly speed up the build process. To help speed it up, the makefile variable '`CPU_COUNT`' may be set in '`local.make`' or on the command line to the number of `.ly` files that LilyPond should process simultaneously, e.g. on a bi-processor or dual core machine:

```
make -j3 CPU_COUNT=3 doc
```

The recommended value of '`CPU_COUNT`' is one plus the number of cores or processors, but it is advisable to set it to a smaller value unless your system has enough RAM to run that many simultaneous LilyPond instances. Also, values for the '`-j`' option that pose problems with '`make`' are less likely to pose problems with '`make doc`' (this applies to both '`-j`' and '`CPU_COUNT`'). For example, with a quad-core processor, it is possible for '`make -j5 CPU_COUNT=5 doc`' to work consistently even if '`make -j5`' rarely succeeds.

AJAX search

To build the documentation with interactive searching, use:

```
make doc AJAX_SEARCH=1
```

This requires PHP, and you must view the docs via a http connection (you cannot view them on your local filesystem).

> **Note:** Due to potential security or load issues, this option is not enabled in the official documentation builds. Enable at your own risk.

Installing documentation

The HTML, PDF and if available Info files can be installed into the standard documentation path by issuing

```
make install-doc
```

This also installs Info documentation with images if the installation prefix is properly set; otherwise, instructions to complete proper installation of Info documentation are printed on standard output.

To install the Info documentation separately, run:

```
make install-info
```

Note that to get the images in Info documentation, `install-doc` target creates symbolic links to HTML and PDF installed documentation tree in '*prefix*`/share/info`', in order to save disk space, whereas `install-info` copies images in '*prefix*`/share/info`' subdirectories.

It is possible to build a documentation tree in '`out-www/online-root/`', with special processing, so it can be used on a website with content negotiation for automatic language selection; this can be achieved by issuing

```
make WEB_TARGETS=online doc
```

and both 'offline' and 'online' targets can be generated by issuing

```
make WEB_TARGETS="offline online" doc
```

Several targets are available to clean the documentation build and help with maintaining documentation; an overview of these targets is available with

```
make help
```

from every directory in the build tree. Most targets for documentation maintenance are available from '`Documentation/`'; for more information, see Section "Documentation work" in *Contributor's Guide*.

The makefile variable `QUIET_BUILD` may be set to `1` for a less verbose build output, just like for building the programs.

Building documentation without compiling

The documentation can be built locally without compiling LilyPond binary, if LilyPond is already installed on your system.

From a fresh Git checkout, do

```
./autogen.sh   # ignore any warning messages
cp GNUmakefile.in GNUmakefile
make -C scripts && make -C python
nice make LILYPOND_EXTERNAL_BINARY=/path/to/bin/lilypond doc
```

Please note that this may break sometimes – for example, if a new feature is added with a test file in input/regression, even the latest development release of LilyPond will fail to build the docs.

You may build the manual without building all the '`input/*`' stuff (i.e. mostly regression tests): change directory, for example to '`Documentation/`', issue `make doc`, which will build

documentation in a subdirectory 'out-www' from the source files in current directory. In this case, if you also want to browse the documentation in its post-processed form, change back to top directory and issue

```
make out=www WWW-post
```

Known issues and warnings

You may also need to create a script for pngtopnm and pnmtopng. On GNU/Linux, I use this:

```
export LD_LIBRARY_PATH=/usr/lib
exec /usr/bin/pngtopnm "$@"
```

On MacOS X with fink, I use this:

```
export DYLD_LIBRARY_PATH=/sw/lib
exec /sw/bin/pngtopnm "$@"
```

On MacOS X with macports, you should use this:

```
export DYLD_FALLBACK_LIBRARY_PATH=/opt/local/lib
exec /opt/local/bin/pngtopnm "$@"
```

4.6.3 Testing LilyPond binary

LilyPond comes with an extensive suite that exercises the entire program. This suite can be used to test that the binary has been built correctly.

The test suite can be executed with:

```
make test
```

If the test suite completes successfully, the LilyPond binary has been verified.

More information on the regression test suite is found at Section "Regression tests" in *Contributor's Guide*.

4.7 Problems

For help and questions use lilypond-user@gnu.org. Send bug reports to bug-lilypond@gnu.org.

Bugs that are not fault of LilyPond are documented here.

Bison 1.875

There is a bug in bison-1.875: compilation fails with "parse error before 'goto'" in line 4922 due to a bug in bison. To fix, please recompile bison 1.875 with the following fix

```
$ cd lily; make out/parser.cc
$ vi +4919 out/parser.cc
# append a semicolon to the line containing "__attribute__ ((__unused__))
# save
$ make
```

Compiling on MacOS X

Here are special instructions for compiling under MacOS X. These instructions assume that dependencies are installed using MacPorts. The instructions have been tested using OS X 10.5 (Leopard).

First, install the relevant dependencies using MacPorts.

Next, add the following to your relevant shell initialization files. This is ~/.profile by default. You should create this file if it does not exist.

```
export PATH=/opt/local/bin:/opt/local/sbin:$PATH
export DYLD_FALLBACK_LIBRARY_PATH=/opt/local/lib:$DYLD_FALLBACK_LIBRARY_PATH
```

Now you must edit the generated '`config.make`' file. Change

```
FLEXLEXER_FILE = /usr/include/FlexLexer.h
```

to:

```
FLEXLEXER_FILE = /opt/local/include/FlexLexer.h
```

At this point, you should verify that you have the appropriate fonts installed with your ghostscript installation. Check `ls /opt/local/share/ghostscript/fonts` for: 'c0590*' files (.pfb, .pfb and .afm). If you don't have them, run the following commands to grab them from the ghostscript SVN server and install them in the appropriate location:

```
svn export http://svn.ghostscript.com/ghostscript/tags/urw-fonts-1.0.7pre44/
sudo mv urw-fonts-1.0.7pre44/* /opt/local/share/ghostscript/fonts/
rm -rf urw-fonts-1.07pre44
```

Now run the `./configure` script. To avoid complications with automatic font detection, add

```
--with-ncsb-dir=/opt/local/share/ghostscript/fonts
```

Solaris

Solaris7, ./configure

'`./configure`' needs a POSIX compliant shell. On Solaris7, '`/bin/sh`' is not yet POSIX compliant, but '`/bin/ksh`' or bash is. Run configure like

```
CONFIG_SHELL=/bin/ksh ksh -c ./configure
```

or

```
CONFIG_SHELL=/bin/bash bash -c ./configure
```

FreeBSD

To use system fonts, dejaview must be installed. With the default port, the fonts are installed in '`usr/X11R6/lib/X11/fonts/dejavu`'.

Open the file '`$LILYPONDBASE/usr/etc/fonts/local.conf`' and add the following line just after the `<fontconfig>` line. (Adjust as necessary for your hierarchy.)

```
<dir>/usr/X11R6/lib/X11/fonts</dir>
```

International fonts

On Mac OS X, all fonts are installed by default. However, finding all system fonts requires a bit of configuration; see this post on the `lilypond-user` mailing list.

On Linux, international fonts are installed by different means on every distribution. We cannot list the exact commands or packages that are necessary, as each distribution is different, and the exact package names within each distribution changes. Here are some hints, though:

```
Red Hat Fedora

    taipeifonts fonts-xorg-truetype ttfonts-ja fonts-arabic \
         ttfonts-zh_CN fonts-ja fonts-hebrew

Debian GNU/Linux

   apt-get install emacs-intl-fonts xfonts-intl-.* \
        ttf-kochi-gothic ttf-kochi-mincho \
        xfonts-bolkhov-75dpi xfonts-cronyx-100dpi xfonts-cronyx-75dpi
```

Using lilypond python libraries

If you want to use lilypond's python libraries (either running certain build scripts manually, or using them in other programs), set `PYTHONPATH` to '`python/out`' in your build directory, or '`.../usr/lib/lilypond/current/python`' in the installation directory structure.

4.8 Concurrent stable and development versions

It can be useful to have both the stable and the development versions of Lilypond available at once. One way to do this on GNU/Linux is to install the stable version using the precompiled binary, and run the development version from the source tree. After running `make all` from the top directory of the Lilypond source files, there will be a binary called `lilypond` in the `out` directory:

```
<path to>/lilypond/out/bin/lilypond
```

This binary can be run without actually doing the `make install` command. The advantage to this is that you can have all of the latest changes available after pulling from git and running `make all`, without having to uninstall the old version and reinstall the new.

So, to use the stable version, install it as usual and use the normal commands:

```
lilypond foobar.ly
```

To use the development version, create a link to the binary in the source tree by saving the following line in a file somewhere in your `$PATH`:

```
exec <path to>/lilypond/out/bin/lilypond "$@"
```

Save it as `Lilypond` (with a capital L to distinguish it from the stable `lilypond`), and make it executable:

```
chmod +x Lilypond
```

Then you can invoke the development version this way:

```
Lilypond foobar.ly
```

TODO: ADD

- other compilation tricks for developers

4.9 Build system

We currently use make and stepmake, which is complicated and only used by us. Hopefully this will change in the future.

Version-specific texinfo macros

- made with `scripts/build/create-version-itexi.py` and `scripts/build/create-weblinks-itexi.py`
- used extensively in the `WEBSITE_ONLY_BUILD` version of the website (made with '`website.make`', used on lilypond.org)
- not (?) used in the main docs?
- the numbers in VERSION file: MINOR_VERSION should be 1 more than the last release, VERSION_DEVEL should be the last **online** release. Yes, VERSION_DEVEL is less than VERSION.

5 Documentation work

There are currently 11 manuals for LilyPond, not including the translations. Each book is available in HTML, PDF, and info. The documentation is written in a language called `texinfo` – this allows us to generate different output formats from a single set of source files.

To organize multiple authors working on the documentation, we use a Version Control System (VCS) called git, previously discussed in Section 3.2 [Starting with Git], page 11.

5.1 Introduction to documentation work

Our documentation tries to adhere to our Section 5.4 [Documentation policy], page 54. This policy contains a few items which may seem odd. One policy in particular is often questioned by potential contributors: we do not repeat material in the Notation Reference, and instead provide links to the "definitive" presentation of that information. Some people point out, with good reason, that this makes the documentation harder to read. If we repeated certain information in relevant places, readers would be less likely to miss that information.

That reasoning is sound, but we have two counter-arguments. First, the Notation Reference – one of *five* manuals for users to read – is already over 500 pages long. If we repeated material, we could easily exceed 1000 pages! Second, and much more importantly, LilyPond is an evolving project. New features are added, bugs are fixed, and bugs are discovered and documented. If features are discussed in multiple places, the documentation team must find every instance. Since the manual is so large, it is impossible for one person to have the location of every piece of information memorized, so any attempt to update the documentation will invariably omit a few places. This second concern is not at all theoretical; the documentation used to be plagued with inconsistent information.

If the documentation were targeted for a specific version – say, LilyPond 2.10.5 – and we had unlimited resources to spend on documentation, then we could avoid this second problem. But since LilyPond evolves (and that is a very good thing!), and since we have quite limited resources, this policy remains in place.

A few other policies (such as not permitting the use of tweaks in the main portion of NR 1+2) may also seem counter-intuitive, but they also stem from attempting to find the most effective use of limited documentation help.

Before undertaking any large documentation work, contributors are encouraged to contact the Section 13.2 [Meisters], page 128.

5.2 Documentation suggestions

Small additions

For additions to the documentation,

1. Tell us where the addition should be placed. Please include both the section number and title (i.e. "LM 2.13 Printing lyrics").
2. Please write exact changes to the text.
3. A formal patch to the source code is *not* required; we can take care of the technical details.
4. Send the suggestions to the `bug-lilypond` mailing list as discussed in Section "Contact" in *General Information*.
5. Here is an example of a perfect documentation report:

```
To: bug-lilypond@gnu.org
From: helpful-user@example.net
Subject: doc addition
```

```
In LM 2.13 (printing lyrics), above the last line ("More options,
like..."), please add:

----
To add lyrics to a divided part, use blah blah blah.  For example,

\score {
  \notes {blah <<blah>> }
  \lyrics {blah <<blah>> }
  blah blah blah
}
----

In addition, the second sentence of the first paragraph is
confusing.  Please delete that sentence (it begins "Users
often...") and replace it with this:
----
To align lyrics with something, do this thing.
----

Have a nice day,
Helpful User
```

Larger contributions

To replace large sections of the documentation, the guidelines are stricter. We cannot remove parts of the current documentation unless we are certain that the new version is an improvement.

1. Ask on the lilypond-devel mailing list if such a rewrite is necessary; somebody else might already be working on this issue!
2. Split your work into small sections; this makes it much easier to compare the new and old documentation.
3. Please prepare a formal git patch.

Once you have followed these guidelines, please send a message to lilypond-devel with your documentation submissions. Unfortunately there is a strict "no top-posting" check on the mailing list; to avoid this, add:

> I'm not top posting.

(you must include the >) to the top of your documentation addition.

We may edit your suggestion for spelling, grammar, or style, and we may not place the material exactly where you suggested, but if you give us some material to work with, we can improve the manual much faster. Thanks for your interest!

5.3 Texinfo introduction and usage policy

5.3.1 Texinfo introduction

The language is called Texinfo; you can see its manual here:

http://www.gnu.org/software/texinfo/manual/texinfo/

However, you don't need to read those docs. The most important thing to notice is that text is text. If you see a mistake in the text, you can fix it. If you want to change the order of something, you can cut-and-paste that stuff into a new location.

> **Note:** Rule of thumb: follow the examples in the existing docs. You can learn most of what you need to know from this; if you want to do anything fancy, discuss it on `lilypond-devel` first.

5.3.2 Documentation files

All manuals live in '`Documentation/`'.

In particular, there are four user manuals, their respective master source files are '`learning.tely`' (LM, Learning Manual), '`notation.tely`' (NR, Notation Reference), '`music-glossary.tely`' (MG, Music Glossary), and '`lilypond-program`' (AU). Each chapter is written in a separate file, ending in '`.itely`' for files containing lilypond code, and '`.itexi`' for files without lilypond code, located in a subdirectory associated to the manual ('`learning/`' for '`learning.tely`', and so on); list the subdirectory of each manual to determine the filename of the specific chapter you wish to modify.

Developer manuals live in '`Documentation/`' too. Currently there is only one: the Contributor's Guide '`contrib-guide.texi`' you are reading.

Snippet files are part of documentation, and the Snippet List (SL) lives in '`Documentation/`' just like the manuals. For information about how to modify the snippet files and SL, see Chapter 7 [LSR work], page 74.

5.3.3 Sectioning commands

Most of the manual operates at the

```
@node Foo
@subsubsection Foo
```

level. Sections are created with

```
@node Foo
@subsection Foo
```

- Please leave two blank lines above a `@node`; this makes it easier to find sections in texinfo.
- Do not use any `@` commands for a `@node`. They may be used for any `@sub...` sections or headings however.

```
not:
@node @code{Foo} Bar
@subsection @code{Foo} Bar

but instead:
@node Foo Bar
@subsection @code{Foo} Bar
```

- If a heading is desired without creating a `@node`, please use the following:

```
@subheading Foo
```

- Sectioning commands (`@node` and `@section`) must not appear inside an `@ignore`. Separate those commands with a space, ie `@n ode`.

Nodes must be included inside a

```
@menu
* foo::
* bar::
@end menu
```

construct. These are easily constructed with automatic tools; see Section 5.6 [Scripts to ease doc work], page 58.

5.3.4 LilyPond formatting

- Most LilyPond examples throughout the documentation can be produced with:

  ```
  @lilypond[verbatim,quote,relative=1]
  ```

  or

  ```
  @lilypond[verbatim,quote,relative=2]
  ```

  If using any combination of `\header{}`, `\score{}` or `\layout{}` in your example, then you must omit the `relative` variable and either use absolute entry mode or an explicit `\relative{}` construction.

  If using `\book{}` in your example then you must also omit the `relative` variable and either use absolute entry mode or an explicit `\relative{}` construction. However, you must also include the `papersize=X` variable, where `X` is a defined paper size from within '`scm/paper.scm`'. This is to avoid the default `a4` paper size being used and leaving too much unnecessary whitespace and potentially awkward page breaks in the PDFs.

  The preferred `papersize`s are `a5`, `a6` or `a8landscape`.

  `a8landscape` works best for a single measure with a single title and/or single `tagline`:

  ```
  @lilypond[papersize=a8landscape,verbatim]
  \book {
    \header {
      title = "A scale in LilyPond"
    }
    \relative {
      c d e f
    }
  }
  @end lilypond
  ```

  and can also be used to easily show features that require page breaks (i.e. page numbers) without taking large amounts of space within the documentation. Do not use the `quote` option with this paper size.

  `a5` or `a6` paper sizes are best used for examples that have more than two measures of music or require multiple staves (i.e. to illustrate cross-staff features, RH and LH parts etc.) and where `\book{}` constructions are required or where `a8landscape` produces an example that is too cramped. Depending on the example the `quote` option may need to be omitted.

  In rare cases, other options may be used (or omitted), but ask first.

- Please avoid using extra spacing either after or within the `@lilypond` parameters.

  ```
  not:          @lilypond [verbatim, quote, relative=1]
  but instead:  @lilypond[verbatim,quote,relative=1]
  ```

- Inspirational headwords are produced with:

  ```
  @lilypondfile[quote,ragged-right,line-width=16\cm,staffsize=16]
  {pitches-headword.ly}
  ```

- LSR snippets are linked with:

  ```
  @lilypondfile[verbatim,lilyquote,ragged-right,texidoc,doctitle]
  {filename.ly}
  ```

- Use two spaces for indentation in lilypond examples (no tabs).

- All engravers should have double-quotes around them:

  ```
  \consists "Spans_arpeggio_engraver"
  ```

  LilyPond does not strictly require this, but it is a useful convention to follow.

- All context or layout object strings should be prefaced with `#`. Again, LilyPond does not strictly require this, but it is helpful to get users accustomed to this scheme construct, i.e. `\set Staff.instrumentName = #"cello"`
- Try to avoid using `#'` or `#`` within when describing context or layout properties outside of an `@example` or `@lilypond`, unless the description explicitly requires it.

  i.e. "...setting the `transparent` property leaves the object where it is, but makes it invisible."
- If possible, only write one bar per line.
- If you only have one bar per line, omit bar checks. If you must put more than one bar per line (not recommended), then include bar checks.
- Tweaks should, if possible, also occur on their own line.

```
not:          \override TextScript #'padding = #3 c1^"hi"
but instead:  \override TextScript #'padding = #3
              c1^"hi"
```

  excepted in Templates, where 'doctitle' may be omitted.
- Avoid long stretches of input code. Nobody is going to read them in print. Create small examples. However, this does not mean it has be minimal.
- Specify durations for at least the first note of every bar.
- If possible, end with a complete bar.
- Comments should go on their own line, and be placed before the line(s) to which they refer.
- For clarity, always use `{ }` marks even if they are not technically required; i.e.

```
not:

\context Voice \repeat unfold 2 \relative c' {
  c2 d
}

but instead:

\context Voice {
  \repeat unfold 2 {
    \relative c' {
      c2 d
    }
  }
}
```

- Add a space around `{ }` marks; i.e.

```
not:          \chordmode{c e g}
but instead:  \chordmode { c e g }
```

- Use `{ }` marks for additional `\markup` format commands; i.e.

```
not:          c^\markup \tiny\sharp
but instead:  c^\markup { \tiny \sharp }
```

- Remove any space around `< >` marks; i.e.

```
not:           < c e g > 4
but instead:   <c e g>4
```

- Beam, slur and tie marks should begin immediately after the first note with beam and phrase marks ending immediately after the last.

```
a8\( ais16[ b cis( d] b) cis4~ b' cis,\)
```

- If you want to work on an example outside of the manual (for easier/faster processing), use this header:

```
\paper {
  indent = 0\mm
  line-width = 160\mm - 2.0 * 0.4\in
  ragged-right = ##t
  force-assignment = #""
  line-width = #(- line-width (* mm  3.000000))
}

\layout {
}
```

  You may not change any of these values. If you are making an example demonstrating special `\paper{}` values, contact the Documentation Editor.

5.3.5 Text formatting

- Lines should be less than 72 characters long. (We personally recommend writing with 66-char lines, but do not bother modifying existing material). Also see the recommendations for fixed-width fonts in the Section 5.3.6 [Syntax survey], page 50.
- Do not use tabs.
- Do not use spaces at the beginning of a line (except in `@example` or `@verbatim` environments), and do not use more than a single space between words. 'makeinfo' copies the input lines verbatim without removing those spaces.
- Use two spaces after a period.
- In examples of syntax, use `@var{`*`musicexpr`*`}` for a music expression.
- Don't use `@rinternals{}` in the main text. If you're tempted to do so, you're probably getting too close to "talking through the code". If you really want to refer to a context, use `@code{}` in the main text and `@rinternals{}` in the `@seealso`.

5.3.6 Syntax survey

Comments

- `@c ...` — single line comment. '`@c NOTE:`' is a comment which should remain in the final version. (gp only command ;)
- `@ignore` — multi-line comment:

```
@ignore
...
@end ignore
```

Cross references

Enter the exact `@node` name of the target reference between the brackets (eg. '`@ref{Syntax survey}`'). Do not split a cross-reference across two lines – this causes the cross-reference to be rendered incorrectly in html documents.

- `@ref{...}` — link within current manual.
- `@rchanges{...}` — link to Changes.
- `@rcontrib{...}` — link to Contributor's Guide.
- `@ressay{...}` — link to Engraving Essay.

- `@rextend{...}` — link to Extending LilyPond.
- `@rglos{...}` — link to the Music Glossary.
- `@rinternals{...}` — link to the Internals Reference.
- `@rlearning{...}` — link to Learning Manual.
- `@rlsr{...}` — link to a Snippet section.
- `@rprogram{...}` — link to Application Usage.
- `@ruser{...}` — link to Notation Reference.
- `@rweb{...}` — link to General Information.

External links

- `@email{...}` — create a `mailto:` E-mail link.
- `@uref{`*`URL`*`[, `*`link text`*`]}` — link to an external url. Use within an `@example ... @end example`.

```
@example
@uref{URL [, link text ]}
@end example
```

Fixed-width font

- `@code{...}`, `@samp{...}` —

  Use the `@code{...}` command when referring to individual language-specific tokens (keywords, commands, engravers, scheme symbols, etc.) in the text. Ideally, a single `@code{...}` block should fit within one line in the PDF output.

  Use the `@samp{...}` command when you have a short example of user input, unless it constitutes an entire `@item` by itself, in which case `@code{...}` is preferable. Otherwise, both should only be used when part of a larger sentence within a paragraph or `@item`. Do not use `@code{...}` or `@samp{...}` inside an `@example` block, and do not use either as a free-standing paragraph; use `@example` instead.

  A single unindented line in the PDF has space for about 79 fixed-width characters (76 if indented). Within an `@item` there is space for about 75 fixed-width characters. Each additional level of `@itemize` or `@enumerate` shortens the line by about 4 columns.

  However, even short blocks of `@code{...}` and `@samp{...}` can run into the margin if the Texinfo line-breaking algorithm gets confused. Additionally, blocks that are longer than this may in fact print nicely; it all depends where the line breaks end up. If you compile the docs yourself, check the PDF output to make sure the line breaks are satisfactory.

  The Texinfo setting `@allowcodebreaks` is set to `false` in the manuals, so lines within `@code{...}` or `@samp{...}` blocks will only break at spaces, not at hyphens or underscores. If the block contains spaces, use `@w{@code{...}}` or `@w{@samp{...}}` to prevent unexpected line breaks.

  The Texinfo settings `txicodequoteundirected` and `txicodequotebacktick` are both set in the manuals, so backticks (`` ` ``) and apostrophes (`'`) placed within blocks of `@code`, `@example`, or `@verbatim` are not converted to left- and right-angled quotes (‘ ’) as they normally are within the text, so the apostrophes in ‘`@w{@code{\relative c''}}`’ will display correctly. However, these settings do not affect the PDF output for anything within a `@samp` block (even if it includes a nested `@code` block), so entering ‘`@w{@samp{\relative c''}}`’ wrongly produces ‘`\relative c’’`’ in PDF. Consequently, if you want to use a `@samp{...}` block which contains backticks or apostrophes, you should instead use ‘`@q{@code{...}}`’ (or ‘`@q{@w{@code{...}}}`’ if the block also contains spaces). Note that backslashes within `@q{...}` blocks must be entered as ‘`@bs{}`’, so the example above would be coded as ‘`@q{@w{@code{@bs{}relative c''}}}`’.

- `@command{...}` — Use when referring to command-line commands within the text (eg. '`@command{convert-ly}`'). Do not use inside an `@example` block.
- `@example` — Use for examples of program code. Do not add extraneous indentation (i.e. don't start every line with whitespace). Use the following layout (notice the use of blank lines). Omit the `@noindent` if the text following the example starts a new paragraph:

```
...text leading into the example...

@example
...
@end example

@noindent
continuation of the text...
```

  Individual lines within an `@example` block should not exceed 74 characters; otherwise they will run into the margin in the PDF output, and may get clipped. If an `@example` block is part of an `@item`, individual lines in the `@example` block should not exceed 70 columns. Each additional level of `@itemize` or `@enumerate` shortens the line by about 4 columns.

  For long command line examples, if possible, use a trailing backslash to break up a single line, indenting the next line with 2 spaces. If this isn't feasible, use '`@smallexample ... @end smallexample`' instead, which uses a smaller fontsize. Use `@example` whenever possible, but if needed, `@smallexample` can fit up to 90 characters per line before running into the PDF margin. Each additional level of `@itemize` or `@enumerate` shortens a `@smallexample` line by about 5 columns.
- `@file{...}` — Use when referring to filenames and directories in the text. Do not use inside an `@example` block.
- `@option{...}` — Use when referring to command-line options in the text (eg. '`@option{--format}`'). Do not use inside an `@example` block.
- `@verbatim` — Prints the block exactly as it appears in the source file (including whitespace, etc.). For program code examples, use `@example` instead. `@verbatim` uses the same format as `@example`.

  Individual lines within an `@verbatim` block should not exceed 74 characters; otherwise they will run into the margin in the PDF output, and may get clipped. If an `@verbatim` block is part of an `@item`, individual lines in the `@verbatim` block should not exceed 70 columns. Each additional level of `@itemize` or `@enumerate` shortens the line by about 4 columns.

Indexing

- `@cindex ...` — General index. Please add as many as you can. Don't capitalize the first word.
- `@funindex ...` — is for a \lilycommand.

Lists

- `@enumerate` — Create an ordered list (with numbers). Always put '`@item`' on its own line. As an exception, if all the items in the list are short enough to fit on single lines, placing them on the '`@item`' lines is also permissible. '`@item`' and '`@end enumerate`' should always be preceded by a blank line.

```
@enumerate

@item
A long multi-line item like this one must begin
```

```
on a line of its own and all the other items in
the list must do so too.

@item
Even short ones

@end enumerate
@enumerate

@item Short item

@item Short item

@end enumerate
```

- `@itemize` — Create an unordered list (with bullets). Use the same format as `@enumerate`. Do not use '`@itemize @bullet`'.

Special characters

- `--`, `---` — Create an en dash (–) or an em dash (—) in the text. To print two or three literal hyphens in a row, wrap one of them in a `@w{...}` (eg. '`-@w{-}-`').
- `@@`, `@{`, `@}` — Create an at-sign (@), a left curly bracket ({), or a right curly bracket (}).
- `@bs{}` — Create a backslash within a `@q{...}`, `@qq{...}`, or `@warning{...}` block. This is a custom LilyPond macro, not a builtin @-command in Texinfo. Texinfo would also allow '`\\`', but this breaks the PDF output.
- `@tie{}` — Create a *variable-width* non-breaking space in the text (use '`@w{ }`' for a single *fixed-width* non-breaking space). Variables or numbers which consist of a single character (probably followed by a punctuation mark) should be tied properly, either to the previous or the next word. Example: '`The letter@tie{}@q{I} is skipped`'

Miscellany

- `@notation{...}` — refers to pieces of notation, e.g. '`@notation{clef}`'. Also use for specific lyrics ('`the @notation{A - men} is centered`'). Only use once per subsection per term.
- `@q{...}` — Single quotes. Used for 'vague' terms. To get a backslash (\), you must use '`@bs{}`'.
- `@qq{...}` — Double quotes. Used for actual quotes ("he said") or for introducing special input modes. To get a backslash (\), you must use '`@bs{}`'.
- `@var{...}` — Use for metasyntactic variables (such as *`foo`*, *`bar`*, *`arg1`*, etc.). In most cases, when the `@var{...}` command appears in the text (and not in an `@example` block) it should be wrapped with an appropriate texinfo code-highlighting command (such as `@code`, `@samp`, `@file`, `@command`, etc.). For example: '`@code{@var{foo}}`', '`@file{@var{myfile.ly}}`', '`@samp{git checkout @var{branch}}`', etc. This improves readability in the PDF and HTML output.
- `@version{}` — Return the current LilyPond version string. Use '`@w{@version{}}`' if it's at the end of a line (to prevent an ugly line break in PDF); use '`@w{"@version{}"}`' if you need it in quotes.
- `@w{...}` — Do not allow any line breaks.
- `@warning{...}` — produces a "Note: " box. Use for important messages. To get a backslash (\), you must use '`@bs{}`'.

5.3.7 Other text concerns

- References must occur at the end of a sentence, for more information see the texinfo manual. Ideally this should also be the final sentence of a paragraph, but this is not required. Any link in a doc section must be duplicated in the `@seealso` section at the bottom.
- Introducing examples must be done with

```
. (i.e. finish the previous sentence/paragraph)
: (i.e. `in this example:')
, (i.e. `may add foo with the blah construct,')
```

  The old "sentence runs directly into the example" method is not allowed any more.
- Abbrevs in caps, e.g., HTML, DVI, MIDI, etc.
- Colon usage
  1. To introduce lists
  2. When beginning a quote: "So, he said,...".

     This usage is rarer. Americans often just use a comma.
  3. When adding a defining example at the end of a sentence.
- Non-ASCII characters which are in utf-8 should be directly used; this is, don't say '`Ba@ss{}tuba`' but '`Baßtuba`'. This ensures that all such characters appear in all output formats.

5.4 Documentation policy

5.4.1 Books

There are four parts to the documentation: the Learning Manual, the Notation Reference, the Program Reference, and the Music Glossary.

- Learning Manual:

  The LM is written in a tutorial style which introduces the most important concepts, structure and syntax of the elements of a LilyPond score in a carefully graded sequence of steps. Explanations of all musical concepts used in the Manual can be found in the Music Glossary, and readers are assumed to have no prior knowledge of LilyPond. The objective is to take readers to a level where the Notation Reference can be understood and employed to both adapt the templates in the Appendix to their needs and to begin to construct their own scores. Commonly used tweaks are introduced and explained. Examples are provided throughout which, while being focussed on the topic being introduced, are long enough to seem real in order to retain the readers' interest. Each example builds on the previous material, and comments are used liberally. Every new aspect is thoroughly explained before it is used.

  Users are encouraged to read the complete Learning Manual from start-to-finish.
- Notation Reference: a (hopefully complete) description of LilyPond input notation. Some material from here may be duplicated in the Learning Manual (for teaching), but consider the NR to be the "definitive" description of each notation element, with the LM being an "extra". The goal is _not_ to provide a step-by-step learning environment – do not avoid using notation that has not be introduced previously in the NR (for example, use \break if appropriate). This section is written in formal technical writing style.

  Avoid duplication. Although users are not expected to read this manual from start to finish, they should be familiar with the material in the Learning Manual (particularly "Fundamental Concepts"), so do not repeat that material in each section of this book. Also watch out for common constructs, like ^ - _ for directions – those are explained in NR 3.

In NR 1, you can write: DYNAMICS may be manually placed above or below the staff, see @ref{Controlling direction and placement}.

Most tweaks should be added to LSR and not placed directly in the '`.itely`' file. In some cases, tweaks may be placed in the main text, but ask about this first.

Finally, you should assume that users know what the notation means; explaining musical concepts happens in the Music Glossary.

- Application Usage: information about using the program lilypond with other programs (lilypond-book, operating systems, GUIs, convert-ly, etc). This section is written in formal technical writing style.

  Users are not expected to read this manual from start to finish.

- Music Glossary: information about the music notation itself. Explanations and translations about notation terms go here.

  Users are not expected to read this manual from start to finish.

- Internals Reference: not really a documentation book, since it is automagically generated from the source, but this is its name.

5.4.2 Section organization

- The order of headings inside documentation sections should be:

```
main docs
@predefined
@endpredefined
@snippets
@seealso
@knownissues
```

- You *must* include a @seealso.
  - The order of items inside the @seealso section is

```
Music Glossary:
@rglos{foo},
@rglos{bar}.

Learning Manual:
@rlearning{baz},
@rlearning{foozle}.

Notation Reference:
@ruser{faazle},
@ruser{boo}.

Application Usage:
@rprogram{blah}.

Essay on automated music engraving:
@ressay{yadda}.

Extending LilyPond:
@rextend{frob}.

Installed Files:
@file{path/to/dir/blahz}.
```

```
Snippets: @rlsr{section}.

Internals Reference:
@rinternals{fazzle},
@rinternals{booar}.
```

- If there are multiple entries, separate them by commas but do not include an 'and'.
- Always end with a period.
- Place each link on a new line as above; this makes it much easier to add or remove links. In the output, they appear on a single line.

  ("Snippets" is REQUIRED; the others are optional)
- Any new concepts or links which require an explanation should go as a full sentence(s) in the main text.
- Don't insert an empty line between @seealso and the first entry! Otherwise there is excessive vertical space in the PDF output.

- To create links, use @ref{} if the link is within the same manual.
- @predefined ... @endpredefined is for commands in '`ly/*-init.ly`'
- Do not include any real info in second-level sections (i.e. 1.1 Pitches). A first-level section may have introductory material, but other than that all material goes into third-level sections (i.e. 1.1.1 Writing Pitches).
- The @knownissues should not discuss any issues that are in the tracker, unless the issue is Priority-Postponed. The goal is to discuss any overall architecture or syntax decisions which may be interpreted as bugs. Normal bugs should not be discussed here, because we have so many bugs that it would be a huge task to keep the @knownissues current and accurate all the time.

5.4.3 Checking cross-references

Cross-references between different manuals are heavily used in the documentation, but they are not checked during compilation. However, if you compile the documentation, a script called check_texi_refs can help you with checking and fixing these cross-references; for information on usage, cd into a source tree where documentation has been built, cd into Documentation and run:

```
make check-xrefs
make fix-xrefs
```

Note that you have to find yourself the source files to fix cross-references in the generated documentation such as the Internals Reference; e.g. you can grep scm/ and lily/.

Also of interest may be the linkdoc checks on kainhofer.com. Be warned that these docs are not completely rebuilt every day, so it might not accurately reflect the current state of the docs.

http://kainhofer.com/~lilypond/linkdoc/

5.4.4 General writing

- Do not forget to create @cindex entries for new sections of text. Enter commands with @funindex, i.e.

  ```
  @cindex pitches, writing in different octaves
  @funindex \relative
  ```

  Do not bother with the @code{} (they are added automatically). These items are added to both the command index and the unified index. Both index commands should go in front of the actual material.

- @cindex entries should not be capitalized, i.e.

  ```
  @cindex time signature
  ```

  is preferred instead of "Time signature". Only use capital letters for musical terms which demand them, e.g. "D.S. al Fine".
- For scheme function index entries, only include the final part, i.e.

  ```
  @funindex modern-voice-cautionary
       and NOT
  @funindex #(set-accidental-style modern-voice-cautionary)
  ```
- Use American spelling. LilyPond's internal property names use this convention.
- Here is a list of preferred terms to be used:
  - *Simultaneous* NOT concurrent.
  - *Measure*: the unit of music.
  - *Bar line*: the symbol delimiting a measure NOT barline.
  - *Note head* NOT notehead.
  - *Chord construct* NOT just chord (when referring to < ... >)
  - *Staff* NOT stave.
  - *Staves* NOT Staffs: Phrases such as 'multiple @internalsref{Staff}s' should be rephrased to 'multiple @internalsref{Staff} contexts'.

5.4.5 Technical writing style

These refer to the NR. The LM uses a more gentle, colloquial style.

- Do not refer to LilyPond in the text. The reader knows what the manual is about. If you do, capitalization is LilyPond.
- If you explicitly refer to 'lilypond' the program (or any other command to be executed), write @command{lilypond}.
- Do not explicitly refer to the reader/user. There is no one else besides the reader and the writer.
- Avoid contractions (don't, won't, etc.). Spell the words out completely.
- Avoid abbreviations, except for commonly used abbreviations of foreign language terms such as etc. and i.e.
- Avoid fluff ("Notice that," "as you can see," "Currently,").
- The use of the word 'illegal' is inappropriate in most cases. Say 'invalid' instead.

5.5 Tips for writing docs

In the NR, I highly recommend focusing on one subsection at a time. For each subsection,

- check the mundane formatting. Are the headings (@predefined, @seealso, etc.) in the right order?
- add any appropriate index entries.
- check the links in the @seealso section – links to music glossary, internal references, and other NR sections are the main concern. Check for potential additions.
- move LSR-worthy material into LSR. Add the snippet, delete the material from the '.itely' file, and add a @lilypondfile command.
- check the examples and descriptions. Do they still work? **Do not** assume that the existing text is accurate/complete; some of the manual is highly out of date.
- is the material in the @knownissues still accurate?

- can the examples be improved (made more explanatory), or is there any missing info? (feel free to ask specific questions on -user; a couple of people claimed to be interesting in being "consultants" who would help with such questions)

In general, I favor short text explanations with good examples – "an example is worth a thousand words". When I worked on the docs, I spent about half my time just working on those tiny lilypond examples. Making easily-understandable examples is much harder than it looks.

Tweaks

In general, any \set or \override commands should go in the "select snippets" section, which means that they should go in LSR and not the '`.itely`' file. For some cases, the command obviously belongs in the "main text" (i.e. not inside `@`predefined or `@`seealso or whatever) – instrument names are a good example of this.

```
\set Staff.instrumentName = #"foo"
```

On the other side of this,

```
\override Score.Hairpin #'after-line-breaking = ##t
```

clearly belongs in LSR.

I'm quite willing to discuss specific cases if you think that a tweaks needs to be in the main text. But items that can go into LSR are easier to maintain, so I'd like to move as much as possible into there.

It would be "nice" if you spent a lot of time crafting nice tweaks for users... but my recommendation is **not** to do this. There's a lot of doc work to do without adding examples of tweaks. Tweak examples can easily be added by normal users by adding them to the LSR.

One place where a documentation writer can profitably spend time writing or upgrading tweaks is creating tweaks to deal with known issues. It would be ideal if every significant known issue had a workaround to avoid the difficulty.

See also

Section 7.2 [Adding and editing snippets], page 74.

5.6 Scripts to ease doc work

Building only one section of the documentation

In order to save build time, a script is available to build only one section of the documentation in English with a default html appearance.

The script is available as:

```
scripts/auxiliar/doc-section.sh
```

This script will require customization for your site if your LilyPond git repository is anyplace but `$HOME/lilypond`.

Assuming that no customization is required, you can setup the single section build with:

```
mkdir $HOME/lilypond/tempdocs
cp $HOME/lilypond/Documentation/out/version.itexi $HOME/lilypond/tempdocs
```

You can then build a section of the documentation with:

```
scripts/auxiliar/doc-section.sh MANUAL SECTION
```

where `SECTION` is the name of the file containing the section to be built, and `MANUAL` is replaced by the name of the directory containing the section. So, for example, to build section 1.1 of the Notation Reference, use the command:

```
scripts/auxiliar/doc-section.sh notation pitches
```

This script will not work for building sections of the Contributors' guide. For building sections of the Contributors' Guide, use:

```
scripts/auxiliar/cg-section.sh SECTION
```

where `SECTION` is the name of the file containing the sections to be built. For example, to build section 4 of the Contributors' guide, use:

```
scripts/auxiliar/cg-section.sh doc-work
```

Like `doc-section.sh`, `cg-section.sh` may need to be customized for your installation.

Stripping whitespace and generating menus

> **Note:** This script assumes that the file conforms to our doc policy; a few files still need work in this regard.

To automatically regenerate `@menu` portions and strip whitespace, use:

```
scripts/auxiliar/node-menuify.py FILENAME
```

Stripping whitespace only

To remove extra whitespace from the ends of lines, run

```
scripts/auxiliar/strip-whitespace.py Documentation/FILENAME
```

Updating doc with `convert-ly`

Don't. This should be done by programmers when they add new features. If you notice that it hasn't been done, complain to `lilypond-devel`.

5.7 Docstrings in scheme

Material in the Internals reference is generated automatically from our source code. Any doc work on Internals therefore requires modifying files in '`scm/*.scm`'. Texinfo is allowed in these docstrings.

Most documentation writers never touch these, though. If you want to work on them, please ask for help.

5.8 Translating the documentation

The mailing list `translations@lilynet.net` is dedicated to LilyPond web site and documentation translation; on this list, you will get support from the Translations Meister and experienced translators, and we regularly discuss translation issues common to all languages. All people interested in LilyPond translations are invited to subscribe to this list regardless of the amount of their contribution, by sending an email to `translations-request@lilynet.net` with subject `subscribe` and an empty message body. Unless mentioned explicitly, or except if a translations coordinator contacts you privately, you should send questions, remarks and patches to the list `translations@lilynet.net`. Please note that traffic is high on the English-speaking list `lilypond-user@gnu.org`, so it may take some time before your request or contribution is handled.

5.8.1 Getting started with documentation translation

First, get the sources of branch `lilypond/translation` from the Git repository, see Section 3.2 [Starting with Git], page 11.

Translation requirements

Working on LilyPond documentation translations requires the following pieces of software, in order to make use of dedicated helper tools:

- Python 2.4 or higher,
- GNU Make,
- Gettext,
- Git.

It is not required to build LilyPond and the documentation to translate the documentation. However, if you have enough time and motivation and a suitable system, it can be very useful to build at least the documentation so that you can check the output yourself and more quickly; if you are interested, see Chapter 4 [Compiling], page 34.

Before undertaking any large translation work, contributors are encouraged to contact the Section 13.2 [Meisters], page 128.

Which documentation can be translated

The makefiles and scripts infrastructure currently supports translation of the following documentation:

- the web site, the Learning Manual, the Notation Reference and Application Usage – Texinfo source, PDF and HTML output; Info output might be added if there is enough demand for it;
- the Changes document.

Support for translating the following pieces of documentation should be added soon, by decreasing order of priority:

- automatically generated documentation: markup commands, predefined music functions;
- the Snippets List;
- the Internals Reference.

Starting translation in a new language

At top of the source directory, do

```
./autogen.sh
```

or (if you want to install your self-compiled LilyPond locally)

```
./autogen.sh --prefix=$HOME
```

If you want to compile LilyPond – which is almost required to build the documentation, but is not required to do translation only – fix all dependencies and rerun `./configure` (with the same options as for `autogen.sh`).

Then `cd` into '`Documentation/`' and run

```
make ISOLANG=MY-LANGUAGE new-lang
```

where *MY-LANGUAGE* is the ISO 639 language code.

Finally, add a language definition for your language in '`python/langdefs.py`'.

5.8.2 Documentation translation details

Please follow all the instructions with care to ensure quality work.

All files should be encoded in UTF-8.

Files to be translated

Translation of '`Documentation/foo/bar`' should be '`Documentation/`*`LANG`*`/foo/bar`'. Unmentioned files should not be translated.

Priorities:

- 1. delivery,
- 2. 3. 4. 5. 6. later,
- 7. optional.

Files of priority 1 should be submitted along all files generated by starting a new language in the same commit and thus a unique patch, and the translation of files marked with priority 2 should be committed to Git at the same time and thus sent in a single patch. Files marked with priority 3 or more may be submitted individually. Word counts (excluding LilyPond snippets) are given for each file. For knowing how to commit your work to Git, then make patches of your new translations as well as corrections and updates, see Section 3.3 [Basic Git procedures], page 15.

```
-1- Web site
585   web.texi
4506  web/introduction.itexi
1183  web/download.itexi
1139  macros.itexi
9     po/lilypond-doc.pot (translate to po/MY_LANGUAGE.po)
0     search-box.ihtml
---   lilypond-texi2html.init (section TRANSLATIONS)
7422  total

-2- Tutorial
1200  web/manuals.itexi
124   learning.tely
2535  learning/tutorial.itely
4184  learning/common-notation.itely
8043  total

-3- Fundamental Concepts, starting of Usage and Community
11139 learning/fundamental.itely -- Fundamental concepts
135   usage.tely
3622  usage/running.itely
1189  usage/updating.itely
1755  web/community.itexi
17840 total

-4- Rest of Learning manual and Suggestions on writing LilyPond files
15408 learning/tweaks.itely -- Tweaking output
225   learning/templates.itely -- Templates
2694  usage/suggestions.itely -- Suggestions on writing LilyPond files
18327 total

-5- Notation reference
355   notation.tely
91    notation/notation.itely -- Musical notation
4479  notation/pitches.itely
6048  notation/rhythms.itely
```

```
1726  notation/expressive.itely
930   notation/repeats.itely
2163  notation/simultaneous.itely
2056  notation/staff.itely
931   notation/editorial.itely
2716  notation/text.itely
81    notation/specialist.itely -- Specialist notation
4807  notation/vocal.itely
1855  notation/chords.itely
702   notation/piano.itely
806   notation/percussion.itely
826   notation/guitar.itely
66    notation/strings.itely
242   notation/bagpipes.itely
4752  notation/ancient.itely
7069  notation/input.itely -- Input syntax
2164  notation/non-music.itely -- Non-musical notation
11017 notation/spacing.itely -- Spacing issues
12248 notation/changing-defaults.itely -- Changing defaults
5187  notation/programming-interface.itely -- Interfaces for programmers
1989  notation/notation-appendices.itely -- Notation manual tables
252   notation/cheatsheet.itely -- Cheat sheet
75558 total

-6- Rest of Application Usage
3764  usage/lilypond-book.itely -- LilyPond-book
1122  usage/converters.itely -- Converting from other formats
4886  total

-7- Appendices whose translation is optional
326   essay/literature.itely
1222  learning/scheme-tutorial.itely (should be revised first)
1548  total
```

In addition, not listed above, Snippets' titles and descriptions should be translated; they are a part of the Notation Reference and therefore their priority is 5.

Translating the Web site and other Texinfo documentation

Every piece of text should be translated in the source file, except Texinfo comments, text in `@lilypond` blocks and a few cases mentioned below.

Node names are translated, but the original node name in English should be kept as the argument of `@translationof` put after the section title; that is, every piece in the original file like

```
@node Foo bar
@section_command Bar baz
```

should be translated as

```
@node translation of Foo bar
@section_command translation of Bar baz
@translationof Foo bar
```

The argument of `@rglos` commands and the first argument of `@rglosnamed` commands must not be translated, as it is the node name of an entry in Music Glossary.

Every time you translate a node name in a cross-reference, i.e. the argument of commands `@ref, @rprogram, @rlearning, @rlsr, @ruser` or the first argument of their `*named` variants, you should make sure the target node is defined in the correct source file; if you do not intend to translate the target node right now, you should at least write the node definition (that is, the `@node @section_commmand @translationof` trio mentioned above) in the expected source file and define all its parent nodes; for each node you have defined this way but have not translated, insert a line that contains `@untranslated`. That is, you should end up for each untranslated node with something like

```
@node translation of Foo bar
@section_command translation of Bar baz
@translationof Foo bar

@untranslated
```

> **Note:** you do not have to translate the node name of a cross-reference to a node that you do not have translated. If you do, you must define an "empty" node like explained just above; this will produce a cross-reference with the translated node name in output, although the target node will still be in English. On the opposite, if all cross-references that refer to an untranslated node use the node name in English, then you do not have to define such an "empty" node, and the cross-reference text will appear in English in the output. The choice between these two strategies implies its particular maintenance requirements and is left to the translators, although the opinion of the Translation meister leans towards not translating these cross-references.

Please think of the fact that it may not make sense translating everything in some Texinfo files, and you may take distance from the original text; for instance, in the translation of the web site section Community, you may take this into account depending on what you know the community in your language is willing to support, which is possible only if you personally assume this support, or there exists a public forum or mailing list listed in Community for LilyPond in your language:

- Section "Bug reports" in *General Information*: this page should be translated only if you know that every bug report sent on your language's mailing list or forum will be handled by someone who will translate it to English and send it on bug-lilypond or add an issue in the tracker, then translate back the reply from developers.
- Section "Help us" in *General Information*: this page should be translated very freely, and possibly not at all: ask help for contributing to LilyPond for tasks that LilyPond community in your language is able and going to handle.

In any case, please mark in your work the sections which do not result from the direct translation of a piece of English translation, using comments i.e. lines starting with '`@c`'.

Finally, press in Emacs `C-c C-u C-a` to update or generate menus. This process should be made easier in the future, when the helper script `texi-langutils.py` and the makefile target are updated.

Some pieces of text manipulated by build scripts that appear in the output are translated in a '`.po`' file – just like LilyPond output messages – in '`Documentation/po`'. The Gettext domain is named `lilypond-doc`, and unlike `lilypond` domain it is not managed through the Free Translation Project.

Take care of using typographic rules for your language, especially in '`macros.itexi`'.

If you wonder whether a word, phrase or larger piece of text should be translated, whether it is an argument of a Texinfo command or a small piece sandwiched between two Texinfo commands, try to track whether and where it appears in PDF and/or HTML output as visible text. This piece of advice is especially useful for translating '`macros.itexi`'.

Please keep verbatim copies of music snippets (in `@lilypond` blocs). However, some music snippets containing text that shows in the rendered music, and sometimes translating this text really helps the user to understand the documentation; in this case, and only in this case, you may as an exception translate text in the music snippet, and then you must add a line immediately before the `@lilypond` block, starting with

```
@c KEEP LY
```

Otherwise the music snippet would be reset to the same content as the English version at next `make snippet-update` run – see [Updating documentation translation], page 65.

When you encounter

```
@lilypondfile[<number of fragment options>,texidoc]{filename.ly}
```

in the source, open '`Documentation/snippets/`*`filename`*`.ly`', translate the `texidoc` header field it contains, enclose it with `texidoc`*`MY-LANGUAGE`*` = "` and `"`, and write it into '`Documentation/`*`MY-LANGUAGE`*`/texidocs/`*`filename`*`.texidoc`'. Additionally, you may translate the snippet's title in `doctitle` header field, in case `doctitle` is a fragment option used in `@lilypondfile`; you can do this exactly the same way as `texidoc`. For instance, '`Documentation/`*`MY-LANGUAGE`*`/texidocs/`*`filename`*`.texidoc`' may contain

```
doctitlees = "Spanish title baz"
texidoces = "
Spanish translation blah
"
```

Then, you should get these translated strings into compiled snippets in '`Documentation/snippets`', see 'General guidelines' in Section 7.2 [Adding and editing snippets], page 74.

`@example` blocks need not be verbatim copies, e.g. variable names, file names and comments should be translated.

Finally, please carefully apply every rule exposed in Section 5.3 [Texinfo introduction and usage policy], page 46, and Section 5.4 [Documentation policy], page 54. If one of these rules conflicts with a rule specific to your language, please ask the Translation meister on `translations@lilynet.net` list and/or the Documentation Editors on `lilypond-devel@gnu.org` list.

Adding a Texinfo manual

In order to start translating a new manual whose basename is *FOO*, do

```
cd Documentation/MY-LANGUAGE
cp ../FOO.tely .
mkdir FOO
cp web/GNUmakefile FOO
```

then append *FOO* to variable `SUBDIRS` in Documentation/*MY-LANGUAGE*/GNUmakefile, then translate file *MY-LANGUAGE*/*FOO*.tely and run `skeleton-update`:

```
cd Documentation/
make ISOLANG=MY-LANGUAGE TEXI_LANGUTIL_FLAGS=--head-only skeleton-update
```

Your are now ready to translate the new manual exactly like the web site or the Learning Manual.

5.8.3 Documentation translation maintenance

Several tools have been developed to make translations maintenance easier. These helper scripts make use of the power of Git, the version control system used for LilyPond development.

You should use them whenever you would like to update the translation in your language, which you may do at the frequency that fits your and your cotranslators' respective available times. In the case your translation is up-do-date (which you can discover in the first subsection below), it is enough to check its state every one or two weeks. If you feel overwhelmed by the quantity of documentation to be updated, see [Maintaining without updating translations], page 67.

Check state of translation

First pull from Git – see Section 3.3.2 [Pulling and rebasing], page 15, but DO NOT rebase unless you are sure to master the translation state checking and updating system – then cd into '`Documentation/`' (or at top of the source tree, replace `make` with `make -C Documentation`) and run

```
make ISOLANG=MY_LANGUAGE check-translation
```

This presents a diff of the original files since the most recent revision of the translation. To check a single file, cd into '`Documentation/`' and run

```
make CHECKED_FILES=MY_LANGUAGE/manual/foo.itely check-translation
```

In case this file has been renamed since you last updated the translation, you should specify both old and new file names, e.g. `CHECKED_FILES=MY_LANGUAGE/{manual,user}/foo.itely`.

To see only which files need to be updated, do

```
make ISOLANG=MY_LANGUAGE check-translation | grep 'diff --git'
```

To avoid printing terminal colors control characters, which is often desirable when you redirect output to a file, run

```
make ISOLANG=MY_LANGUAGE NO_COLOR=1 check-translation
```

You can see the diffs generated by the commands above as changes that you should make in your language to the existing translation, in order to make your translation up to date.

> **Note:** do not forget to update the committish in each file you have completely updated, see [Updating translation committishes], page 66.

Global state of the translation is recorded in '`Documentation/translations.itexi`', which is used to generate Translations status page. To update that page, do from '`Documentation/`'

```
make translation-status
```

This will also leave '`out/translations-status.txt`', which contains up-to-dateness percentages for each translated file, and update word counts of documentation files in this Guide.

See also

[Maintaining without updating translations], page 67.

Updating documentation translation

Instead of running `check-translation`, you may want to run `update-translation`, which will run your favorite text editor to update files. First, make sure environment variable `EDITOR` is set to a text editor command, then run from '`Documentation/`'

```
make ISOLANG=MY_LANGUAGE update-translation
```

or to update a single file

```
make CHECKED_FILES=MY_LANGUAGE/manual/foo.itely update-translation
```

For each file to be updated, `update-translation` will open your text editor with this file and a diff of the file in English; if the diff cannot be generated or is bigger than the file in English itself, the full file in English will be opened instead.

> **Note:** do not forget to update the committish in each file you have completely updated, see [Updating translation committishes], page 66.

Texinfo skeleton files, i.e. '`.itely`' files not yet translated, containing only the first node of the original file in English can be updated automatically: whenever `make check-translation` shows that such files should be updated, run from '`Documentation/`'

```
make ISOLANG=MY_LANGUAGE skeleton-update
```

'`.po`' message catalogs in '`Documentation/po/`' may be updated by issuing from '`Documentation/`' or '`Documentation/po/`'

```
make po-update
```

> **Note:** if you run po-update and somebody else does the same and pushes before you push or send a patch to be applied, there will be a conflict when you pull. Therefore, it is better that only the Translation meister runs this command.

Updating music snippets can quickly become cumbersome, as most snippets should be identical in all languages. Fortunately, there is a script that can do this odd job for you (run from '`Documentation/`'):

```
make ISOLANG=MY_LANGUAGE snippet-update
```

This script overwrites music snippets in '*`MY_LANGUAGE/foo/every.itely`*' with music snippets from '*`foo/every.itely`*'. It ignores skeleton files, and keeps intact music snippets preceded with a line starting with `@c KEEP LY`; it reports an error for each '`.itely`' that has not the same music snippet count in both languages. Always use this script with a lot of care, i.e. run it on a clean Git working tree, and check the changes it made with `git diff` before committing; if you don't do so, some `@lilypond` snippets might be broken or make no sense in their context.

When you have updated texidocs in '`Documentation/`*`MY-LANGUAGE`*`/texidocs`', you can get these changes into compiled snippets in '`Documentation/snippets`', see 'General guidelines' in Section 7.2 [Adding and editing snippets], page 74.

Finally, a command runs the three update processes above for all enabled languages (from '`Documentation/`'):

```
make all-translations-update
```

Use this command with caution, and keep in mind it will not be really useful until translations are stabilized after the end of GDP and GOP.

See also

[Maintaining without updating translations], page 67, Section 7.2 [Adding and editing snippets], page 74.

Updating translation committishes

At the beginning of each translated file except PO files, there is a committish which represents the revision of the sources which you have used to translate this file from the file in English.

When you have pulled and updated a translation, it is very important to update this committish in the files you have completely updated (and only these); to do this, first commit possible

changes to any documentation in English which you are sure to have done in your translation as well, then replace in the up-to-date translated files the old committish by the committish of latest commit, which can be obtained by doing

```
git rev-list HEAD |head -1
```

A special case is updating Snippet documentation strings in '`Documentation/MY-LANGUAGE/texidocs`'. For these to be correctly marked as up-to-date, first run `makelsr.py` as explained in Section 7.2 [Adding and editing snippets], page 74, and commit the resulting compiled snippets left in '`Documentation/snippets/`'. Say the SHA1 ID code of this commit is <C>. Now edit again your translated files in '`Documentation/MY-LANGUAGE/texidocs`' adjusting the 40-digit committish that appears in the text to be <C>; finally, commit these updated files. Not doing so would result in changes made both to your updates and original snippets to persistently appear in the check-translation output as if they were out of sync.

This two-phase mechanism avoids the (practically) unsolvable problem of guessing what committish will have our update, and pretending to put this very committish on the files in the same commit.

See also

Chapter 7 [LSR work], page 74.

5.8.4 Translations management policies

These policies show the general intent of how the translations should be managed, they aim at helping translators, developers and coordinators work efficiently.

Maintaining without updating translations

Keeping translations up to date under heavy changes in the documentation in English may be almost impossible, especially as during the former Grand Documentation Project (GDP) or the Grand Organization Project (GOP) when a lot of contributors brings changes. In addition, translators may be — and that is a very good thing — involved in these projects too.

it is possible — and even recommended — to perform some maintenance that keeps translated documentation usable and eases future translation updating. The rationale below the tasks list motivates this plan.

The following tasks are listed in decreasing priority order.

1. Update macros.itexi. For each obsolete macro definition, if it is possible to update macro usage in documentation with an automatic text or regexp substitution, do it and delete the macro definition from '`macros.itexi`'; otherwise, mark this macro definition as obsolete with a comment, and keep it in '`macros.itexi`' until the documentation translation has been updated and no longer uses this macro.
2. Update '`*.tely`' files completely with `make check-translation` – you may want to redirect output to a file because of overwhelming output, or call check-translation.py on individual files, see [Check state of translation], page 65.
3. In '`.itelys`', match sections and .itely file names with those from English docs, which possibly involves moving nodes contents in block between files, without updating contents itself. In other words, the game is catching where has gone each section. In Learning manual, and in Notation Reference sections which have been revised in GDP, there may be completely new sections: in this case, copy `@node` and `@section`-command from English docs, and add the marker for untranslated status `@untranslated` on a single line. Note that it is not possible to exactly match subsections or subsubsections of documentation in English, when contents has been deeply revised; in this case, keep obsolete (sub)subsections in the translation, marking them with a line `@c obsolete` just before the node.

   Emacs with Texinfo mode makes this step easier:

- without Emacs AucTeX installed, `C-c C-s` shows structure of current Texinfo file in a new buffer `*Occur*`; to show structure of two files simultaneously, first split Emacs window in 4 tiles (with `C-x 1` and `C-x 2`), press `C-c C-s` to show structure of one file (e.g. the translated file), copy `*Occur*` contents into `*Scratch*`, then press `C-c C-s` for the other file.

  If you happen to have installed AucTeX, you can either call the macro by doing `M-x texinfo-show-structure` or create a key binding in your '`~/.emacs`', by adding the four following lines:

```
(add-hook 'Texinfo-mode-hook
          '(lambda ()
             (define-key Texinfo-mode-map "\C-cs"
              'texinfo-show-structure)))
```

  and then obtain the structure in the `*Occur*` buffer with `C-c s`.
- Do not bother updating `@menu`s when all menu entries are in the same file, just do `C-c C-u C-a` ("update all menus") when you have updated all the rest of the file.
- Moving to next or previous node using incremental search: press `C-s` and type `node` (or `C-s @node` if the text contains the word 'node') then press `C-s` to move to next node or `C-r` to move to previous node. Similar operation can be used to move to the next/previous section. Note that every cursor move exits incremental search, and hitting `C-s` twice starts incremental search with the text entered in previous incremental search.
- Moving a whole node (or even a sequence of nodes): jump to beginning of the node (quit incremental search by pressing an arrow), press `C-SPACE`, press `C-s node` and repeat `C-s` until you have selected enough text, cut it with `C-w` or `C-x`, jump to the right place (moving between nodes with the previous hint is often useful) and paste with `C-y` or `C-v`.

4. Update sections finished in the English documentation; check sections status at http://lilypondwiki.tuxfamily.org/index.php?title=Documentation_coordination.
5. Update documentation PO. It is recommended not to update strings which come from documentation that is currently deeply revised in English, to avoid doing the work more than once.
6. Fix broken cross-references by running (from '`Documentation/`')

   `make ISOLANG=`*`YOUR-LANGUAGE`* `fix-xrefs`

   This step requires a successful documentation build (with `make doc`). Some cross-references are broken because they point to a node that exists in the documentation in English, which has not been added to the translation; in this case, do not fix the cross-reference but keep it `"broken"`, so that the resulting HTML link will point to an existing page of documentation in English.

Rationale

You may wonder if it would not be better to leave translations as-is until you can really start updating translations. There are several reasons to do these maintenance tasks right now.

- This will have to be done sooner or later anyway, before updating translation of documentation contents, and this can already be done without needing to be redone later, as sections of documentation in English are mostly revised once. However, note that not all documentation sectioning has been revised in one go, so all this maintenance plan has to be repeated whenever a big reorganization is made.
- This just makes translated documentation take advantage of the new organization, which is better than the old one.

- Moving and renaming sections to match sectioning of documentation in English simplify future updating work: it allows updating the translation by side-by-side comparison, without bothering whether cross-reference names already exist in the translation.
- Each maintenance task except 'Updating PO files' can be done by the same person for all languages, which saves overall time spent by translators to achieve this task: the node names and section titles are in English, so you can do. It is important to take advantage of this now, as it will be more complicated (but still possible) to do step 3 in all languages when documentation is compiled with `texi2html` and node names are directly translated in source files.

Managing documentation translation with Git

This policy explains how to manage Git branches and commit translations to Git.

- Translation changes matching master branch are preferably made on `lilypond/translation` branch; they may be pushed directly to `master` only if they do not break compilation of LilyPond and its documentation, and in this case they should be pushed to `lilypond/translation` too. Similarly, changes matching `stable/X.Y` are preferably made on `lilypond/X.Ytranslation`.
- `lilypond/translation` Git branch may be merged into master only if LilyPond (`make all`) and documentation (`make doc`) compile successfully.
- `master` Git branch may be merged into `lilypond/translation` whenever `make` and `make doc` are successful (in order to ease documentation compilation by translators), or when significant changes had been made in documentation in English in master branch.
- General maintenance may be done by anybody who knows what he does in documentation in all languages, without informing translators first. General maintenance include simple text substitutions (e.g. automated by sed), compilation fixes, updating Texinfo or lilypond-book commands, updating macros, updating ly code, fixing cross-references, and operations described in [Maintaining without updating translations], page 67.

5.8.5 Technical background

A number of Python scripts handle a part of the documentation translation process. All scripts used to maintain the translations are located in '`scripts/auxiliar/`'.

- '`check_translation.py`' – show diff to update a translation,
- '`texi-langutils.py`' – quickly and dirtily parse Texinfo files to make message catalogs and Texinfo skeleton files,
- '`texi-skeleton-update.py`' – update Texinfo skeleton files,
- '`update-snippets.py`' – synchronize ly snippets with those from English docs,
- '`translations-status.py`' – update translations status pages and word counts in the file you are reading,
- '`tely-gettext.py`' – gettext node names, section titles and references in the sources; WARNING only use this script once for each file, when support for "makeinfo –html" has been dropped.

Other scripts are used in the build process, in '`scripts/build/`':

- '`mass-link.py`' – link or symlink files between English documentation and documentation in other languages.

Python modules used by scripts in '`scripts/auxiliar/`' or '`scripts/build/`' (but not by installed Python scripts) are located in '`python/auxiliar/`':

- '`manuals_definitions.py`' – define manual names and name of cross-reference Texinfo macros,

- '`buildlib.py`' – common functions (read piped output of a shell command, use Git),
- '`postprocess_html.py`' (module imported by '`www_post.py`') – add footer and tweak links in HTML pages.

And finally

- '`python/langdefs.py`' – language definitions module

6 Website work

6.1 Introduction to website work

The website is *not* written directly in HTML; instead, the source is Texinfo, which is then generated into HTML, PDF, and Info formats. The sources are

```
Documentation/web.texi
Documentation/web/*.texi
```

Unless otherwise specified, follow the instructions and policies given in Chapter 5 [Documentation work], page 45. That chapter also contains a quick introduction to Texinfo; consulting an external Texinfo manual should be not necessary.

Exceptions to the documentation policies

- Sectioning: the website only uses chapters and sections; no subsections or subsubsections.
- `@ref{}`s to other manuals (`@ruser`, `@rlearning`, etc): you can't link to any pieces of automatically generated documentation, like the IR or certain NR appendices.
- The bibliography in Community->Publications is generated automatically from '`.bib`' files; formatting is done automatically by '`texi-web.bst`'.
- ...
- For anything not listed here, just follow the same style as the existing website texinfo files.

6.2 Uploading and security

The website is generated hourly by user `graham` the host `lilypond.org`. For security reasons, we do not use the makefiles and scripts directly from git; copies of the relevant scripts are examined and copied to `~graham/lilypond/trusted-scripts/`

Initial setup

You should symlink your own '`~/lilypond/`' to '`~graham/lilypond/`'

If this directory does not exist, make it. Git master should go in '`~/lilypond/lilypond-git/`' but make sure you enable:

```
git config core.filemode false
```

If you have created any files in '`~graham/lilypond/`' then please run:

```
chgrp lilypond ~graham/lilypond/ -R
chmod 775 ~graham/lilypond/ -R
```

To reduce the CPU burden on the shared host (as well as some security concerns), the '`Documentation/pictures/`' and '`Documentation/web/ly-examples/`' directories are **not** compiled. You need to upload them, and if they ever change, a user in the `lilypond` group must upload them to '`~graham/lilypond/media`' on the host.

Upload latest pictures/ and ly-examples/ (local script):

> **Note:** You may need to change a number of items in the below script.

```
### upload-lily-web-media.sh
#!/bin/sh
BUILD_DIR=$HOME/src/build-lilypond

PICS=$BUILD_DIR/Documentation/pictures/out-www/
EXAMPLES=$BUILD_DIR/Documentation/web/ly-examples/out-www/
```

```
cd $BUILD_DIR
rsync -a $PICS graham@lilypond.org:lilypond/media/pictures
rsync -a $EXAMPLES graham@lilypond.org:lilypond/media/ly-examples
```

Normal maintenance

Get latest source code:

```
### update-git.sh
#!/bin/sh
cd $HOME/lilypond/lilypond-git
git fetch origin
git merge origin/master
```

Check for any updates to trusted scripts / files:

```
### check-git.sh
#!/bin/sh
GIT=$HOME/lilypond/lilypond-git
DEST=$HOME/lilypond/trusted-scripts
diff -u $DEST/website.make $GIT/make/website.make
diff -u $DEST/lilypond-texi2html.init $GIT/Documentation/lilypond-texi2html.init
diff -u $DEST/extract_texi_filenames.py $GIT/scripts/build/extract_texi_filenames.py
diff -u $DEST/create-version-itexi.py $GIT/scripts/build/create-version-itexi.py
diff -u $DEST/create-weblinks-itexi.py $GIT/scripts/build/create-weblinks-itexi.py
diff -u $DEST/mass-link.py $GIT/scripts/build/mass-link.py
diff -u $DEST/website_post.py $GIT/scripts/build/website_post.py
diff -u $DEST/bib2texi.py $GIT/scripts/build/bib2texi.py
diff -u $DEST/lilypond.org.htaccess $GIT/Documentation/web/server/lilypond.org.htaccess
diff -u $DEST/website-dir.htaccess $GIT/Documentation/web/server/website-dir.htaccess
```

If the changes look ok, make them trusted:

```
### copy-from-git.sh
#!/bin/sh
GIT=$HOME/lilypond/lilypond-git
DEST=$HOME/lilypond/trusted-scripts
cp $GIT/make/website.make $DEST/website.make
cp $GIT/Documentation/lilypond-texi2html.init $DEST/lilypond-texi2html.init
cp $GIT/scripts/build/extract_texi_filenames.py $DEST/extract_texi_filenames.py
cp $GIT/scripts/build/create-version-itexi.py $DEST/create-version-itexi.py
cp $GIT/scripts/build/create-weblinks-itexi.py $DEST/create-weblinks-itexi.py
cp $GIT/scripts/build/mass-link.py $DEST/mass-link.py
cp $GIT/scripts/build/website_post.py $DEST/website_post.py
cp $GIT/scripts/build/bib2texi.py $DEST/bib2texi.py
cp $GIT/Documentation/web/server/lilypond.org.htaccess $DEST/lilypond.org.htaccess
cp $GIT/Documentation/web/server/website-dir.htaccess $DEST/website-dir.htaccess
```

Build the website:

```
### make-website.sh
#!/bin/sh
DEST=$HOME/web/
BUILD=$HOME/lilypond/build-website
mkdir -p $BUILD
cd $BUILD
cp $HOME/lilypond/trusted-scripts/website.make .
```

```
make -f website.make WEBSITE_ONLY_BUILD=1 website
rsync -raO $BUILD/out-website/website/ $DEST/website/
cp $BUILD/out-website/pictures $DEST
cp $BUILD/out-website/.htaccess $DEST
```

Cronjob to automate the trusted portions:

```
# website-rebuild.cron
11 * * * * $HOME/lilypond/trusted-scripts/update-git.sh >/dev/null 2>&1
22 * * * * $HOME/lilypond/trusted-scripts/make-website.sh >/dev/null 2>&1
```

Additional information

Some information about the website is stored in '`~graham/lilypond/*.txt`'; this information should not be shared with people without trusted access to the server.

6.3 Debugging website and docs locally

- Install apache2, or any other http server. These instructions assume that you also enable `mod_userdir`, and use `$HOME/public_html` as the location.
- Build the online docs and website:

  ```
  make WEB_TARGETS="offline online" doc
  make website
  ```

- Move the built stuff into those directories. It's highly recommended to have your build dir and www dir on the same partition. (make `$HOME/public_html/` a symlink if necessary)

  ```
  mv out-website/website/ $HOME/public_html
  mv $HOME/public_html/website/pictures $HOME/public_html/
  mkdir -p $HOME/public_html/doc/v2.13/
  mv out-www/online-root/* $HOME/public_html/doc/v2.13/
  ```

6.4 Translating the website

As it has much more audience, the website should be translated before the documentation; see Section 5.8 [Translating the documentation], page 59.

In addition to the normal documentation translation practices, there are a few additional things to note:

- Build the website with:

  ```
  make website
  ```

  however, please note that this command is not designed for being run multiple times. If you see unexpected output (mainly the page footers getting all messed up), then delete your '`out-website`' directory and run `make website` again.
- Some of the translation infrastructure is defined in python files; you must look at the `### translation data` sections in:

  ```
  scripts/build/create-weblinks-itexi.py
  scripts/build/website_post.py
  ```

- Translations are not included by default in `make website`. To test your translation, edit the `WEB_LANGS` line in '`make/website.make`'. Do not submit a patch to add your language to this file unless `make website` completes with less than 5 warnings.
- Links to manuals are done with macros like `@manualDevelLearningSplit`. To get translated links, you must change that to `@manualDevelLearningSplit-es` (for es/Spanish translations, for example).

7 LSR work

7.1 Introduction to LSR

The LilyPond Snippet Repository (LSR) is a collection of lilypond examples. A subset of these examples are automatically imported into the documentation, making it easy for users to contribute to the docs without learning Git and Texinfo.

7.2 Adding and editing snippets

General guidelines

When you create (or find!) a nice snippet, if it supported by LilyPond version running on LSR, please add it to LSR. Go to LSR and log in – if you haven't already, create an account. Follow the instructions on the website. These instructions also explain how to modify existing snippets.

If you think the snippet is particularly informative and you think it should be included in the documentation, tag it with "docs" and one or more other categories, or ask somebody who has editing permissions to do it on the development list.

Please make sure that the lilypond code follows the guidelines in Section 5.3.4 [LilyPond formatting], page 48.

If a new snippet created for documentation purposes compiles with LilyPond version currently on LSR, it should be added to LSR, and a reference to the snippet should be added to the documentation.

If the new snippet uses new features that are not available in the current LSR version, the snippet should be added to '`Documentation/snippets/new`' and a reference should be added to the manual.

Snippets created or updated in '`Documentation/snippets/new`' should be copied to '`Documentation/snippets`' by invoking at top of the source tree

```
scripts/auxiliar/makelsr.py
```

This also copies translated texidoc fields and snippet titles into snippets in '`Documentation/snippets`'. Note: this, in turn, could make the translated texidoc fields to appear as out of sync when you run `make check-translation`, if the originals changed from the last translation update, even if the translations are also updated; see Section 5.8.3 [Documentation translation maintenance], page 65 for details about updating the docs; in particular, see [Updating translation committishes], page 66 to learn how to mark these translated fields as fully updated.

Be sure that `make doc` runs successfully before submitting a patch, to prevent breaking compilation.

Formatting snippets in '`Documentation/snippets/new`'

When adding a file to this directory, please start the file with

```
\version "2.x.y"
\header {
% Use existing LSR tags other than 'docs'; see makelsr.py for
% the list of tags used to sort snippets.  E.g.:
  lsrtags = "rhythms,expressive-marks"
% This texidoc string will be formatted by Texinfo
  texidoc = "
This code demonstrates ...
"
```

```
% Please put doctitle last so that the '% begin verbatim'
% mark will be added correctly by makelsr.py.
  doctitle = "Snippet title"
}
```

\noindent and name the file '`snippet-title.ly`'.

7.3 Approving snippets

The main task of LSR editors is approving snippets. To find a list of unapproved snippets, log into LSR and select "No" from the dropdown menu to the right of the word "Approved" at the bottom of the interface, then click "Enable filter".

Check each snippet:

1. Does the snippet make sense and does what the author claims that it does? If you think the snippet is particularly helpful, add the "docs" tag and at least one other tag.
2. If the snippet is tagged with "docs", check to see if it matches our guidelines for Section 5.3.4 [LilyPond formatting], page 48.

   Also, snippets tagged with "docs" should not be explaining (replicating) existing material in the docs. They should not refer to the docs; the docs should refer to them.
3. If the snippet uses scheme, check that everything looks good and there are no security risks.

> **Note:** Somebody could sneak a `#'(system "rm -rf /")` command into our source tree if you do not do this! Take this step **VERY SERIOUSLY**.

7.4 LSR to Git

1. Make sure that `convert-ly` and `lilypond` commands in current PATH are in a bleeding edge version – latest release from master branch, or even better a fresh snapshot from Git master branch.
2. From the top source directory, run:

   ```
   wget http://lsr.dsi.unimi.it/download/lsr-snippets-docs-YYYY-MM-DD.tar.gz
   tar -xzf lsr-snippets-docs-YYYY-MM-DD.tar.gz
   scripts/auxiliar/makelsr.py lsr-snippets-docs-YYYY-MM-DD
   ```

   where *YYYY-MM-DD* is the current date, e.g. 2009-02-28.
3. Follow the instructions printed on the console to manually check for unsafe files.

> **Note:** Somebody could sneak a `#'(system "rm -rf /")` command into our source tree if you do not do this! Take this step **VERY SERIOUSLY**.

4. Do a git add / commit / push.

Note that whenever there is one snippet from '`Documentation/snippets/new`' and the other from LSR with the same file name, the one from '`Documentation/snippets/new`' will be copied by `makelsr.py`.

7.5 Fixing snippets in LilyPond sources

In case some snippet from '`Documentation/snippets`' causes the documentation compilation to fail, the following steps should be followed to fix it reliably.

1. Look up the snippet filename '*foo*`.ly`' in the error output or log, then fix the file '`Documentation/snippets/`*foo*`.ly`' to make the documentation build successfully.
2. Determine where it comes from by looking at its first line, e.g. run

   ```
   head -1 Documentation/snippets/foo.ly
   ```
3. **In case the snippet comes from LSR**, apply the fix to the snippet in LSR and send a notification email to a LSR editor with CC to the development list – see Section 7.2 [Adding and editing snippets], page 74. The failure may sometimes not be caused by the snippet in LSR but by the syntax conversion made by `convert-ly`; in this case, try to fix `convert-ly` or report the problem on the development list, then run `makelsr.py` again, see Section 7.4 [LSR to Git], page 75. In some cases, when some features has been introduced or vastly changed so it requires (or takes significant advantage of) important changes in the snippet, it is simpler and recommended to write a new version of the snippet in '`Documentation/snippets/new`', then run `makelsr.py`.
4. **In case the snippet comes from** '`Documentation/snippets/new`', apply in '`Documentation/snippets/new/`*foo*`.ly`' the same fix you did in '`Documentation/snippets/`*foo*`.ly`'. In case the build failure was caused by a translation string, you may have to fix '`input/texidocs/`*foo*`.texidoc`' instead.
5. In any case, commit all changes to Git.

7.6 Renaming a snippet

Due to the potential duality of snippets (i.e. they may exist both in the LSR database, and in `Documentation/snippets/new/`), this process is a bit more involved than we might like.

1. Send an email LSR editor, requesting the renaming.
2. The LSR editor does the renaming (or debates the topic with you), then warns the LSR-to-git person (wanted: better title) about the renaming.
3. LSR-to-git person does his normal job, but then also renames any copies of the snippets in `Documentation/snippets/new/`, and any instances of the snippet name in the documentation.

   `git grep` is highly recommended for this task.

7.7 Updating LSR to a new version

To update LSR, perform the following steps:

1. Download the latest snippet tarball, extract it, and run `convert-ly` on all files using the command-line option `--to=VERSION` to ensure snippets are updated to the correct stable version.
2. Copy relevant snippets (i.e., snippets whose version is equal to or less than the new version of LilyPond) from '`Documentation/snippets/new/`' into the tarball.

   You must not rename any files during this, or the next, stage.
3. Verify that all files compile with the new version of LilyPond, ideally without any warnings or errors. To ease the process, you may use the shell script that appears after this list.

   Due to the workload involved, we *do not* require that you verify that all snippets produce the expected output. If you happen to notice any such snippets and can fix them, great; but as long as all snippets compile, don't delay this step due to some weird output. If a snippet is broken, the hordes of willing web-2.0 volunteers will fix it. It's not our problem.
4. Create a tarball and send it back to Sebastiano.
5. When LSR has been updated, download another snippet tarball, verify that the relevant snippets from '`Documentation/snippets/new/`' were included, then delete those snippets from '`Documentation/snippets/new/`'.

Here is a shell script to run all '`.ly`' files in a directory and redirect terminal output to text files, which are then searched for the word "failed" to see which snippets do not compile.

```
#!/bin/bash

for LILYFILE in *.ly
do
  STEM=$(basename "$LILYFILE" .ly)
  echo "running $LILYFILE..."
  lilypond --format=png -ddelete-intermediate-files "$LILYFILE" >& "$STEM".txt
done

grep failed *.txt
```

8 Issues

This chapter deals with defects, feature requests, and miscellaneous development tasks.

8.1 Introduction to issues

> **Note:** Unless otherwise specified, all the tasks in this chapter are "simple" tasks: they can be done by a normal user with nothing more than a web browser, email, and lilypond.

"Issues" isn't just a politically-correct term for "bug". We use the same tracker for feature requests and code TODOs, so the term "bug" wouldn't be accurate. Despite the difference between "issue" and "bug", we call our team of contributors who organize issues the *Bug Squad*.

The Bug Squad is mainly composed of non-programmers – their job is to *organize* issues, not solve them. Their duties include removing false bug reports, ensuring that any real bug report contains enough information for developers, and checking that a developer's fix actually resolves the problem.

New volunteers for the Bug Squad should contact the Section 13.2 [Meisters], page 128.

8.2 Bug Squad setup

We highly recommend that you configure your email to use effective sorting; this can reduce your workload *immensely*. The email folders names were chosen specifically to make them work if you sort your folders alphabetically.

1. Read every section of this chapter, Chapter 8 [Issues], page 78.
2. If you do not have one already, create a gmail account and send the email address to the Section 13.2 [Meisters], page 128.
3. Subscribe your gmail account to `bug-lilypond`.
4. Configure your google code account:
   1. Wait until your gmail account is listed in:

      `http://code.google.com/p/lilypond/people/list`
   2. Sign in to google code by clicking in the top-right corner of:

      `http://code.google.com/p/lilypond/issues/list`

      You cannot log if you have Google Sharing `http://www.googlesharing.net/` enabled.
   3. Go to your "Profile", and select "Settings".
   4. Scroll down to "Issue change notification", and make sure that you have *selected* "If I starred the issue".
5. Configure your email client:
   1. Any email sent with your gmail address in the `To:` or `CC:` fields should go to a `bug-answers` folder.

      When setting up your filtering rules, be aware that Google Code might use different versions of your email address, such as ones ending in `@googlemail.com` or `@gmail.com`.
   2. Any other email either from, or CC'd to,

      `lilypond@googlecode.com`

      should go into a separate `bug-ignore` folder. Alternately, you may automatically delete these emails.

      You will **not read** these emails as part of your Bug Squad duties. If you are curious, go ahead and read them later, but it does **not** count as Bug Squad work.

3. Any other email sent to (or CC'd to):

   `bug-lilypond`

   should go into a separate `bug-current` folder.

8.3 Bug Squad checklists

When you do Bug Squad work, start at the top of this page and work your way down. Stop when you've done 15 minutes.

Please use the email sorting described in Section 8.2 [Bug Squad setup], page 78. This means that (as Bug Squad members) you will only ever respond to emails sent or CC'd to the `bug-lilypond` mailing list.

Emails to you personally

You are not expected to work on Bug Squad matters outside of your 15 minutes, but sometimes a confused user will send a bug report (or an update to a report) to you personally. If that happens, please forward such emails to the `bug-lilypond` list so that the currently-active Bug Squad member(s) can handle the message.

Daily schedule

The Bug Meister is omitted from the daily schedule.

```
Sunday: Colin
Monday: Dmytro
Tuesday: James Bailey
Wednesday: Ralph
Thursday: Patrick
Friday: Urs
Saturday: Kieren
```

Emails to `bug-answers`

Some of these emails will be comments on issues that you added to the tracker.

If they are asking for more information, give the additional information.

- If the email says that the issue was classified in some other manner, read the rationale given and take that into account for the next issue you add.
- Otherwise, move them to your `bug-ignore` folder.

Some of these emails will be discussions about Bug Squad work; read those.

Emails to `bug-current`

Dealing with these emails is your main task. Your job is to get rid of these emails in the first method which is applicable:

1. If the email has already been handled by a Bug Squad member (i.e. check to see who else has replied to it), delete it.
2. If the email is a question about how to use LilyPond, reply with this response:

   ```
   For questions about how to use LilyPond, please read our
   documentation available from:
     http://lilypond.org/website/manuals.html
   or ask the lilypond-user mailing list.
   ```

3. If the email mentions "the latest git", or any version number that has not yet been officially released, forward it to `lilypond-devel`.

4. If a bug report is not in the form of a Tiny example, direct the user to resubmit the report with this response:

   ```
   I'm sorry, but due to our limited resources for handling bugs, we
   can only accept reports in the form of Tiny examples.  Please see
   step 2 in our bug reporting guidelines:
     http://lilypond.org/website/bug-reports.html
   ```

5. If anything is unclear, ask the user for more information.

   How does the graphical output differ from what the user expected? What version of lilypond was used (if not given) and operating system (if this is a suspected cause of the problem)? In short, if you cannot understand what the problem is, ask the user to explain more. It is the user's responsibility to explain the problem, not your responsibility to understand it.

6. If the behavior is expected, the user should be told to read the documentation:

   ```
   I believe that this is the expected behaviour -- please read our
   documentation about this topic.  If you think that it really is a
   mistake, please explain in more detail.  If you think that the
   docs are unclear, please suggest an improvement as described by
   \Simple tasks -- Documentation"
    on:
     http://lilypond.org/website/help-us.html
   ```

7. If the issue already exists in the tracker, send an email to that effect:

   ```
   This issue has already been reported; you can follow the
   discussion and be notified about fixes here:
   ```

   (copy+paste the google code issue URL)

8. Accept the report as described in Section 8.5 [Adding issues to the tracker], page 83.

All emails should be CC'd to the `bug-lilypond` list so that other Bug Squad members know that you have processed the email.

> **Note:** There is no option for "ignore the bug report" – if you cannot find a reason to reject the report, you must accept it.

Regular maintenance

After **every release** (both stable and unstable):

- Regression test comparison: if anything has changed suspiciously, ask if it was deliberate. The official comparison is online, at:

  ```
  http://lilypond.org/test/
  ```

  More information is available from in Section 9.2 [Precompiled regression tests], page 86.

- Issues to verify: try to reproduce the bug with the latest official GUB version; if you cannot reproduce the bug, mark the item "Verified" (i.e. "the fix has been verified to work").

  ```
  http://code.google.com/p/lilypond/issues/list?can=7
  ```

  A few (approximately 10%) of these fixed issues relate to the build system or fundamental architecture changes; there is no way for you to verify these. Leave those issues alone; somebody else will handle them.

- Check for any incorrectly-classified items in the tracker. This generally just means looking at the grid to see any items without a Type or Priority.

8.4 Issue classification

The Bug Squad should classify issues according to the guidelines given by developers. Every issue should have a Status, Type, and Priority; the other fields are optional.

Status (mandatory)

Open issues:

- New: the item was added by a non-member, despite numerous warnings not to do this. Should be reviewed by a member of the Bug Squad.
- Accepted: the Bug Squad added it, or reviewed the item.
- Started: a contributor is working on a fix. Owner should change to be this contributor.

Closed issues:

- Invalid: issue should not have been added in the current state.
- Duplicate: issue already exists in the tracker.
- Fixed: a contributor claims to have fixed the bug. The Bug Squad should check the fix with the next official binary release (not by compiling the source from git). Owner should be set to that contributor.
- Verified: Bug Squad has confirmed that the issue is closed. This means that nobody should ever need look at the report again – if there is any information in the issue that should be kept, open a new issue for that info.

Owner (optional)

Newly-added issues should have *no owner*. When a contributor indicates that he has Started or Fixed an item, he should become the owner.

Type (mandatory)

The issue's Type should be the first relevant item in this list.

- Type-Collision: overlapping notation.
- Type-Defect: a problem in the core program. (the `lilypond` binary, scm files, fonts, etc).
- Type-Documentation: inaccurate, missing, confusing, or desired additional info. Must be fixable by editing a texinfo, ly, or scm file.
- Type-Build: problem or desired features in the build system. This includes the makefiles, stepmake, python scripts, and GUB.
- Type-Scripts: problem or desired feature in the non-build-system scripts. Mostly used for convert-ly, lilypond-book, etc.
- Type-Enhancement: a feature request for the core program. The distinction between enhancement and defect isn't extremely clear; when in doubt, mark it as enhancement.
- Type-Other: anything else.

Priority (mandatory)

Currently, only Critical items will block a stable release.

- Priority-Critical: LilyPond segfaults, a regression (see below) against a previous stable version or a regression against a fix developed for this version. This does not apply where the "regression" occurred because a feature was removed deliberately - this is not a bug.
- Priority-High: An issue which produces output which does not accurately reflect the input (e.g. where the user would expect an accidental, but none is shown) or which produces aesthetically poor output in a situation which could be expected to crop up frequently in real-world music. It should not be used where the problem can be avoided with a simple

workaround. It can also be used to flag where new code in a development version is not functioning as it should. This level is also used for issues which produce no output and fail to give the user a clue about what's wrong.

- Priority-Medium: Normal priority - use this as the default.
- Priority-Low: A minor problem which produces slightly undesirable output, or which will only occur in contrived examples, or which is very easily worked around.
- Priority-Postponed: no fix planned. Generally used for things which nobody wants to touch.

Note that these are initial classifications and can be subject to change by others in the development team. For example, a regression against an old stable version which hasn't been noticed for a long time and which is unlikely to get fixed could be downgraded from Priority-Critical by one of the programmers.

Opsys (optional)

Issues that only affect specific operating systems.

Patch (optional)

Normal Bug Squad members should not add or modify Patch issues; leave them to the Patch Meister.

- Patch-new: the patch has not been checked for "obvious" mistakes. When in doubt, use this tag.
- Patch-review: the patch has no "obvious" mistakes (as checked by the Patch Meister), and is ready for review from main developers.

  Developers with git push ability can use this category, skipping over `patch-new`.
- Patch-needs_work: a developer has some concerns about the patch. This does not necessarily mean that the patch must be changed; in some cases, the developer's concerns can be resolved simply by discussion the situation or providing notation examples.

  If the patch is updated, the category should be changed to `patch-new` (for normal contributors) or `patch-new` (for developers who are very confident about their patch).
- Patch-abandoned: the author has not responded to review comments for a few months.

Other items (optional)

Other labels:

- Regression: it used to work intentionally in an earlier stable release. If the earlier output was accidental (i.e. we didn't try to stop a collision, but it just so happened that two grobs didn't collide), then breaking it does not count as a regression.

  To help decide whether the change is a regression, and therefore should be Priority-Critical, please adopt the following process:

  1. Are you certain the change is OK? If so, do nothing.
  2. Are you certain that the change is bad? Add it to the tracker as a Critical issue, regression.
  3. If you're not certain either way, add it to the tracker as a Critical issue, regression but be aware that it may be recategorised or marked invalid.

  In particular, anything that breaks a regression test is a regression.
- Frog: the fix is believed to be suitable for a new contributor (does not require a great deal of knowledge about LilyPond). The issue should also have an estimated time in a comment.
- Maintainability: hinders development of LilyPond. For example, improvements to the build system, or "helper" python scripts.

- Bounty: somebody is willing to pay for the fix. Only add this tag if somebody has offered an exact figure in US dollars or euros.
- Warning: graphical output is fine, but lilypond prints a false/misleading warning message. Alternately, a warning should be printed (such as a bar line error), but was not. Also applies to warnings when compiling the source code or generating documentation.
- Security: might potentially be used.
- Performance: might potentially be used.

If you particularly want to add a label not in the list, go ahead, but this is not recommended.

8.5 Adding issues to the tracker

> **Note:** This should only be done by the Bug Squad or experienced developers. Normal users should not do this; instead, they should follow the guidelines for Section "Bug reports" in *General Information*.

In order to assign labels to issues, Bug Squad members should log in to their google account before adding an item.

1. Check if the issue falls into any previous category given on the relevant checklists in Section 8.3 [Bug Squad checklists], page 79. If in doubt, add a new issue for a report. We would prefer to have some incorrectly-added issues rather than lose information that should have been added.
2. Add the issue and classify it according to the guidelines in Section 8.4 [Issue classification], page 81. In particular, the item should have `Status`, `Type-`, and `Priority-` labels.

   Include output with the first applicable method:

   - If the issue has a notation example which fits in one system, generate a small '`bug.preview.png`' file with:

     ```
     lilypond -dpreview bug.ly
     ```

   - If the issue has an example which requires more than one system (i.e. a spacing bug), generate a '`bug.png`' file with:

     ```
     lilypond --png bug.ly
     ```

   - If the issue requires one or two pages of output, then generate a '`bug.png`' file with the normal:

     ```
     lilypond --png bug.ly
     ```

   - If the issue cannot be shown with less than three pages, then generate a '`bug.pdf`' file with:

     ```
     lilypond --pdf bug.ly
     ```

     Note that this is likely to be extremely rare; most bugs should fit into the first two categories above.
3. After adding the issue, please send a response email to the same group(s) that the initial patch was sent to. If the initial email was sent to multiple mailing lists (such as both `user` and `bugs`), then reply to all those mailing lists as well. The email should contain a link to the issue you just added.

8.6 Patch handling

> **Note:** This is not a Bug Squad responsibility; we have a separate person handling this task.

There is a single Patch Meister, and a number of Patch Helpers (rename this?). The list of known patches awaiting review is:

http://code.google.com/p/lilypond/issues/list?can=2&q=label:patch&sort=patch

Helpers: adding patches

The primary duty is to add patches to the google tracker; we have a bad track record of losing patches in email. Patches generally come to the `lilypond-devel` mailing list, but are sometimes sent to `bug-lilypond`, `lilypond-users`, or `frogs` mailing list instead.

- Unless a patch is clearly in response to an existing issue, add a new issue with the `Patch-new` label and a link to the patch (either on the mailing list archives or the codereview url).

  Issue numbers are cheap; losing developers because they got fed up with us losing their hard work is expensive.
- If the patch is clearly in response to an existing issue, then update that issue with the `Patch-new` label and a link to the patch (either on the mailing list archives or the codereview url).
- After adding the issue, please send a response email to the same group(s) that the initial patch was sent to.

  If the initial email was sent to multiple mailing lists (such as both `bugs` and `devel`), then reply to all those mailing lists as well. The email should contain a link to the issue you just added.

Helpers: `Patch-review` label

The secondary duty is to do make sure that every issue in the tracker with a `Patch-review` label has passed these "obvious" tests:

- Applies automatically to git master.

  It's ok to have offsets, but not conflicts.
- Regtest comparison looks ok; no unexpected changes.
- Descriptive subject line.

  Avoid subjects like "fixes 123"; instead write "Doc: discuss stacking-dir for BassFigure-Alignment (fix 123)".
- Compiles docs from scratch. Only check this if you have reason to suspect it might not work.
- (maybe)

  Check code indentation and style. This should be easier post-GOP when we have a better-defined code style.

Patch Meister

The Patch Meister will:

- send "countdown" emails to `lilypond-devel` when patches appear to be ready.
- send general requests to review patches, or even nasty requests to review patches.
- downgrade patches from `Patch-review` to `Patch-needs_work` as appropriate.
- downgrade patches from `Patch-needs_work` to `Patch-abandoned` if no actions have been taken in four weeks.

8.7 Summary of project status

Project overview

Grid view provides the best overview:

http://code.google.com/p/lilypond/issues/list?mode=grid&y=Priority&x=Type&cells=ids

Hindering development

These issues stop or slow development work:

http://code.google.com/p/lilypond/issues/list?can=2&q=label:Maintainability&mode=grid&y

Easy tasks

Issues tagged with `Frog` indicates a task suitable for a relatively new contributor. The time given is a quick (inaccurate) estimate of the time required for somebody who is familiar with material in this manual, but does not know anything else about LilyPond development.

http://code.google.com/p/lilypond/issues/list?can=2&q=label:Frog&mode=grid&y=Priority&x

Patches to review

Patches which have no “obvious” problems:

http://code.google.com/p/lilypond/issues/list?can=2&q=label:patch-review

9 Regression tests

9.1 Introduction to regression tests

LilyPond has a complete suite of regression tests that are used to ensure that changes to the code do not break existing behavior. These regression tests comprise small LilyPond snippets that test the functionality of each part of LilyPond.

Regression tests are added when new functionality is added to LilyPond. We do not yet have a policy on when it is appropriate to add or modify a regtest when bugs are fixed. Individual developers should use their best judgement until this is clarified during the Section 13.4 [Grand Organization Project (GOP)], page 129.

The regression tests are compiled using special `make` targets. There are three primary uses for the regression tests. First, successful completion of the regression tests means that LilyPond has been properly built. Second, the output of the regression tests can be manually checked to ensure that the graphical output matches the description of the intended output. Third, the regression test output from two different versions of LilyPond can be automatically compared to identify any differences. These differences should then be manually checked to ensure that the differences are intended.

Regression tests ("regtests") are available in precompiled form as part of the documentation. Regtests can also be compiled on any machine that has a properly configured LilyPond build system.

9.2 Precompiled regression tests

Regression test output

As part of the release process, the regression tests are run for every LilyPond release. Full regression test output is available for every stable version and the most recent development version.

Regression test output is available in HTML and PDF format. Links to the regression test output are available at the developer's resources page for the version of interest.

The latest stable version of the regtests is found at:

`http://lilypond.org/doc/stable/input/regression/collated-files.html`

The latest development version of the regtests is found at:

`http://lilypond.org/doc/latest/input/regression/collated-files.html`

Regression test comparison

Each time a new version is released, the regtests are compiled and the output is automatically compared with the output of the previous release. The result of these comparisons is archived online:

`http://lilypond.org/test/`

Checking these pages is a very important task for the LilyPond project. You are invited to report anything that looks broken, or any case where the output quality is not on par with the previous release, as described in Section "Bug reports" in *General Information*.

> **Note:** The special regression test '`test-output-distance.ly`' will always show up as a regression. This test changes each time it is run, and serves to verify that the regression tests have, in fact, run.

What to look for

The test comparison shows all of the changes that occurred between the current release and the prior release. Each test that has a significant difference in output is displayed, with the old version on the left and the new version on the right.

Regression tests whose output is the same for both versions are not shown in the test comparison.

- Images: green blurs in the new version show the approximate location of elements in the old version.

  There are often minor adjustments in spacing which do not indicate any problem.
- Log files: show the difference in command-line output.

  The main thing to examine are any changes in page counts – if a file used to fit on 1 page but now requires 4 or 5 pages, something is suspicious!
- Profile files: give information about TODO? I don't know what they're for.

> **Note:** The automatic comparison of the regtests checks the LilyPond bounding boxes. This means that Ghostscript changes and changes in lyrics or text are not found.

9.3 Compiling regression tests

Developers may wish to see the output of the complete regression test suite for the current version of the source repository between releases. Current source code is available; see Chapter 3 [Working with source code], page 11. Then you will need to build the LilyPond binary; see Section 4.5 [Compiling LilyPond], page 38.

Uninstalling the previous LilyPond version is not necessary, nor is running `make install`, since the tests will automatically be compiled with the LilyPond binary you have just built in your source directory.

From this point, the regtests are compiled with:

```
make test
```

If you have a multi-core machine you may want to use the '`-j`' option and *CPU_COUT* variable, as described in [Saving time with CPU_COUNT], page 40. For a quad-core processor the complete command would be:

```
make -j5 CPU_COUNT=5 test
```

The regtest output will then be available in '`input/regression/out-test`'. '`input/regression/out-test/collated-examples.html`' contains a listing of all the regression tests that were run, but none of the images are included. Individual images are also available in this directory.

The primary use of '`make test`' is to verify that the regression tests all run without error. The regression test page that is part of the documentation is created only when the documentation is built, as described in Section 4.6.2 [Generating documentation], page 39. Note that building the documentation requires more installed components than building the source code, as described in Section 4.2.3 [Requirements for building documentation], page 35.

9.4 Regtest comparison

Before modified code is committed to master, a regression test comparison must be completed to ensure that the changes have not caused problems with previously working code. The comparison is made automatically upon compiling the regression test suite twice.

1. Before making changes, a baseline should be established by running:

   ```
   make test-baseline
   ```

2. Make your changes, or apply the patch(es) to consider.
3. Compile the source with '`make`' as usual.
4. Check for unintentional changes to the regtests:

   ```
   make check
   ```

   After this has finished, a regression test comparison will be available at:

   ```
   out/test-results/index.html
   ```

   For each regression test that differs between the baseline and the changed code, a regression test entry will displayed. Ideally, the only changes would be the changes that you were working on. If regressions are introduced, they must be fixed before committing the code.

   > **Note:** The special regression test '`test-output-distance.ly`' will always show up as a regression. This test changes each time it is run, and serves to verify that the regression tests have, in fact, run.

5. If you are happy with the results, then stop now.

   If you want to continue programming, then make any additional code changes, and continue.

6. Compile the source with '`make`' as usual.
7. To re-check files that differed between the initial '`make test-baseline`' and your post-changes '`make check`', run:

   ```
   make test-redo
   ```

   This updates the regression list at '`out/test-results/index.html`'. It does *not* redo '`test-output-distance.ly`'.

8. When all regressions have been resolved, the output list will be empty.
9. Once all regressions have been resolved, a final check should be completed by running:

   ```
   make test-clean
   make check
   ```

   This cleans the results of the previous '`make check`', then does the automatic regression comparison again.

9.5 Finding the cause of a regression

Git has special functionality to help tracking down the exact commit which causes a problem. See the git manual page for `git bisect`. This is a job that non-programmers can do, although it requires familiarity with git, ability to compile LilyPond, and generally a fair amount of technical knowledge. A brief summary is given below, but you may need to consult other documentation for in-depth explanations.

Even if you are not familiar with git or are not able to compile LilyPond you can still help to narrow down the cause of a regression simply by downloading the binary releases of different LilyPond versions and testing them for the regression. Knowing which version of LilyPond first exhibited the regression is helpful to a developer as it shortens the `git bisect` procedure.

Once a problematic commit is identified, the programmers' job is much easier. In fact, for most regression bugs, the majority of the time is spent simply finding the problematic commit.

More information is in Chapter 9 [Regression tests], page 86.

git bisect setup

We need to set up the bisect for each problem we want to investigate.

Suppose we have an input file which compiled in version 2.13.32, but fails in version 2.13.38 and above.

1. Begin the process:

   ```
   git bisect start
   ```

2. Give it the earliest known bad tag:

   ```
   git bisect bad release/2.13.38-1
   ```

   (you can see tags with: `git tag`)

3. Give it the latest known good tag:

   ```
   git bisect good release/2.13.32-1
   ```

   You should now see something like:

   ```
   Bisecting: 195 revisions left to test after this (roughly 8 steps)
   [b17e2f3d7a5853a30f7d5a3cdc6b5079e77a3d2a] Web: Announcement
   update for the new \LilyPond Report"
   .
   ```

git bisect actual

1. Compile the source:

   ```
   make
   ```

2. Test your input file:

   ```
   out/bin/lilypond test.ly
   ```

3. Test results?

   - Does it crash, or is the output bad? If so:

     ```
     git bisect bad
     ```

   - Does your input file produce good output? If so:

     ```
     git bisect good
     ```

4. Once the exact problem commit has been identified, git will inform you with a message like:

   ```
   6d28aebbaaab1be9961a00bf15a1ef93acb91e30 is the first bad commit
   %%% ... blah blah blah ...
   ```

   If there is still a range of commits, then git will automatically select a new version for you to test. Go to step #1.

Recommendation: use two terminal windows

- One window is open to the `build/` directory, and alternates between these commands:

  ```
  make
  out/bin/lilypond test.ly
  ```

- One window is open to the top source directory, and alternates between these commands:

  ```
  git bisect good
  git bisect bad
  ```

9.6 Memory and coverage tests

In addition to the graphical output of the regression tests, it is possible to test memory usage and to determine how much of the source code has been exercised by the tests.

Memory usage

For tracking memory usage as part of this test, you will need GUILE CVS; especially the following patch: http://www.lilypond.org/vc/old/gub.darcs/patches/guile-1.9-gcstats.patch.

Code coverage

For checking the coverage of the test suite, do the following

```
./scripts/auxiliar/build-coverage.sh
# uncovered files, least covered first
./scripts/auxiliar/coverage.py  --summary out-cov/*.cc
# consecutive uncovered lines, longest first
./scripts/auxiliar/coverage.py  --uncovered out-cov/*.cc
```

9.7 MusicXML tests

LilyPond comes with a complete set of regtests for the MusicXML language. Originally developed to test '`musicxml2ly`', these regression tests can be used to test any MusicXML implementation.

The MusicXML regression tests are found at '`input/regression/musicxml/`'.

The output resulting from running these tests through '`muscxml2ly`' followed by '`lilypond`' is available in the LilyPond documentation:

http://lilypond.org/doc/latest/input/regression/musicxml/collated-files

10 Programming work

10.1 Overview of LilyPond architecture

LilyPond processes the input file into graphical and musical output in a number of stages. This process, along with the types of routines that accomplish the various stages of the process, is described in this section. A more complete description of the LilyPond architecture and internal program execution is found in Erik Sandberg's master's thesis.

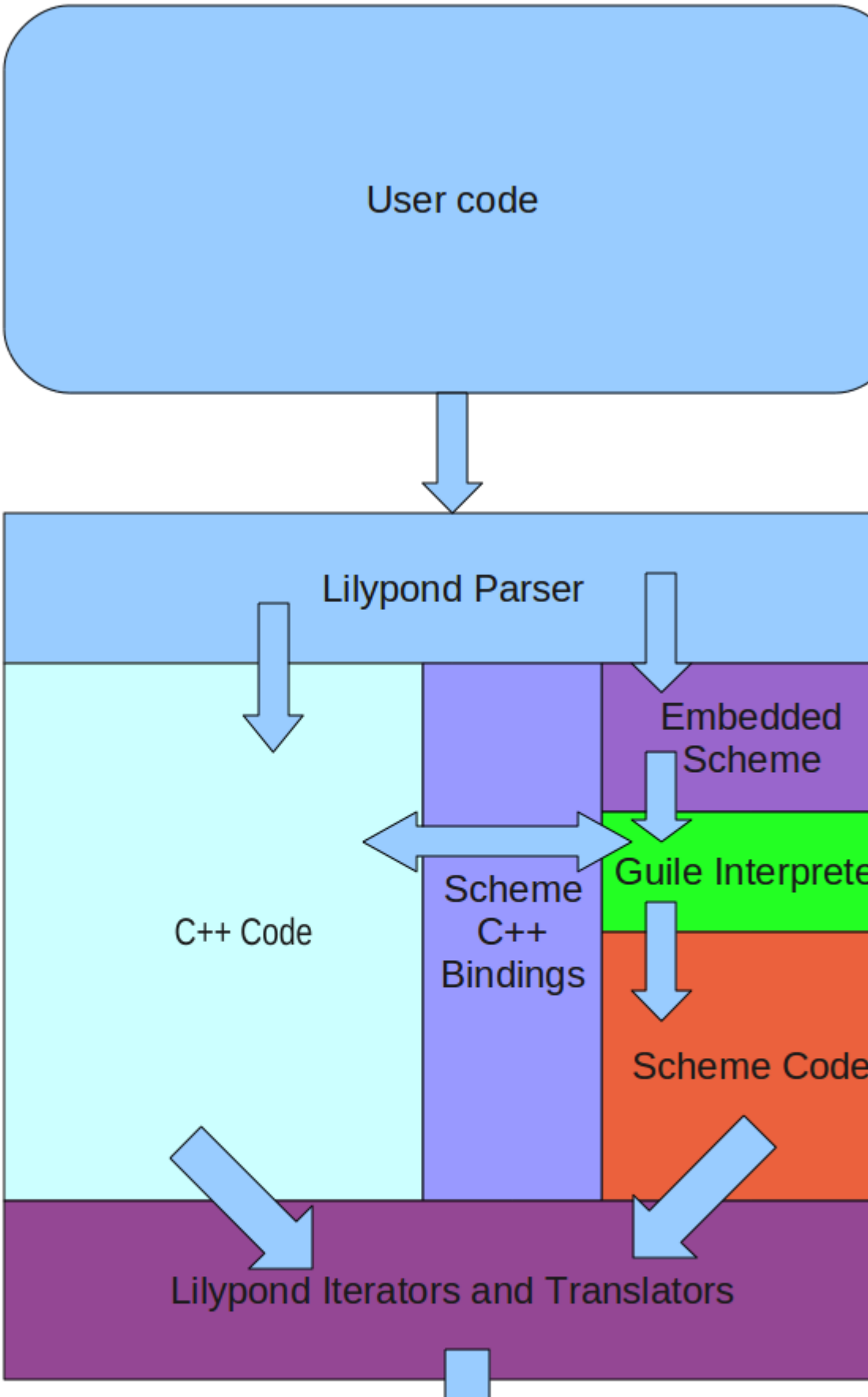
The first stage of LilyPond processing is *parsing*. In the parsing process, music expressions in LilyPond input format are converted to music expressions in Scheme format. In Scheme format, a music expression is a list in tree form, with nodes that indicate the relationships between various music events. The LilyPond parser is written in Bison.

The second stage of LilyPond processing is *iterating*. Iterating assigns each music event to a context, which is the environment in which the music will be finally engraved. The context is responsible for all further processing of the music. It is during the iteration stage that contexts are created as necessary to ensure that every note has a Voice type context (e.g. Voice, TabVoice, DrumVoice, CueVoice, MensuralVoice, VaticanaVoice, GregorianTranscriptionVoice), that the Voice type contexts exist in appropriate Staff type contexts, and that parallel Staff type contexts exist in StaffGroup type contexts. In addition, during the iteration stage each music event is assigned a moment, or a time in the music when the event begins.

Each type of music event has an associated iterator. Iterators are defined in '`*-iterator.cc`'. During iteration, an event's iterator is called to deliver that music event to the appropriate context(s).

The final stage of LilyPond processing is *translation*. During translation, music events are prepared for graphical or midi output. The translation step is accomplished by the polymorphic base class Translator through its two derived classes: Engraver (for graphical output) and Performer (for midi output).

Translators are defined in C++ files named '`*-engraver.cc`' and '`*-performer.cc`'. Much of the work of translating is handled by Scheme functions, which is one of the keys to LilyPond's exceptional flexibility.



10.2 LilyPond programming languages

Programming in LilyPond is done in a variety of programming languages. Each language is used for a specific purpose or purposes. This section describes the languages used and provides links to reference manuals and tutorials for the relevant language.

10.2.1 C++

The core functionality of LilyPond is implemented in C++.

C++ is so ubiquitous that it is difficult to identify either a reference manual or a tutorial. Programmers unfamiliar with C++ will need to spend some time to learn the language before attempting to modify the C++ code.

The C++ code calls Scheme/GUILE through the GUILE interface, which is documented in the GUILE Reference Manual.

10.2.2 Flex

The LilyPond lexer is implemented in Flex, an implementation of the Unix lex lexical analyser generator. Resources for Flex can be found here.

10.2.3 GNU Bison

The LilyPond parser is implemented in Bison, a GNU parser generator. The Bison homepage is found at gnu.org. The manual (which includes both a reference and tutorial) is available in a variety of formats.

10.2.4 GNU Make

GNU Make is used to control the compiling process and to build the documentation and the website. GNU Make documentation is available at the GNU website.

10.2.5 GUILE or Scheme

GUILE is the dialect of Scheme that is used as LilyPond's extension language. Many extensions to LilyPond are written entirely in GUILE. The GUILE Reference Manual is available online.

Structure and Interpretation of Computer Programs, a popular textbook used to teach programming in Scheme is available in its entirety online.

An introduction to Guile/Scheme as used in LilyPond can be found in the Section "Scheme tutorial" in *Extending*.

10.2.6 MetaFont

MetaFont is used to create the music fonts used by LilyPond. A MetaFont tutorial is available at the METAFONT tutorial page.

10.2.7 PostScript

PostScript is used to generate graphical output. A brief PostScript tutorial is available online. The PostScript Language Reference is available online in PDF format.

10.2.8 Python

Python is used for XML2ly and is used for building the documentation and the website.

Python documentation is available at python.org.

10.3 Programming without compiling

Much of the development work in LilyPond takes place by changing '`*.ly`' or '`*.scm`' files. These changes can be made without compiling LilyPond. Such changes are described in this section.

10.3.1 Modifying distribution files

Much of LilyPond is written in Scheme or LilyPond input files. These files are interpreted when the program is run, rather than being compiled when the program is built, and are present in all LilyPond distributions. You will find '`.ly`' files in the '`ly/`' directory and the Scheme files in the '`scm/`' directory. Both Scheme files and '`.ly`' files can be modified and saved with any text editor. It's probably wise to make a backup copy of your files before you modify them, although you can reinstall if the files become corrupted.

Once you've modified the files, you can test the changes just by running LilyPond on some input file. It's a good idea to create a file that demonstrates the feature you're trying to add. This file will eventually become a regression test and will be part of the LilyPond distribution.

10.3.2 Desired file formatting

Files that are part of the LilyPond distribution have Unix-style line endings (LF), rather than DOS (CR+LF) or MacOS 9 and earlier (CR). Make sure you use the necessary tools to ensure that Unix-style line endings are preserved in the patches you create.

Tab characters should not be included in files for distribution. All indentation should be done with spaces. Most editors have settings to allow the setting of tab stops and ensuring that no tab characters are included in the file.

Scheme files and LilyPond files should be written according to standard style guidelines. Scheme file guidelines can be found at `http://community.schemewiki.org/?scheme-style`. Following these guidelines will make your code easier to read. Both you and others that work on your code will be glad you followed these guidelines.

For LilyPond files, you should follow the guidelines for LilyPond snippets in the documentation. You can find these guidelines at Section 5.3 [Texinfo introduction and usage policy], page 46.

10.4 Finding functions

When making changes or fixing bugs in LilyPond, one of the initial challenges is finding out where in the code tree the functions to be modified live. With nearly 3000 files in the source tree, trial-and-error searching is generally ineffective. This section describes a process for finding interesting code.

10.4.1 Using the ROADMAP

The file ROADMAP is located in the main directory of the lilypond source. ROADMAP lists all of the directories in the LilyPond source tree, along with a brief description of the kind of files found in each directory. This can be a very helpful tool for deciding which directories to search when looking for a function.

10.4.2 Using grep to search

Having identified a likely subdirectory to search, the grep utility can be used to search for a function name. The format of the grep command is

```
grep -i functionName subdirectory/*
```

This command will search all the contents of the directory subdirectory/ and display every line in any of the files that contains functionName. The `-i` option makes `grep` ignore case – this can be very useful if you are not yet familiar with our capitalization conventions.

The most likely directories to grep for function names are '`scm/`' for scheme files, ly/ for lilypond input ('`*.ly`') files, and '`lily/`' for C++ files.

10.4.3 Using git grep to search

If you have used git to obtain the source, you have access to a powerful tool to search for functions. The command:

```
git grep functionName
```

will search through all of the files that are present in the git repository looking for functionName. It also presents the results of the search using `less`, so the results are displayed one page at a time.

10.4.4 Searching on the git repository at Savannah

You can also use the equivalent of git grep on the Savannah server.

- Go to http://git.sv.gnu.org/gitweb/?p=lilypond.git
- In the pulldown box that says commit, select grep.
- Type functionName in the search box, and hit enter/return

This will initiate a search of the remote git repository.

10.5 Code style

This section describes style guidelines for LilyPond source code.

10.5.1 Languages

C++ and Python are preferred. Python code should use PEP 8.

10.5.2 Filenames

Definitions of classes that are only accessed via pointers (*) or references (&) shall not be included as include files.

```
filenames

     ".hh"   Include files
          ".cc"      Implementation files
          ".icc"     Inline definition files
          ".tcc"     non inline Template defs

in emacs:

          (setq auto-mode-alist
                (append '(("\\.make$" . makefile-mode)
                     ("\\.cc$" . c++-mode)
                     ("\\.icc$" . c++-mode)
                     ("\\.tcc$" . c++-mode)
                     ("\\.hh$" . c++-mode)
                     ("\\.pod$" . text-mode)
                     )
                   auto-mode-alist))
```

The class Class_name is coded in ‘class-name.*’

10.5.3 Indentation

Standard GNU coding style is used. In emacs:

```
          (add-hook 'c++-mode-hook
               '(lambda() (c-set-style "gnu")
```

```
                 ))
```

If you like using font-lock, you can also add this to your '.emacs':

```
(setq font-lock-maximum-decoration t)
(setq c++-font-lock-keywords-3
      (append
       c++-font-lock-keywords-3
       '(("\\b\\(a-zA-Z_?+_\\)\\b" 1 font-lock-variable-name-face) ("\\b\\(A-Z
       ))
```

Some source files may not currently have proper indenting. If this is the case, it is desirable to fix the improper indenting when the file is modified, with the hope of continually improving the code.

Indenting files with fixcc.py

LilyPond provides a python script that will correct the indentation on a c++ file:

```
scripts/auxiliar/fixcc.py lily/my-test-file.cc
```

Be sure you replace '`my-test-file.cc`' with the name of the file that you edited.

If you are editing a file that contains an ADD_TRANSLATOR or ADD_INTERFACE macro, the fixcc.py script will move the final parenthesis up one line from where it should be. Please check the end of the file before you run fixcc.py, and then put the final parenthesis and semicolon back on a line by themselves.

Indenting files with emacs in script mode

> **Note:** this is pending some confirmation on -devel. July 2009 -gp

Command-line script to format stuff with emacs:

```
#!/bin/sh
emacs $1 -batch --eval '(indent-region (point-min) (point-max) nil)' -f save-buffer
```

(that's all on one line)

Save it as a shell script, then run on the file(s) you modified.

Indenting with vim

Although emacs indentation is the LilyPond standard, acceptable indentation can usually be accomplished with vim. Some hints for vim are as follows:

A workable .vimrc:

```
set cindent
set smartindent
set autoindent
set expandtab
set softtabstop=2
set shiftwidth=2
filetype plugin indent on
set incsearch
set ignorecase smartcase
set hlsearch
set confirm
set statusline=%F%m%r%h%w\ %{&ff}\ %Y\ [ASCII=\%03.3b]\ [HEX=\%02.2B]\ %04l,%04v\ %p%%\ [LEN
set laststatus=2
set number
```

```
" Remove trailing whitespace on write
autocmd BufWritePre * :%s/\s\+$//e
```

With this .vimrc, files can be reindented automatically by highlighting the lines to be indented in visual mode (use V to enter visual mode) and pressing =.

A scheme.vim file will help improve the indentation. This one was suggested by Patrick McCarty. It should be saved in ~/.vim/after/syntax/scheme.vim.

```
" Additional Guile-specific 'forms'
syn keyword schemeSyntax define-public define*-public
syn keyword schemeSyntax define* lambda* let-keywords*
syn keyword schemeSyntax defmacro defmacro* define-macro
syn keyword schemeSyntax defmacro-public defmacro*-public
syn keyword schemeSyntax use-modules define-module
syn keyword schemeSyntax define-method define-class

" Additional LilyPond-specific 'forms'
syn keyword schemeSyntax define-markup-command define-markup-list-command
syn keyword schemeSyntax define-safe-public define-music-function
syn keyword schemeSyntax def-grace-function

" All of the above should influence indenting too
set lw+=define-public,define*-public
set lw+=define*,lambda*,let-keywords*
set lw+=defmacro,defmacro*,define-macro
set lw+=defmacro-public,defmacro*-public
set lw+=use-modules,define-module
set lw+=define-method,define-class
set lw+=define-markup-command,define-markup-list-command
set lw+=define-safe-public,define-music-function
set lw+=def-grace-function

" These forms should not influence indenting
set lw-=if
set lw-=set!

" Try to highlight all ly: procedures
syn match schemeFunc "ly:[^) ]\+"
```

10.5.4 Naming Conventions

Naming conventions have been established for LilyPond source code.

Classes and Types

Classes begin with an uppercase letter, and words in class names are separated with _:

```
This_is_a_class
```

Members

Member variable names end with an underscore:

```
Type Class::member_
```

Macros

Macro names should be written in uppercase completely, with words separated by _:

```
THIS_IS_A_MACRO
```

Variables

Variable names should be complete words, rather than abbreviations. For example, it is preferred to use `thickness` rather than `th` or `t`.

Multi-word variable names in C++ should have the words separated by the underscore character ('_'):

```
cxx_multiword_variable
```

Multi-word variable names in Scheme should have the words separated by a hyphen ('-'):

```
scheme-multiword-variable
```

10.5.5 Broken code

Do not write broken code. This includes hardwired dependencies, hardwired constants, slow algorithms and obvious limitations. If you can not avoid it, mark the place clearly, and add a comment explaining shortcomings of the code.

Ideally, the comment marking the shortcoming would include TODO, so that it is marked for future fixing.

We reject broken-in-advance on principle.

10.5.6 Code comments

Comments may not be needed if descriptive variable names are used in the code and the logic is straightforward. However, if the logic is difficult to follow, and particularly if non-obvious code has been included to resolve a bug, a comment describing the logic and/or the need for the non-obvious code should be included.

There are instances where the current code could be commented better. If significant time is required to understand the code as part of preparing a patch, it would be wise to add comments reflecting your understanding to make future work easier.

10.5.7 Handling errors

As a general rule, you should always try to continue computations, even if there is some kind of error. When the program stops, it is often very hard for a user to pinpoint what part of the input causes an error. Finding the culprit is much easier if there is some viewable output.

So functions and methods do not return errorcodes, they never crash, but report a programming_error and try to carry on.

Error and warning messages need to be localized.

10.5.8 Localization

This document provides some guidelines to help programmers write proper user messages. To help translations, user messages must follow uniform conventions. Follow these rules when coding for LilyPond. Hopefully, this can be replaced by general GNU guidelines in the future. Even better would be to have an English (en_BR, en_AM) guide helping programmers writing consistent messages for all GNU programs.

Non-preferred messages are marked with '+'. By convention, ungrammatical examples are marked with '*'. However, such ungrammatical examples may still be preferred.

- Every message to the user should be localized (and thus be marked for localization). This includes warning and error messages.
- Do not localize/gettextify:
  - 'programming_error ()'s
  - 'programming_warning ()'s
  - debug strings

  - output strings (PostScript, TeX, etc.)
- Messages to be localized must be encapsulated in '_ (STRING)' or '_f (FORMAT, ...)'. E.g.:

```
warning (_ ("need music in a score"));
error (_f ("cannot open file: `%s'", file_name));
```

  In some rare cases you may need to call 'gettext ()' by hand. This happens when you pre-define (a list of) string constants for later use. In that case, you'll probably also need to mark these string constants for translation, using '_i (STRING)'. The '_i' macro is a no-op, it only serves as a marker for 'xgettext'.

```
char const* messages[] = {
  _i ("enable debugging output"),
  _i ("ignore lilypond version"),
  0
};

void
foo (int i)
{
  puts (gettext (messages i));
}
```

  See also '`flower/getopt-long.cc`' and '`lily/main.cc`'.
- Do not use leading or trailing whitespace in messages. If you need whitespace to be printed, prepend or append it to the translated message

```
message ("Calculating line breaks..." + " ");
```

- Error or warning messages displayed with a file name and line number never start with a capital, eg,

```
foo.ly: 12: not a duration: 3
```

  Messages containing a final verb, or a gerund ('-ing'-form) always start with a capital. Other (simpler) messages start with a lowercase letter

```
Processing foo.ly...
`foo': not declared.
Not declaring: `foo'.
```

- Avoid abbreviations or short forms, use 'cannot' and 'do not' rather than 'can't' or 'don't' To avoid having a number of different messages for the same situation, well will use quoting like this '"message: '%s'"' for all strings. Numbers are not quoted:

```
_f ("cannot open file: `%s'", name_str)
_f ("cannot find character number: %d", i)
```

- Think about translation issues. In a lot of cases, it is better to translate a whole message. English grammar must not be imposed on the translator. So, instead of

```
stem at  + moment.str () +  does not fit in beam
```

  have

```
_f ("stem at %s does not fit in beam", moment.str ())
```

- Split up multi-sentence messages, whenever possible. Instead of

```
warning (_f ("out of tune!  Can't find: `%s'", "Key_engraver"));
warning (_f ("cannot find font `%s', loading default", font_name));
```

  rather say:

```
warning (_ ("out of tune:"));
warning (_f ("cannot find: `%s', "Key_engraver"));
```

```
warning (_f ("cannot find font: `%s', font_name));
warning (_f ("Loading default font"));
```

- If you must have multiple-sentence messages, use full punctuation. Use two spaces after end of sentence punctuation. No punctuation (esp. period) is used at the end of simple messages.

```
_f ("Non-matching braces in text `%s', adding braces", text)
_ ("Debug output disabled.  Compiled with NPRINT.")
_f ("Huh?  Not a Request: `%s'.  Ignoring.", request)
```

- Do not modularize too much; words frequently cannot be translated without context. It is probably safe to treat most occurrences of words like stem, beam, crescendo as separately translatable words.
- When translating, it is preferable to put interesting information at the end of the message, rather than embedded in the middle. This especially applies to frequently used messages, even if this would mean sacrificing a bit of eloquency. This holds for original messages too, of course.

```
en: cannot open: `foo.ly'
+   nl: kan `foo.ly' niet openen (1)
kan niet openen: `foo.ly'*   (2)
niet te openen: `foo.ly'*    (3)
```

  The first nl message, although grammatically and stylistically correct, is not friendly for parsing by humans (even if they speak dutch). I guess we would prefer something like (2) or (3).
- Do not run make po/po-update with GNU gettext < 0.10.35

10.6 Debugging LilyPond

The most commonly used tool for debugging LilyPond is the GNU debugger gdb. The gdb tool is used for investigating and debugging core Lilypond code written in C++. Another tool is available for debugging Scheme code using the Guile debugger. This section describes how to use both gdb and the Guile Debugger.

10.6.1 Debugging overview

Using a debugger simplifies troubleshooting in at least two ways.

First, breakpoints can be set to pause execution at any desired point. Then, when execution has paused, debugger commands can be issued to explore the values of various variables or to execute functions.

Second, the debugger can display a stack trace, which shows the sequence in which functions have been called and the arguments passed to the called functions.

10.6.2 Debugging C++ code

The GNU debugger, gdb, is the principal tool for debugging C++ code.

Compiling LilyPond for use with gdb

In order to use gdb with LilyPond, it is necessary to compile LilyPond with debugging information. This is accomplished by running the following commands in the main LilyPond source directory.

```
./configure  --disable-optimising
make
```

This will create a version of LilyPond containing debugging information that will allow the debugger to tie the source code to the compiled code.

You should not do *make install* if you want to use a debugger with LilyPond. The *make install* command will strip debugging information from the LilyPond binary.

Typical gdb usage

Once you have compiled the Lilypond image with the necessary debugging information it will have been written to a location in a subfolder of your current working directory:

```
out/bin/lilypond
```

This is important as you will need to let gdb know where to find the image containing the symbol tables. You can invoke gdb from the command line using the following:

```
gdb out/bin/lilypond
```

This loads the LilyPond symbol tables into gdb. Then, to run LilyPond on '`test.ly`' under the debugger, enter the following:

```
run test.ly
```

at the gdb prompt.

As an alternative to running gdb at the command line you may try a graphical interface to gdb such as ddd:

```
ddd out/bin/lilypond
```

You can also use sets of standard gdb commands stored in a .gdbinit file (see next section).

Typical .gdbinit files

The behavior of gdb can be readily customized through the use of a *.gdbinit* file. A *.gdbinit* file is a file named *.gdbinit* (notice the "." at the beginning of the file name) that is placed in a user's home directory.

The *.gdbinit* file below is from Han-Wen. It sets breakpoints for all errors and defines functions for displaying scheme objects (ps), grobs (pgrob), and parsed music expressions (pmusic).

```
file lily/out/lilypond
b programming_error
b Grob::programming_error

define ps
   print ly_display_scm($arg0)
end
define pgrob
  print ly_display_scm($arg0->self_scm_)
  print ly_display_scm($arg0->mutable_property_alist_)
  print ly_display_scm($arg0->immutable_property_alist_)
  print ly_display_scm($arg0->object_alist_)
end
define pmusic
  print ly_display_scm($arg0->self_scm_)
  print ly_display_scm($arg0->mutable_property_alist_)
  print ly_display_scm($arg0->immutable_property_alist_)
end
```

10.6.3 Debugging Scheme code

Scheme code can be developed using the Guile command line interpreter `top-repl`. You can either investigate interactively using just Guile or you can use the debugging tools available within Guile.

Using Guile interactively with LilyPond

In order to experiment with Scheme programming in the LilyPond environment, it is necessary to have a Guile interpreter that has all the LilyPond modules loaded. This requires the following steps.

First, define a Scheme symbol for the active module in the '`.ly`' file:

```
#(module-define! (resolve-module '(guile-user))
                 'lilypond-module (current-module))
```

Now place a Scheme function in the '`.ly`' file that gives an interactive Guile prompt:

```
#(top-repl)
```

When the '`.ly`' file is compiled, this causes the compilation to be interrupted and an interactive guile prompt to appear. Once the guile prompt appears, the LilyPond active module must be set as the current guile module:

```
guile> (set-current-module lilypond-module)
```

You can demonstrate these commands are operating properly by typing the name of a LilyPond public scheme function to check it has been defined:

```
guile> fret-diagram-verbose-markup
#<procedure fret-diagram-verbose-markup (layout props marking-list)>
```

If the LilyPond module has not been correctly loaded, an error message will be generated:

```
guile> fret-diagram-verbose-markup
ERROR: Unbound variable: fret-diagram-verbose-markup
ABORT: (unbound-variable)
```

Once the module is properly loaded, any valid LilyPond Scheme expression can be entered at the interactive prompt.

After the investigation is complete, the interactive guile interpreter can be exited:

```
guile> (quit)
```

The compilation of the '`.ly`' file will then continue.

Using the Guile debugger

To set breakpoints and/or enable tracing in Scheme functions, put

```
\include "guile-debugger.ly"
```

in your input file after any scheme procedures you have defined in that file. This will invoke the Guile command-line after having set up the environment for the debug command-line. When your input file is processed, a guile prompt will be displayed. You may now enter commands to set up breakpoints and enable tracing by the Guile debugger.

Using breakpoints

At the guile prompt, you can set breakpoints with the `set-break!` procedure:

```
guile> (set-break! my-scheme-procedure)
```

Once you have set the desired breakpoints, you exit the guile repl frame by typing:

```
guile> (quit)
```

Then, when one of the scheme routines for which you have set breakpoints is entered, guile will interrupt execution in a debug frame. At this point you will have access to Guile debugging commands. For a listing of these commands, type:

```
debug> help
```

Alternatively you may code the breakpoints in your Lilypond source file using a command such as:

```
#(set-break! my-scheme-procedure)
```

immediately after the `\include` statement. In this case the breakpoint will be set straight after you enter the `(quit)` command at the guile prompt.

Embedding breakpoint commands like this is particularly useful if you want to look at how the Scheme procedures in the '`.scm`' files supplied with LilyPond work. To do this, edit the file in the relevant directory to add this line near the top:

```
(use-modules (scm guile-debugger))
```

Now you can set a breakpoint after the procedure you are interested in has been declared. For example, if you are working on routines called by *print-book-with* in '`lily-library.scm`':

```
(define (print-book-with parser book process-procedure)
  (let* ((paper (ly:parser-lookup parser '$defaultpaper))
 (layout (ly:parser-lookup parser '$defaultlayout))
 (outfile-name (get-outfile-name parser)))
    (process-procedure book paper layout outfile-name)))

(define-public (print-book-with-defaults parser book)
  (print-book-with parser book ly:book-process))

(define-public (print-book-with-defaults-as-systems parser book)
  (print-book-with parser book ly:book-process-to-systems))
```

At this point in the code you could add this to set a breakpoint at print-book-with:

```
(set-break! print-book-with)
```

Tracing procedure calls and evaluator steps

Two forms of trace are available:

```
(set-trace-call! my-scheme-procedure)
```

and

```
(set-trace-subtree! my-scheme-procedure)
```

`set-trace-call!` causes Scheme to log a line to the standard output to show when the procedure is called and when it exits.

`set-trace-subtree!` traces every step the Scheme evaluator performs in evaluating the procedure.

10.7 Tracing object relationships

Understanding the LilyPond source often boils down to figuring out what is happening to the Grobs. Where (and why) are they being created, modified and destroyed? Tracing Lily through a debugger in order to identify these relationships can be time-consuming and tedious.

In order to simplify this process, a facility has been added to display the grobs that are created and the properties that are set and modified. Although it can be complex to get set up, once set up it easily provides detailed information about the life of grobs in the form of a network graph.

Each of the steps necessary to use the graphviz utility is described below.

1. Installing graphviz

   In order to create the graph of the object relationships, it is first necessary to install Graphviz. graphviz is available for a number of different platforms:

   http://www.graphviz.org/Download..php

2. Modifying config.make

   In order for the Graphviz tool to work, config.make must be modified. It is probably a good idea to first save a copy of config.make under a different name. Then, edit config.make by removing every occurrence of `-DNDEBUG`.

3. Rebuilding LilyPond

   The executable code of LilyPond must be rebuilt from scratch:

   ```
   make -C lily clean && make -C lily
   ```

4. Create a graphviz-compatible '`.ly`' file

   In order to use the graphviz utility, the '`.ly`' file must include '`ly/graphviz-init.ly`', and should then specify the grobs and symbols that should be tracked. An example of this is found in '`input/regression/graphviz.ly`'.

5. Run lilypond with output sent to a log file

   The Graphviz data is sent to stderr by lilypond, so it is necessary to redirect stderr to a logfile:

   ```
   lilypond graphviz.ly 2> graphviz.log
   ```

6. Edit the logfile

   The logfile has standard lilypond output, as well as the Graphviz output data. Delete everything from the beginning of the file up to but not including the first occurrence of `digraph`.

   Also, delete the final liypond message about successs from the end of the file.

7. Process the logfile with `dot`

   The directed graph is created from the log file with the program `dot`:

   ```
   dot -Tpdf graphviz.log > graphviz.pdf
   ```

The pdf file can then be viewed with any pdf viewer.

When compiled without `-DNDEBUG`, lilypond may run slower than normal. The original configuration can be restored by either renaming the saved copy of `config.make` or rerunning `configure`. Then rebuild lilypond with

```
make -C lily clean && make -C lily
```

10.8 Adding or modifying features

When a new feature is to be added to LilyPond, it is necessary to ensure that the feature is properly integrated to maintain its long-term support. This section describes the steps necessary for feature addition and modification.

10.8.1 Write the code

You should probably create a new git branch for writing the code, as that will separate it from the master branch and allow you to continue to work on small projects related to master.

Please be sure to follow the rules for programming style discussed earlier in this chapter.

10.8.2 Write regression tests

In order to demonstrate that the code works properly, you will need to write one or more regression tests. These tests are typically '`.ly`' files that are found in '`input/regression`'.

Regression tests should be as brief as possible to demonstrate the functionality of the code.

Regression tests should generally cover one issue per test. Several short, single-issue regression tests are preferred to a single, long, multiple-issue regression test.

Use existing regression tests as templates to demonstrate the type of header information that should be included in a regression test.

10.8.3 Write convert-ly rule

If the modification changes the input syntax, a convert-ly rule should be written to automatically update input files from older versions.

convert-ly rules are found in python/convertrules.py

If possible, the convert-ly rule should allow automatic updating of the file. In some cases, this will not be possible, so the rule will simply point out to the user that the feature needs manual correction.

Updating version numbers

If a development release occurs between you writing your patch and having it approved+pushed, you will need to update the version numbers in your tree. This can be done with:

```
scripts/auxiliar/update-patch-version old.version.number new.version.number
```

It will change all files in git, so use with caution and examine the resulting diff.

10.8.4 Automatically update documentation

`convert-ly` should be used to update the documentation, the snippets, and the regression tests. This not only makes the necessary syntax changes, it also tests the `convert-ly` rules.

The automatic updating is performed by moving to the top-level source directory, then running:

```
scripts/auxiliar/update-with-convert-ly.sh
```

If you did an out-of-tree build, pass in the relative path:

```
BUILD_DIR=../build-lilypond/ scripts/auxiliar/update-with-convert-ly.sh
```

10.8.5 Manually update documentation

Where the convert-ly rule is not able to automatically update the inline lilypond code in the documentation (i.e. if a NOT_SMART rule is used), the documentation must be manually updated. The inline snippets that require changing must be changed in the English version of the docs and all translated versions. If the inline code is not changed in the translated documentation, the old snippets will show up in the English version of the documentation.

Where the convert-ly rule is not able to automatically update snippets in Documentation/snippets/, those snippets must be manually updated. Those snippets should be copied to Documentation/snippets/new. The comments at the top of the snippet describing its automatic generation should be removed. All translated texidoc strings should be removed. The comment "% begin verbatim" should be removed. The syntax of the snippet should then be manually edited.

Where snippets in Documentation/snippets are made obsolete, the snippet should be copied to Documentation/snippets/new. The comments and texidoc strings should be removed as described above. Then the body of the snippet should be changed to:

```
\markup {
  This snippet is deprecated as of version X.Y.Z and
  will be removed from the documentation.
}
```

where X.Y.Z is the version number for which the convert-ly rule was written.

Update the snippet files by running:

```
scripts/auxiliar/makelsr.py
```

Where the convert-ly rule is not able to automatically update regression tests, the regression tests in input/regression should be manually edited.

Although it is not required, it is helpful if the developer can write relevant material for inclusion in the Notation Reference. If the developer does not feel qualified to write the documentation, a documentation editor will be able to write it from the regression tests. The text that is added to or removed from the documentation should be changed only in the English version.

10.8.6 Edit changes.tely

An entry should be added to Documentation/changes.tely to describe the feature changes to be implemented. This is especially important for changes that change input file syntax.

Hints for changes.tely entries are given at the top of the file.

New entries in changes.tely go at the top of the file.

The changes.tely entry should be written to show how the new change improves LilyPond, if possible.

10.8.7 Verify successful build

When the changes have been made, successful completion must be verified by doing

```
make all
make doc
```

When these commands complete without error, the patch is considered to function successfully.

Developers on Windows who are unable to build LilyPond should get help from a Linux or OSX developer to do the make tests.

10.8.8 Verify regression tests

In order to avoid breaking LilyPond, it is important to verify that the regression tests succeed, and that no unwanted changes are introduced into the output. This process is described in Section 9.4 [Regtest comparison], page 87.

Typical developer's edit/compile/test cycle

TODO: is `[-jX CPU_COUNT=X]` useful for `test-baseline`, `check`, `clean`, `test-redo`? Neil Puttock says it is useful for everything but `clean`, which is disk-limited. Need to check formally.

- Initial test:

```
make [-jX]
make test-baseline
make [-jX CPU_COUNT=X] check
```

- Edit/compile/test cycle:

```
## edit source files, then...

make clean                      ## only if needed (see below)
make [-jX]                      ## only if needed (see below)
make test-redo                  ## redo files differing from baseline
make [-jX CPU_COUNT=X] check    ## CPU_COUNT here?
```

- Reset:

```
make test-clean
```

If you modify any source files that have to be compiled (such as '`.cc`' or '`.hh`' files in '`flower/`' or '`lily/`'), then you must run `make` before `make test-redo`, so `make` can compile the modified files and relink all the object files. If you only modify files which are interpreted, like those in the '`scm/`' and '`ly/`' directories, then `make` is not needed before `make test-redo`.

TODO: Fix the following paragraph. You can do `rm mf/out/*` instead of make clean, and you can probably do `make -C mf/ clean` as well, but I haven't checked it – cds

Also, if you modify any font definitions in the '`mf/`' directory then you must run `make clean` and `make` before running `make test-redo`. This will recompile everything, whether modified or not, and takes a lot longer.

Running `make check` will leave an HTML page '`out/test-results/index.html`'. This page shows all the important differences that your change introduced, whether in the layout, MIDI, performance or error reporting.

10.8.9 Post patch for comments

See [Uploading a patch for review], page 19.

10.8.10 Push patch

Once all the comments have been addressed, the patch can be pushed.

If the author has push privileges, the author will push the patch. Otherwise, a developer with push privileges will push the patch.

10.8.11 Closing the issues

Once the patch has been pushed, all the relevant issues should be closed.

On Rietveld, the author should log in an close the issue either by using the 'Edit Issue' link, or by clicking the circled x icon to the left of the issue name.

If the changes were in response to a feature request on the Google issue tracker for LilyPond, the author should change the status to Fixed and a tag 'fixed_x_y_z' should be added, where the patch was fixed in version x.y.z. If the author does not have privileges to change the status, an email should be sent to bug-lilypond requesting the BugMeister to change the status.

10.9 Iterator tutorial

TODO – this is a placeholder for a tutorial on iterators

Iterators are routines written in C++ that process music expressions and sent the music events to the appropriate engravers and/or performers.

10.10 Engraver tutorial

Engravers are C++ classes that catch music events and create the appropriate grobs for display on the page. Though the majority of engravers are responsible for the creation of a single grob, in some cases (e.g. `New_fingering_engraver`), several different grobs may be created.

Engravers listen for events and acknowledge grobs. Events are passed to the engraver in time-step order during the iteration phase. Grobs are made available to the engraver when they are created by other engravers during the iteration phase.

10.10.1 Useful methods for information processing

An engraver inherits the following public methods from the Translator base class, which can be used to process listened events and acknowledged grobs:

- `virtual void initialize ()`
- `void start_translation_timestep ()`
- `void process_music ()`
- `void process_acknowledged ()`
- `void stop_translation_timestep ()`

- `virtual void finalize ()`

These methods are listed in order of translation time, with `initialize ()` and `finalize ()` bookending the whole process. `initialize ()` can be used for one-time initialization of context properties before translation starts, whereas `finalize ()` is often used to tie up loose ends at the end of translation: for example, an unterminated spanner might be completed automatically or reported with a warning message.

10.10.2 Translation process

At each timestep in the music, translation proceeds by calling the following methods in turn:

`start_translation_timestep ()` is called before any user information enters the translators, i.e., no property operations (\set, \override, etc.) or events have been processed yet.

`process_music ()` and `process_acknowledged ()` are called after all events in the current time step have been heard, or all grobs in the current time step have been acknowledged. The latter tends to be used exclusively with engravers which only acknowledge grobs, whereas the former is the default method for main processing within engravers.

`stop_translation_timestep ()` is called after all user information has been processed prior to beginning the translation for the next timestep.

10.10.3 Preventing garbage collection for SCM member variables

In certain cases, an engraver might need to ensure private Scheme variables (with type SCM) do not get swept away by Guile's garbage collector: for example, a cache of the previous key signature which must persist between timesteps. The method `virtual derived_mark () const` can be used in such cases:

```
Engraver_name::derived_mark ()
{
  scm_gc_mark (private_scm_member_)
}
```

10.10.4 Listening to music events

External interfaces to the engraver are implemented by protected macros including one or more of the following:

- `DECLARE_TRANSLATOR_LISTENER (event_name)`
- `IMPLEMENT_TRANSLATOR_LISTENER (Engraver_name, event_name)`

where *event_name* is the type of event required to provide the input the engraver needs and *Engraver_name* is the name of the engraver.

Following declaration of a listener, the method is implemented as follows:

```
IMPLEMENT_TRANSLATOR_LISTENER (Engraver_name, event_name)
void
Engraver_name::listen_event_name (Stream event *event)
{
  ...body of listener method...
}
```

10.10.5 Acknowledging grobs

Some engravers also need information from grobs as they are created and as they terminate. The mechanism and methods to obtain this information are set up by the macros:

- `DECLARE_ACKNOWLEDGER (grob_interface)`
- `DECLARE_END_ACKNOWLEDGER (grob_interface)`

where *grob_interface* is an interface supported by the grob(s) which should be acknowledged. For example, the following code would declare acknowledgers for a `NoteHead` grob (via the `note-head-interface`) and any grobs which support the `side-position-interface`:

```
DECLARE_ACKNOWLEDGER (note_head)
DECLARE_ACKNOWLEDGER (side_position)
```

The `DECLARE_END_ACKNOWLEDGER ()` macro sets up a spanner-specific acknowledger which will be called whenever a spanner ends.

Following declaration of an acknowledger, the method is coded as follows:

```
void
Engraver_name::acknowledge_interface_name (Grob_info info)
{
  ...body of acknowledger method...
}
```

10.10.6 Engraver declaration/documentation

An engraver must have a public macro

- `TRANSLATOR_DECLARATIONS (Engraver_name)`

where `Engraver_name` is the name of the engraver. This defines the common variables and methods used by every engraver.

At the end of the engraver file, one or both of the following macros are generally called to document the engraver in the Internals Reference:

- `ADD_ACKNOWLEDGER (Engraver_name, grob_interface)`
- `ADD_TRANSLATOR (Engraver_name, Engraver_doc, Engraver_creates, Engraver_reads, Engraver_writes)`

where `Engraver_name` is the name of the engraver, `grob_interface` is the name of the interface that will be acknowledged, `Engraver_doc` is a docstring for the engraver, `Engraver_creates` is the set of grobs created by the engraver, `Engraver_reads` is the set of properties read by the engraver, and `Engraver_writes` is the set of properties written by the engraver.

The `ADD_ACKNOWLEDGER` and `ADD_TRANSLATOR` macros use a non-standard indentation system. Each interface, grob, read property, and write property is on its own line, and the closing parenthesis and semicolon for the macro all occupy a separate line beneath the final interface or write property. See existing engraver files for more information.

10.11 Callback tutorial

TODO – This is a placeholder for a tutorial on callback functions.

10.12 LilyPond scoping

The Lilypond language has a concept of scoping, i.e. you can do

```
foo = 1

#(begin
   (display (+ foo 2)))
```

with `\paper`, `\midi` and `\header` being nested scope inside the '`.ly`' file-level scope. `foo = 1` is translated in to a scheme variable definition.

This implemented using modules, with each scope being an anonymous module that imports its enclosing scope's module.

Lilypond's core, loaded from '`.scm`' files, is usually placed in the `lily` module, outside the '`.ly`' level. In the case of

```
lilypond a.ly b.ly
```

we want to reuse the built-in definitions, without changes effected in user-level '`a.ly`' leaking into the processing of '`b.ly`'.

The user-accessible definition commands have to take care to avoid memory leaks that could occur when running multiple files. All information belonging to user-defined commands and markups is stored in a manner that allows it to be garbage-collected when the module is dispersed, either by being stored module-locally, or in weak hash tables.

10.13 LilyPond miscellany

This is a place to dump information that may be of use to developers but doesn't yet have a proper home. Ideally, the length of this section would become zero as items are moved to other homes.

10.13.1 Spacing algorithms

Here is information from an email exchange about spacing algorithms.

On Thu, 2010-02-04 at 15:33 -0500, Boris Shingarov wrote: I am experimenting with some modifications to the line breaking code, and I am stuck trying to understand how some of it works. So far my understanding is that Simple_spacer operates on a vector of Grobs, and it is a well-known Constrained-QP problem (rods = constraints, springs = quadratic function to minimize). What I don't understand is, if the spacer operates at the level of Grobs, which are built at an earlier stage in the pipeline, how are the changes necessitated by differences in line breaking, taken into account? in other words, if I take the last measure of a line and place it on the next line, it is not just a matter of literally moving that graphic to where the start of the next line is, but I also need to draw a clef, key signature, and possibly other fundamental things – but at that stage in the rendering pipeline, is it not too late??

Joe Neeman answered:

We create lots of extra grobs (eg. a BarNumber at every bar line) but most of them are not drawn. See the break-visibility property in item-interface.

10.13.2 Info from Han-Wen email

In 2004, Douglas Linhardt decided to try starting a document that would explain LilyPond architecture and design principles. The material below is extracted from that email, which can be found at http://thread.gmane.org/gmane.comp.gnu.lilypond.devel/2992. The headings reflect questions from Doug or comments from Han-Wen; the body text are Han-Wen's answers.

Figuring out how things work.

I must admit that when I want to know how a program works, I use grep and emacs and dive into the source code. The comments and the code itself are usually more revealing than technical documents.

What's a grob, and how is one used?

Graphical object - they are created from within engravers, either as Spanners (derived class) -slurs, beams- or Items (also a derived class) -notes, clefs, etc.

There are two other derived classes System (derived from Spanner, containing a "line of music") and Paper_column (derived from Item, it contains all items that happen at the same moment). They are separate classes because they play a special role in the linebreaking process.

What's a smob, and how is one used?

A C(++) object that is encapsulated so it can be used as a Scheme object. See GUILE info, "19.3 Defining New Types (Smobs)"

@subheading When is each C++ class constructed and used

- Music classes

  In the parser.yy see the macro calls MAKE_MUSIC_BY_NAME().

- Contexts

  Constructed during "interpreting" phase.

- Engravers

  Executive branch of Contexts, plugins that create grobs, usually one engraver per grob type. Created together with context.

- Layout Objects

  = grobs

- Grob Interfaces

  These are not C++ classes per se. The idea of a Grob interface hasn't crystallized well. ATM, an interface is a symbol, with a bunch of grob properties. They are not objects that are created or destroyed.

- Iterators

  Objects that walk through different music classes, and deliver events in a synchronized way, so that notes that play together are processed at the same moment and (as a result) end up on the same horizontal position.

  Created during interpreting phase.

  BTW, the entry point for interpreting is ly:run-translator (ly_run_translator on the C++ side)

Can you get to Context properties from a Music object?

You can create music object with a Scheme function that reads context properties (the \apply-context syntax). However, that function is executed during Interpreting, so you can not really get Context properties from Music objects, since music objects are not directly connected to Contexts. That connection is made by the Music_iterators

Can you get to Music properties from a Context object?

Yes, if you are given the music object within a Context object. Normally, the music objects enter Contexts in synchronized fashion, and the synchronization is done by Music_iterators.

What is the relationship between C++ classes and Scheme objects?

Smobs are C++ objects in Scheme. Scheme objects (lists, functions) are manipulated from C++ as well using the GUILE C function interface (prefix: scm_)

How do Scheme procedures get called from C++ functions?

scm_call_*, where * is an integer from 0 to 4. Also scm_c_eval_string (), scm_eval ()

How do C++ functions get called from Scheme procedures?

Export a C++ function to Scheme with LY_DEFINE.

What is the flow of control in the program?

Good question. Things used to be clear-cut, but we have Scheme and SMOBs now, which means that interactions do not follow a very rigid format anymore. See below for an overview, though.

Does the parser make Scheme procedure calls or C++ function calls?

Both. And the Scheme calls can call C++ and vice versa. It's nested, with the SCM datatype as lubrication between the interactions

(I think the word "lubrication" describes the process better than the traditional word "glue")

How do the front-end and back-end get started?

Front-end: a file is parsed, the rest follows from that. Specifically,

Parsing leads to a Music + Music_output_def object (see parser.yy, definition of toplevel_expression)

A Music + Music_output_def object leads to a Global_context object (see ly_run_translator ())

During interpreting, Global_context + Music leads to a bunch of Contexts (see Global_translator::run_iterator_on_me ()).

After interpreting, Global_context contains a Score_context (which contains staves, lyrics etc.) as a child. Score_context::get_output () spews a Music_output object (either a Paper_score object for notation or Performance object for MIDI).

The Music_output object is the entry point for the backend (see ly_render_output ()).

The main steps of the backend itself are in

- '`paper-score.cc`' , Paper_score::process_
- '`system.cc`' , System::get_lines()
- The step, where things go from grobs to output, is in System::get_line(): each grob delivers a Stencil (a Device independent output description), which is interpreted by our outputting backends ('`scm/output-tex.scm`' and '`scm/output-ps.scm`') to produce TeX and PS.

Interactions between grobs and putting things into .tex and .ps files have gotten a little more complex lately. Jan has implemented page-breaking, so now the backend also involves Paper_book, Paper_lines and other things. This area is still heavily in flux, and perhaps not something you should want to look at.

How do the front-end and back-end communicate?

There is no communication from backend to front-end. From front-end to backend is simply the program flow: music + definitions gives contexts, contexts yield output, after processing, output is written to disk.

Where is the functionality associated with KEYWORDs?

See '`my-lily-lexer.cc`' (keywords, there aren't that many) and '`ly/*.ly`' (most of the other backslashed `/\words` are identifiers)

What Contexts/Properties/Music/etc. are available when they are processed?

What do you mean exactly with this question?

See '`ly/engraver-init.ly`' for contexts, see '`scm/define-*.scm`' for other objects.

How do you decide if something is a Music, Context, or Grob property?

Why is part-combine-status a Music property when it seems (IMO) to be related to the Staff context?

The Music_iterators and Context communicate through two channels

Music_iterators can set and read context properties, idem for Engravers and Contexts

Music_iterators can send "synthetic" music events (which aren't in the input) to a context. These are caught by Engravers. This is mostly a one way communication channel.

part-combine-status is part of such a synthetic event, used by Part_combine_iterator to communicate with Part_combine_engraver.

Deciding between context and music properties

I'm adding a property to affect how \autochange works. It seems to me that it should be a context property, but the Scheme autochange procedure has a Music argument. Does this mean I should use a Music property?

\autochange is one of these extra strange beasts: it requires look-ahead to decide when to change staves. This is achieved by running the interpreting step twice (see '`scm/part-combiner.scm`' , at the bottom), and storing the result of the first step (where to switch staves) in a Music property. Since you want to influence that where-to-switch list, your must affect the code in make-autochange-music ('`scm/part-combiner.scm`'). That code is called directly from the parser and there are no official "parsing properties" yet, so there is no generic way to tune \autochange. We would have to invent something new for this, or add a separate argument,

```
\autochange #around-central-C ..music..
```

where around-central-C is some function that is called from make-autochange-music.

More on context and music properties

From Neil Puttock, in response to a question about transposition:

Context properties (using \set & \unset) are tied to engravers: they provide information relevant to the generation of graphical objects.

Since transposition occurs at the music interpretation stage, it has no direct connection with engravers: the pitch of a note is fixed before a notehead is created. Consider the following minimal snippet:

```
{ c' }
```

This generates (simplified) a NoteEvent, with its pitch and duration as event properties,

```
(make-music
  'NoteEvent
  'duration
  (ly:make-duration 2 0 1 1)
  'pitch
  (ly:make-pitch 0 0 0)
```

which the Note_heads_engraver hears. It passes this information on to the NoteHead grob it creates from the event, so the head's correct position and duration-log can be determined once it's ready for printing.

If we transpose the snippet,

```
\transpose c d { c' }
```

the pitch is changed before it reaches the engraver (in fact, it happens just after the parsing stage with the creation of a TransposedMusic music object):

```
(make-music
 'NoteEvent
 'duration
 (ly:make-duration 2 0 1 1)
 'pitch
```

```
(ly:make-pitch 0 1 0)
```

You can see an example of a music property relevant to transposition: untransposable.

```
\transpose c d { c'2 \withMusicProperty #'untransposable ##t c' }
```

-> the second c' remains untransposed.

Take a look at '`lily/music.cc`' to see where the transposition takes place.

How do I tell about the execution environment?

I get lost figuring out what environment the code I'm looking at is in when it executes. I found both the C++ and Scheme autochange code. Then I was trying to figure out where the code got called from. I finally figured out that the Scheme procedure was called before the C++ iterator code, but it took me a while to figure that out, and I still didn't know who did the calling in the first place. I only know a little bit about Flex and Bison, so reading those files helped only a little bit.

Han-Wen: GDB can be of help here. Set a breakpoint in C++, and run. When you hit the breakpoint, do a backtrace. You can inspect Scheme objects along the way by doing

```
p ly_display_scm(obj)
```

this will display OBJ through GUILE.

10.13.3 Music functions and GUILE debugging

Ian Hulin was trying to do some debugging in music functions, and came up with the following question

HI all, I'm working on the Guile Debugger Stuff, and would like to try debugging a music function definition such as:

```
conditionalMark = #(define-music-function (parser location) ()
    #{ \tag #'instrumental-part {\mark \default}  #} )
```

It appears conditionalMark does not get set up as an equivalent of a Scheme

```
(define conditionalMark = define-music-function(parser location () ...
```

although something gets defined because Scheme apparently recognizes

```
#(set-break! conditionalMark)
```

later on in the file without signalling any Guile errors.

However the breakpoint trap is never encountered as define-music-function passed things on to ly:make-music-function, which is really C++ code ly_make_music_function, so Guile never finds out about the breakpoint.

Han-Wen answered as follows:

You can see the definition by doing

```
#(display conditionalMark)
```

noindent inside the '`.ly`' file.

The breakpoint failing may have to do with the call sequence. See '`parser.yy`', run_music_function(). The function is called directly from C++, without going through the GUILE evaluator, so I think that is why there is no debugger trap.

11 Release work

11.1 Development phases

There are 2 states of development on `master`:

1. **Normal development**: Any commits are fine.
2. **Build-frozen**: Do not require any additional or updated libraries or make non-trivial changes to the build process. Any such patch (or branch) may not be merged with master during this period.

   This should occur approximately 1 month before any alpha version of the next stable release, and ends when the next unstable branch begins.

After announcing a beta release, branch `stable/2.x`. There are 2 states of development for this branch:

1. **Normal maintenance**: The following patches **MAY NOT** be merged with this branch:
   - Any change to the input syntax. If a file compiled with a previous `2.x` (beta) version, then it must compile in the new version.

     Exception: any bugfix to a Critical issue.
   - New features with new syntax *may be committed*, although once committed that syntax cannot change during the remainder of the stable phase.
   - Any change to the build dependencies (including programming libraries, documentation process programs, or python modules used in the buildscripts). If a contributor could compile a previous lilypond `2.x`, then he must be able to compile the new version.
2. **Release prep**: Only translation updates and important bugfixes are allowed.

11.2 Minor release checklist

A "minor release" means an update of `y` in `2.x.y`.

Pre-release

1. Switch to the release branch, get changes, prep release announcement:

   ```
   git checkout release/unstable
   git merge origin
   vi Documentation/web/news-front.itexi Documentation/web/news.itexi
   ```
2. Commit, push, switch back to master:

   ```
   git commit -m "Release: update news." Documentation/web/
   git push origin
   ```
3. If you do not have the previous release test-output tarball, download it and put it in `regtests/`
4. Build release on GUB by running:

   ```
   make LILYPOND_BRANCH=release/unstable lilypond
   ```

   or something like:

   ```
   make LILYPOND_BRANCH=stable/2.12 lilypond
   ```
5. Check the regtest comparison in '`uploads/webtest/`' for any unintentional breakage. More info in Section 9.2 [Precompiled regression tests], page 86.
6. If any work was done on GUB since the last release, upload binaries to a temporary location, ask for feedback, and wait a day or two in case there's any major problems.

> **Note:** Always do this for a stable release.

Actual release

1. If you're not the right user on the webserver, remove the `t` from the rsync command in:

   ```
   test-lily/rsync-lily-doc.py
   test-lily/rsync-test.py
   ```

   `graham` owns v2.13; `han-wen` owns v2.12.

2. Upload GUB by running:

   ```
   make lilypond-upload \
     LILYPOND_REPO_URL=git://git.sv.gnu.org/lilypond.git \
     LILYPOND_BRANCH=release/unstable
   ```

   or something like:

   ```
   make lilypond-upload \
     LILYPOND_REPO_URL=git://git.sv.gnu.org/lilypond.git \
     LILYPOND_BRANCH=stable/2.12
   ```

Post release

1. Switch back to master and get the updated news:

   ```
   git checkout master
   git merge release/unstable
   ```

2. Update '`VERSION`' in lilypond git and upload changes:

   ```
   vi VERSION
   ```

   - VERSION = what you just did +0.0.1
   - DEVEL_VERSION = what you just did (i.e. is now online)
   - STABLE_VERSION = what's online (probably no change here)

   ```
   git commit -m "Release: bump version." VERSION
   git push origin
   ```

3. (for now) do a `make doc` and manually upload:

   ```
   ### upload-lily-web-media.sh
   #!/bin/sh
   BUILD_DIR=$HOME/src/build-lilypond

   PICS=$BUILD_DIR/Documentation/pictures/out-www/
   EXAMPLES=$BUILD_DIR/Documentation/web/ly-examples/out-www/

   cd $BUILD_DIR
   rsync -a $PICS graham@lilypond.org:media/pictures
   rsync -a $EXAMPLES graham@lilypond.org:media/ly-examples
   ```

4. Wait a few hours for the website to update.
5. Email release notice to `info-lilypond`

11.3 Major release checklist

A "major release" means an update of `x` in `2.x.0`.

Main requirements

This is the current official guidelines.

- 0 Critical issues for two weeks (14 days) after the latest release candidate.

Potential requirements

These might become official guidelines in the future.

- Check reg test
- Check all 2ly scripts
- Check for emergencies the docs:

```
grep FIXME --exclude "misc/*" --exclude "*GNUmakefile" \
  --exclude "snippets/*" ????*/*
```

- Check for altered regtests, and document as necessary. (update numbers in the following command as appropriate)

```
git diff -u -r release/2.12.0-1 -r release/2.13.13-1 input/regression/
```

Housekeeping requirements

Before the release:

- write release notes. note: stringent size requirements for various websites, so be brief.
- Run convert-ly on all files, bump parser minimum version.
- Make FTP directories on lilypond.org
- Website:
  - make a link from the old unstable to the next stable in lilypond.org's /doc/ dir. Keep all previous unstable->stable doc symlinks.

    Also, make the old docs self-contained – if there's a redirect in /doc/v2.12/Documentation/index.html , replace it with the index.html.old-2.12 files.

    The post-2.13 docs will need another way of handling the self-containment. It won't be hard to whip up a python script that changes the link to ../../../../manuals.html to ../website/manuals.html , but it's still a 30-minute task that needs to be done before 2.16.
  - doc auto redirects to v2.LATEST-STABLE
  - add these two lines to http://www.lilypond.org/robots.txt:

```
Disallow: /doc/v2.PREVIOUS-STABLE/
Disallow: /doc/v2.CURRENT-DEVELOPMENT/
```

Unsorted

- submit pots for translation: send url of tarball to translation@iro.umontreal.ca, mentioning lilypond-VERSION.pot
- update links to distros providing lilypond packages? link in: '`Documentation/web/download.itexi`'

  This has nothing to do with the release, but it's a "periodic maintenance" task that might make sense to include with releases.
- Send announcements to...

  News:

```
comp.music.research
comp.os.linux.announce

comp.text.tex
rec.music.compose
```

  Mail:

```
info-lilypond@gnu.org

linux-audio-announce@lists.linuxaudio.org
linux-audio-user@lists.linuxaudio.org
linux-audio-dev@lists.linuxaudio.org

tex-music@icking-music-archive.org

--- non-existant?
abcusers@blackmill.net

rosegarden-user@lists.sourceforge.net
info-gnu@gnu.org
noteedit-user@berlios.de

gmane.comp.audio.fomus.devel
gmane.linux.audio.users
gmane.linux.audio.announce
gmane.comp.audio.rosegarden.devel
```

Web:

```
lilypond.org
freshmeat.net
linuxfr.com
http://www.apple.com/downloads
harmony-central.com (news@harmony-central.com)
versiontracker.com [auto]
hitsquad.com [auto]
http://www.svgx.org
https://savannah.gnu.org/news/submit.php?group_id=1673
```

11.4 Release extra notes

Regenerating regression tests

Regenerating regtests (if the lilypond-book naming has changed):

- git checkout release/lilypond-X.Y.Z-A
- take lilypond-book and any related makefile updates from the latest git.
- configure; make; make test
- tar -cjf lilypond-X.Y.Z-A.test-output.tar.bz2 input/regression/out-test/
- mv lilypond-X.Y.Z-A.test-output.tar.bz2 ../gub/regtests/
- cd ../gub/regtests/
- make lilypond

stable/2.12

If releasing stable/2.12, then:

- apply doc patch: patches/rsync-lily.patch (or something like that)
- change infodir in gub/specs/lilypond-doc.py from "lilypond.info" to "lilypond-web.info"

Updating a release (changing a in x.y.z-a)

Really tentative instructions, almost certainly can be done better.

1. change the VERSION back to release you want. push change. (hopefully you'll have forgotten to update it when you made your last release)
2. make sure that there aren't any lilypond files floating around in target/ (like usr/bin/lilypond).
3. build the specific package(s) you want, i.e.

   ```
   bin/gub mingw::lilypond-installer
   make LILYPOND_BRANCH=stable/2.12 -f lilypond.make doc
   bin/gub --platform=darwin-x86 'git://git.sv.gnu.org/lilypond-doc.git?branch=stable/
   ```

   or

   build everything with the normal "make lilypond", then (maybe) manually delete stuff you don't want to upload.
4. manually upload them. good luck figuring out the rsync command(s). Hints are in test-lily/

   or

   run the normal lilypond-upload command, and (maybe) manually delete stuff you didn't want to upload from the server.

12 Build system notes

Note: This chapter is in high flux, and is being run in a "wiki-like" fashion. Do not trust anything you read in this chapter.

12.1 Build system overview

Build system is currently GNU make, with an extra "stepmake" layer on top. Look at files in '`make/`' and '`stepmake/`' and all '`GNUmakefile`'s.

There is wide-spread dissatisfaction with this system, and we are considering changing. This would be a huge undertaking (estimated 200+ hours). This change will probably involve not using GNU make any more – but a discussion about the precise build system will have to wait. Before we reach that point, we need to figure out (at least approximately) what the current build system does.

Fundamentally, a build system does two things:

1. Constructs command-line commands, for example:

```
lilypond-book \
  --tons --of --options \
  pitches.itely
texi2pdf \
  --more --imperial --and --metric --tons --of --options \
  pitches.texi
```

2. If there was a previous build, it decides which parts of the system need to be rebuilt.

When I try to do anything in the build system, it helps to remind myself of this. The "end result" is just a series of command-line commands. All the black magick is just an attempt to construct those commands.

12.2 Tips for working on the build system

- Add:

```
echo "aaa"


echo "bbb"
```

  to the build system files in various places. This will let you track where the program is, in various points of the build.

  PH note. There are lots of places where Make doesn't let you put echo commands. My top tip for tracing how make runs is to put

```
$(error Some Text to display)
```

  This will stop make running and print the text `Some Text to display`.

  End PH note.

- First task: understand how `make website` works, *without* the translations. Looking at the english-only website is the best introduction to the build system... it only covers about 5% of the whole thing, but even that will likely take 10 hours or more.

12.3 General build system notes

12.3.1 How stepmake works

Typing make website runs the file '`GNUmakefile`' from the build directory. This only contains 3 lines:

```
depth = .
include config$(if $(conf),-$(conf),).make
include $(configure-srcdir)/GNUmakefile.in
```

The variable `depth` is used throughout the make system to track how far down the directory structure the make is. The first include sets lots of variables but doesn't "do" anything. The second runs the file '`GNUmakefile.in`' from the top level source directory.

This sets another load of variables, and then includes (i.e. immediately runs) '`stepmake.make`' from the '`make`' subdirectory. This sets a load of other variables, does some testing to see if SCONS (another build tool?) is being used, and then runs '`make/config.make`' - which doesn't seem to exist...

GP: scons is indeed a different build tool; I think that Jan experimented with it 5 years ago or something. It seems like we still have bits and pieces of it floating around.

Next, it runs '`make/toplevel-version.make`', which sets the version variables for major, minor, patch, stable, development and mypatchlevel (which seems to be used for patch numbers for non-stable versions only?).

Next - '`make/local.make`', which doesn't exist.

Then a few more variable and the interesting comment:

```
# Don't try to outsmart us, you puny computer!
# Well, UGH.  This only removes builtin rules from
```

and then tests to see whether BUILTINS_REMOVED is defined. It appears to be when I run make, and so '`stepmake/stepmake/no-builtin-rules.make`' is run. The comment at the head of this file says:

```
# UGH.  GNU make comes with implicit rules.
# We don't want any of them, and can't force users to run
# --no-builtin-rules
```

I've not studied that file at length, but assume it removes all make's build-in rules (e.g. '`*.c`' files are run through the GNU C compiler) - there's a lot of them in here, and a lot of comments, and I'd guess most of it isn't needed.

We return to '`stepmake.make`', where we hit the make rule all: The first line of this is:

```
-include $(addprefix $(depth)/make/,$(addsuffix -inclusions.make, $(LOCALSTEPMAKE_TEMPL
```

which, when the variables are substituted, gives:

```
./make/generic-inclusions.make
./make/lilypond-inclusions.make.
```

(Note - according to the make documentation, -include is only different from include in that it doesn't produce any kind of error message when the included file doesn't exist).

And the first file doesn't exist. Nor the second. Next:

```
-include $(addprefix $(stepdir)/,$(addsuffix -inclusions.make, $(STEPMAKE_TEMPLATES)))
```

which expands to the following files:

```
/home/phil/lilypond-git/stepmake/stepmake/generic-inclusions.make
/home/phil/lilypond-git/stepmake/stepmake/toplevel-inclusions.make
/home/phil/lilypond-git/stepmake/stepmake/po-inclusions.make
/home/phil/lilypond-git/stepmake/stepmake/install-inclusions.make.
```

One little feature to notice here - these are all absolute file locations - the line prior to this used relative locations. And none of these files exist, either. (Further note - I'm assuming all these lines of make I'm following are autogenerated, but that'll be something else to discover.)

Next in ‘stepmake.make’:

```
include $(addprefix $(stepdir)/,$(addsuffix -vars.make, $(STEPMAKE_TEMPLATES)))
```

which expands to:

```
/home/phil/lilypond-git/stepmake/stepmake/generic-vars.make
/home/phil/lilypond-git/stepmake/stepmake/toplevel-vars.make
/home/phil/lilypond-git/stepmake/stepmake/po-vars.make
/home/phil/lilypond-git/stepmake/stepmake/install-vars.make.
```

Woo. They all exist (they should as there’s no - in front of the include). ‘generic-vars.make’ sets loads of variables (funnily enough). ‘toplevel-vars.make’ is very short - one line commented as `# override Generic_vars.make:` and 2 as follows:

```
# urg?
include $(stepdir)/documentation-vars.make
```

I assume the urg comment refers to the fact that this should really just create more variables, but it actually sends us off to ‘/home/phil/lilypond-git/stepmake/stepmake/documentation-vars.make’.

That file is a 3 line variable setting one.

‘po-vars.make’ has the one-line comment `# empty`, as does ‘install-vars.make’.

So now we’re back to ‘stepmake.make’.

The next lines are :

```
# ugh. need to do this because of PATH :=$(top-src-dir)/..:$(PATH)
include $(addprefix $(depth)/make/,$(addsuffix -vars.make, $(LOCALSTEPMAKE_TEMPLATES)))
```

and the include expands to:

```
include ./make/generic-vars.make ./make/lilypond-vars.make.
```

These again set variables, and in some cases export them to allow child `make` processes to use them.

The final 4 lines of ‘stepmake.make’ are:

```
include $(addprefix $(depth)/make/,$(addsuffix -rules.make, $(LOCALSTEPMAKE_TEMPLATES)))
include $(addprefix $(stepdir)/,$(addsuffix -rules.make, $(STEPMAKE_TEMPLATES)))
include $(addprefix $(depth)/make/,$(addsuffix -targets.make, $(LOCALSTEPMAKE_TEMPLATES
include $(addprefix $(stepdir)/,$(addsuffix -targets.make, $(STEPMAKE_TEMPLATES)))
```

which expand as follows:

```
include ./make/generic-rules.make ./make/lilypond-rules.make
include
  /home/phil/lilypond-git/stepmake/stepmake/generic-rules.make
  /home/phil/lilypond-git/stepmake/stepmake/toplevel-rules.make
  /home/phil/lilypond-git/stepmake/stepmake/po-rules.make
  /home/phil/lilypond-git/stepmake/stepmake/install-rules.make
include ./make/generic-targets.make ./make/lilypond-targets.make
include
  /home/phil/lilypond-git/stepmake/stepmake/generic-targets.make
  /home/phil/lilypond-git/stepmake/stepmake/toplevel-targets.make
  /home/phil/lilypond-git/stepmake/stepmake/po-targets.make
  /home/phil/lilypond-git/stepmake/stepmake/install-targets.make
```

‘lilypond-rules.make’ is `#empty`

‘generic-rules.make’ does seem to have 2 rules in it. They are:

```
$(outdir)/%.ly: %.lym4
$(M4) $< | sed "s/\`/,/g" > $
```

```
$(outdir)/%: %.in
rm -f $ cat $< | sed $(sed-atfiles) | sed $(sed-atvariables) > $
```

I believe the first rule is for *.ly files, and has a prerequisite that *.lym4 files must be built first. The recipe is `m4 | sed "s/\`/,/g" >`. Perhaps someone with more Unix/make knowledge can comment on exactly what the rules mean/do.

'`toplevel-rules.make`' is `#empty`

'`po-rules.make`' is `#empty`

'`install-rules.make`' is `#empty`

'`generic-targets.make`' contains 2 lines of comments.

'`lilypond-targets.make`' contains only:

```
## TODO: fail dist or web if no \version present.
check-version:
grep -L version $(LY_FILES)
```

'`stepmake/generic-targets.make`' contains lots of rules - too many to list here - it seems to be the main file for rules. (FWIW I haven't actually found a rule for website: anywhere, although it clearly exists. I have also found that you can display a rule in the terminal by typing, say `make -n website`. This is probably common knowledge.

'`stepmake/toplevel-targets.make`' adds a load of other (and occasionally the same) rules to the gernric-targets.

'`stepmake/po-targets.make`' is rules for po* makes.

'`stepmake/install-targets.make`' has rules for local-install*.

And that's the end of stepmake.make. Back to '`GNUmakefile.in`'.

A bit more info from 27 March. I've put some error traces into `GNUmakefile` in the build directory, and it looks like the following lines actually cause the make to run (putting an error call above them - no make; below them - make):

```
ifeq ($(out),www)
# All web targets, except info image symlinks and info docs are
# installed in non-recursing target from TOP-SRC-DIR
install-WWW:
-$(INSTALL) -m 755 -d $(DESTDIR)$(webdir)
rsync -rl --exclude='*.signature' $(outdir)/offline-root $(DESTDIR)$(webdir)
$(MAKE) -C Documentation omf-local-install
```

I don't currently understand the `ifeq`, since `$(out)` is empty at this point, but the line starting `-$(INSTALL)` translates to:

```
-/usr/bin/python /home/phil/lilypond-git/stepmake/bin/install.py -c -m 755 -d /usr/loca
```

End of work for Sunday 27th.

Another alterative approach to understanding the website build would be to redirect `make -n website` and `make website` to a text file and work through a) what it does and b) where the errors are occurring.

GP: wow, all the above is much more complicated than I've ever looked at stuff – I tend to do a "back first" approach (where I begin from the command-line that I want to modify, figure out where it's generated, and then figure out how to change the generated command-line), rather than a "front first" (where you begin from the "make" command).

12.4 Doc build

12.4.1 Building a bibliography

Bibliography files contain a list of citations, like this:

```
@Book{vinci,
  author = {Vinci, Albert C.},
  title = {Fundamentals of Traditional Music Notation},
  publisher = {Kent State University Press},
  year = {1989}
}
```

There are a variety of types of citation (e.g. Book (as above), article, publication). Each cited publication has a list of entries that can be used to identify the publication. Bibliograpies are normally stored as files with a .bib extension. One part of the doc-build process is transforming the bibliography information into `texinfo` files. The commands to do this are in the '`GNUmakefile`' in the '`Documentation`' directory.

A typical line of the makefile to translate a single bibliography is:

```
$(outdir)/colorado.itexi:
BSTINPUTS=$(src-dir)/essay $(buildscript-dir)/bib2texi \
-s $(top-src-dir)/Documentation/lily-bib \
-o $(outdir)/colorado.itexi \
$(src-dir)/essay/colorado.bib
```

Line by line:

```
$(outdir)/colorado.itexi:
```

We're making the file '`colorado.itexi`' and so this is the make instruction.

```
BSTINPUTS=$(src-dir)/essay $(buildscript-dir)/bib2texi \
```

It's in the '`essay`' directory and we want to run the bib2texi.py script against it.

```
-s $(top-src-dir)/Documentation/lily-bib \
```

The style template is '`lily-bib.bst`' and is found in the '`Documentation`' directory.

```
-o $(outdir)/colorado.itexi \
```

The output file in '`colorado.itexi`'.

```
$(src-dir)/essay/colorado.bib
```

The input file is '`colorado.bib`' in the '`essay`' directory.

The `bib2texi` Python script used to be used with a variety of options, but now is always called using the same options, as above. Its job is to create the file containing the options for `bibtex` (the program that actually does the translation), run bibtex, and then clean up some temporary files. Its main "value add" is the creation of the options file, using this code:

```
open (tmpfile + '.aux', 'w').write (r'''
\relax
\citation{*}
\bibstyle{%(style)s}
\bibdata{%(files)s}''' % vars ())
```

The key items are the style file (now always lily-bib for us) and the input file.

The style file is written in its own specialised language, described to some extent at

http://amath.colorado.edu/documentation/LaTeX/reference/faq/bibtex.pdf

The file '`lily-bib.bst`' also has fairly extensive commenting.

12.5 Website build

Start here: 'make/website.make'

The overall build system begins with Section 12.3.1 [How stepmake works], page 121. Summary: when you type make website this ends up running 'GNUmakefile.in' in the 'git' directory. Right at the bottom, this has the lines:

```
# we want this separate for security; see CG 4.2.  -gp
website:
$(MAKE) config_make=$(config_make) \
top-src-dir=$(top-src-dir) \
-f $(top-src-dir)/make/website.make \
website
```

On my system this expands to:

```
make --no-builtin-rules config_make=./config.make \
top-src-dir=/home/phil/lilypond-git \
-f /home/phil/lilypond-git/make/website.make \
website
```

We see that the $(MAKE) expands to make --no-builtin-rules which is how MAKE is defined higher up the makefile. The -f switch defines the makefile to be used - in this case 'git/make/website.make'. That's where all the action happens.

We believe that note that *none* of the variables that are loaded (from depth to version numbers to whatever) are used in 'website.make'. Instead, 'website.make' sets up its own variables at the top of the file. If you're wondering if there's some smart reason for this, then the answer is "no". It's because I (GP) didn't know/trust the original variables when I was writing that file.

Website build includes Section 12.4.1 [Building a bibliography], page 124.

Output from make -n website

Sorry, including this output directly produces problems in the build system. Please run:

```
make -n website &> my-file.txt
```

to see the full output from the make.

website.make variables

The file begins by setting up some variables. These may/might/probably mirror existing variables, but lacking any docs about those variables, I thought it would be simpler to keep everything in the same file.

Note that for security reasons, we **don't** call scripts in the git dir when building on the web server. See Section 6.2 [Uploading and security], page 71. So we definitely want to keep those definitions for the WEBSITE_ONLY_BUILD.

After some split WEBSITE_ONLY_BUILD vs. normal build definitions, there's another bunch of lines setting up generic variables.

website.make building parts

Parts of 'website.make':

- website: this is the "master" rule. It calls the other rules in order, then copies some extra files around - see below for further of the process it produces.
- website-version: this calls the python scripts below:
  -

```
scripts/build/create-version-itexi.py
```

This writes a @version, @versionStable, and @versionDevel based on the top-level VERSIONS file, to `out-website/version.itexi`

- ```
  scripts/build/create-weblinks-itexi.py
  ```

  This creates a ton of macros in `out-website/weblinks.itexi`. Stuff like @downloadStableLinuxNormal, @downloadStableWidows, `@stableDocsNotationPdf{}`, @downloadDevelSourch-zh.

  It's quite monstrous because it deals with combinations of stable/devel, source/docs, lang/lang/lang*10, etc.

- `website-xrefs:` creates files used for complicated "out-of-build" references to `out-website/*.xref-map`

  If you just write @ref{}, then all's groovy and we wouldn't need this. But if you write @rlearning{}, then our custom texi2html init file needs to know about our custom xref file format, which tells our custom texi2html init file how to create the link.

  GP: we should have a separate @node to discuss xrefs. Also, take a quick look at a generated xref file – it's basically just a list of @node's [sic teenager pluralization rule] from the file.

- `website-bib:` generates the bibliography texinfo files from the .bib files - in the case of the website build these are '`others-did.bib`' and '`we-wrote.bib`'.

- `website-texinfo:` this is the main part; it calles texi2html to generate the actual html. It also has a ton of options to texi2html to pass info to our custom init file.

  We have somewhere between 2-4 different ways "to pass info to our custom init file". This is highly Not Good (tm), but that's how things work at the moment.

  After texi2html, it does some black magick to deal with untranslated nodes in the translations. Despite writing that part, I can't remember how it works. But in theory, you could figure it out by copy&pasting each part of the command (by "part", I mean "stuff before each | pipe"), substituting the variables, then looking at the text that's output. For example,

  ```
  ls $(OUT)/$$l/*.html
  ```

  is going to print a list of all html files, in all languages, in the build directory. Then more stuff happens to each of those files (that's what xargs does).

- `website-css:` just copies files to the build dir.

- `website-pictures, website-examples:` more file copies, with an if statement to handle if you don't have any generated pictures/examples.

- `web-post:` runs:

  ```
  scripts/build/website_post.py
  ```

  which, it adds the "this page is translated in klingon" to the bottom of html pages, and adds the google analytics javascript. It also has hard-coded lilypond version numbers, which is Bad (tm).

Here's a summary of what gets called, in what order, when we run `make website`

```
website:
  website-texinfo:
    website-version:
      creates version.itexi and weblinks.itexi
    website-xrefs:
      runs extract_texi_filenames.py
    website-bibs:
```
```

```
    creates bibliography files, described above
website-css:
  copies css files
website-pictures:
  copies pictures
website-examples:
  copies examples
web-post:
  runs website_post.py
Then some file copying
```

12.6 Building an Ubuntu distro

1. Install ubuntu, reboot
2. Run all updates, reboot if asked
3. Enable src repos, refresh package lists
4. Install LilyPond build deps:

   ```
   sudo apt-get build-dep lilypond
   ```
5. Install git and autoconf:

   ```
   sudo apt-get install git-core gitk autoconf
   ```
6. TEST TO SEE WHETHER EVERYTHING WORKS NOW:
   1. Use lily-git.tcl to grab source files
   2. Go to source dir and do "./autogen.sh" ; make ; make doc
   3. If all compiles, move on to iso creation...
7. Download & install "remastersys":

   ```
   http://sourceforge.net/projects/remastersys/
   ```
8. Copy lily-git.tcl script file into /etc/skel/
9. Modify /etc/remastersys.conf as desired (change .iso name, default live session username, etc)
10. Remove non-essential desktop software as desired
11. Create iso: sudo remastersys dist
12. New iso is in /home/remastersys/remastersys/
13. Test iso by installing in VM and repeating steps above for getting source files and building lp and docs

13 Administrative policies

This chapter discusses miscellaneous administrative issues which don't fit anywhere else.

13.1 Meta-policy for this document

The Contributor's Guide as a whole is still a work in progress, but some chapters are much more complete than others. Chapters which are "almost finished" should not have major changes without a discussion on `-devel`; in other chapters, a disorganized "wiki-style dump" of information is encouraged.

Do not change (other than spelling mistakes) without discussion:

- Chapter 1 [Introduction to contributing], page 1
- Chapter 3 [Working with source code], page 11

Please dump info in an appropriate `@`section within these manuals, but discuss any large-scale reorganization:

- Chapter 4 [Compiling], page 34
- Chapter 5 [Documentation work], page 45
- Chapter 8 [Issues], page 78
- Chapter 9 [Regression tests], page 86
- Chapter 10 [Programming work], page 91

Totally disorganized; do whatever the mao you want:

- Chapter 6 [Website work], page 71
- Chapter 7 [LSR work], page 74
- Chapter 11 [Release work], page 115
- Chapter 13 [Administrative policies], page 128

13.2 Meisters

We have four jobs for organizing a team of contributors:

- Bug Meister: trains new Bug Squad volunteers, organizes who works on which part of their job, checks to make sure that everything is running smoothly, and has final say on our policy for Issues.

  Currently: Phil
- Doc Meister: trains new doc editors/writers, organizes who works on which part of the job, checks to make sure that everything is running smoothly, and has final say on our policy for Documentation. Also includes LSR work.

  Currently: Graham
- Translation Meister: trains new translators, updates the translation priority list, and handles merging branches (in both directions).

  Currently: Francisco
- Frog Meister: is responsible for code patches from (relatively) inexperienced contributors. Keeps track of patches, does initial reviewing of those patches, sends them to `-devel` when they've had some initial review on the Frog list, pesters the `-devel` community into actually reviewing said patches, and finally pushes the patches once they're accepted. This person is *not* responsible for training new programmers, because that would be far too much work – he job is "only" to guide completed patches through our process.

  Currently: Carl

13.3 Administrative mailing list

An mailing list for administrative issues is maintained at `lilypond-hackers@gnu.org`.

This list is intended to be used for discussions that should be kept private. Therefore, the archives are closed to the public.

Subscription to this list is limited to certain senior developers.

At the present time, the list is dormant.

Details about the criteria for membership, the types of discussion to take place on the list, and other policies for the hackers list will be finalized during the Section 13.4 [Grand Organization Project (GOP)], page 129.

13.4 Grand Organization Project (GOP)

GOP has two goals:

- Clarify the various development tasks by writing down the polices and techniques and/or simplifying the tasks directly.
- Get more people involved in development: specifically, find people to do easy tasks to allow advanced developers to concentrate on difficult tasks.

13.4.1 Motivation

Most readers are probably familiar with the LilyPond Grand Documentation Project, which ran from Aug 2007 to Aug 2008. This project involved over 20 people and resulted in an almost complete rewrite of the documentation. Most of those contributors were normal users who decided to volunteer their time and effort to improve lilypond for everybody. By any measure, it was a great success.

The Grand Organization Project aims to do the same thing with a larger scope – instead of focusing purely on documentation, the project aims to improve all parts of LilyPond and its community. Just as with GDP, the main goal is to encourage and train users to become more involved.

If you have never contributed to an open-source project before – especially if you use Windows or OSX and do not know how to program or compile programs – you may be wondering if there's anything you can do. Rest assured that you *can* help.

"Trickle-up" development

One of the reasons I'm organizing GOP is "trickle-up" development. The idea is this: doing easy tasks frees up advanced developers to do harder tasks. Don't ask "am I the *best* person for this job"; instead, ask "am I *capable* of doing this job, so that the current person can do stuff I *can't* do?".

For example, consider lilypond's poor handling of grace notes in conjunction with clef and tempo changes. Fixing this will require a fair amount of code rewriting, and would take an advanced developer a few weeks to do. It's clearly beyond the scope of a normal user, so we might as well sit back and do nothing, right?

No; we *can* help, indirectly. Suppose that our normal user starts answering more emails on lilypond-user. This in turn means that documentation writers don't need to answer those emails, so they can spend more time improving the docs. I've noticed that all doc writers tackle harder and harder subjects, and when they start writing docs on scheme programming and advanced tweaks, they start contributing bug fixes to lilypond. Having people performing these easy-to-moderate bug fixes frees up the advanced developers to work on the really hard stuff... like rewriting the grace note code.

Having 1 more normal user answering emails on lilypond-user won't have a dramatic trick-up affect all by himself, of course. But if we had 8 users volunteering to answer emails, 6 users

starting to write documentation, and 2 users editing LSR... well, that would free up a lot of current bug-fixing-capable contributors to focus on that, and we could start to make a real dent in the number of bugs in lilypond. Quite apart from the eased workload, having that many new helpers will provide a great moral boost!

13.4.2 Ongoing jobs

Although GOP is a short-term project, the main goal is to train more people to handle ongoing jobs. The more people doing these jobs, the ligher the work will be, and the more we can get done with lilypond!

Also, it would be nice if we had at least one "replacement" / "understudy" for each role – too many tasks are only being done by one person, so if that person goes on vacation or gets very busy with other matters, work in that area grinds to a halt.

Jobs for normal users

- Consultant: LilyPond is sometimes critized for not listening to users, but whenever we ask for opinions about specific issues, we never get enough feedback. This is somewhat aggravating. We need a group of users to make a dedicated effort to test and give feedback. If there's new documentation, read it. If there's an experimental binary, download it and try compiling a score with it. If we're trying to name a new command, think about it and give serious suggestions.
- lilypond-user support: I think it would be nice if we had an official team of users helping other users.
- LilyPond Report: Keeping a monthly newsletter running is a non-trivial task. A lot of work is needed to organize it; it would be great if we could split up the work. One person could write the Snippet of the Month, another person could do Quotes of the Month, another person could do interviews, etc.
- Documentation: Although GDP (the Grand Documentation Project) did great work, there's still many tasks remaining.
- Translations: Keeping the documentation translations is a monumental task; we need all the help we can get!

Jobs for advanced users for developers

- Git help for writers: We often receive reports of typos and minor text updates to the documentation. It would be great if somebody could create properly-formatted patches for these corrections.

  Technical requirements: ability to run Section 2.1 [Lilydev], page 5.
- LSR editor: LSR contains many useful examples of lilypond, but some snippets are out of date and need updating. Other snippets need to be advertized, and new snippets need to be sorted. We could use another person to handle LSR.

  Technical requirements: use of a web browser. LilyPond requirements: you should be familiar with most of Notation chapters 1 and 2 (or be willing to read the docs to find out).
- Join the Frogs: "Frogs" are a team of bug-fixers (because frogs eat bugs, and you often find them in Ponds of Lilies) and new feature implementors.

  Technical requirements: development environment (such as Section 2.1 [Lilydev], page 5), ability to read+write scheme and/or C++ code.

13.4.3 Policy decisions

There are a number of policy decisions – some of them fairly important – which we have been postponing for a few years. When GOP begins, we will start discussing them.

> **Note:** The fact that we are not arguing about them right now is not, I repeat **not**, an indication that we do not feel that these issues are not important. It is simply that if we began talking about them now, it would postpone the 2.14 release for a few months.

Note that the presence of an item on this list does *not* mean that everybody thinks that something needs to be done. Inclusion in this simply means that one developer thinks that we should discuss it. We are not going to filter this list; if any developer thinks we should discuss something, just add it to the bottom of the list. (the list is unsorted)

Once GOP starts, the list will be sorted into a rough agenda. We will probably introduce one topic each week – yes, it will therefore take months to get through everything, but we must balance productive work vs. policy administration. If we find that we settle questions faster (or slower) than predicted, we will of course change the speed of new topic introductions.

There are some item(s) not displayed here; these are questions that were posed to me privately, and I do not feel justified in discussing them publicly without the consent of the person(s) that brought them up. They will initially be discussed privately on the lilypond-hackers mailing list – but the first question will be "do we absolutely need to do this privately", and if not, the discussion will take place on lilypond-devel like the other items.

In most policy discussions in lilypond over the past few years, the first half (or more) is wasted arguing on the basis of incorrect or incomplete data; once all the relevant facts are brought to light, the argument is generally resolved fairly quickly. In order to keep the GOP discussions focused, each topic will be introduced with a collection of relevant facts and/or proposals. It is, of course, impossible to predict exactly which facts will be relevant to the discussion – but spending an hour or two collecting information could still save hours of discussion.

> **Note:** The estimated time required for "prep work", and the following discussion, has been added to each item. At the moment, there is an estimated 30 hours of prep work and 140 hours of discussion.

- **Patch reviewing**: At the time of this writing, we have 23 (known) patches waiting for review. Some from main developers; some from new developers. We desperately need more people helping with lilypond, but ignoring patches is the best way to drive potential contributors away. This is not good.

  (prep: 2 hours. discuss: 10 hours)
- **Lessons from the 2.14 release; future release policy**: What went well; what went badly? (how) should we change any policies pertaining to releases? Should an undocumented new feature count as release-blocking?

  (prep: 1 hour. discuss: 15 hours)
- **lilypond-hackers mailing list**: Should we have a private mailing list for senior developers? If so, who should be on it?

  (prep: 2 hours+3 weeks. discuss: 10 hours)
- **Hackers B**:
- **Code style**: New contributors sometimes struggle to follow our indentation and code style – this is especially difficult when parts of our existing source code doesn't have a consistent style. This is problematic... we want new contributors to be struggling with the lilypond architecture, not playing games in their text editors! (ok, we don't actually want them to be struggling with lilypond internals... but given the current state of the CG, it's understandable, and at any rate it's still better than struggling with code style) Speaking academically, C++ code style is a "solved problem". Let's pick one of the existing solutions (probably either astyle, uncrustify, or emacs), and let a computer deal with this.

  (prep: 5 hours. discuss: 15 hours)

- **Git repository(s)**: We currently have a web/ branch in our main repo; this seems misleading to new developers. More generally, should we have branches that aren't related to the master? i.e. should we restrict a git branch to code which is an actual "branch" of development? Also, some of our code (notably the windows and osx lilypad) isn't in a git repository at all. We can add new repositories very easily; should make repositories like

  ```
  git://git.sv.gnu.org/lilypond/gub.git
  git://git.sv.gnu.org/lilypond/lilypad.git
  git://git.sv.gnu.org/lilypond/misc.git
  ```

  ? More information here: http://code.google.com/p/lilypond/issues/detail?id=980

  (prep: 2 hours. discuss: 10 hours)
- **Roadmap of future development**: Many projects have a roadmap of planned (or desired) future work. Should we use one? If so, what should go on it, bearing in mind our volunteer status? Is there any way of having a roadmap that isn't vaporware?

  (prep: 1 hour. discuss: 5 hours)
- **Official links to other organizations?**: There's something called the "software freedom conservancy", and in general, there's a bunch of "umbrella organizations". Joining some of these might give us more visibility, possibly leading to more users, more developers, maybe even financial grants or use in schools, etc.

  (prep: 2 hours. discuss: 5 hours)
- **Mailing lists**: We currently have a mix of official GNU mailing lists and lilynet lists. Is there a strong rationale for having separate mailing list servers? Why not pick one place, and put all our lists there? (or at least, all "permanent" lists?)

  (prep: 1 hour. discuss: 5 hours)
- **Issue tracking with google code**: We use the google issue tracker, but this means that we are relying on a commercial entity for a large part of our development. Would it be better (safer in the long run) to use the savannah bug tracker?

  (prep: 1 hour. discuss: 5 hours)
- **Patch review tool**: Reitveld is inconvenient in some respects: it requires a google account, and there's no way to see all patches relating to lilypond. Should we switch to something like gerritt? http://code.google.com/p/lilypond/issues/detail?id=1184

  (prep: 5 hours. discuss: 15 hours)
- **Subdomains of *.lilypond.org**: Unless Jan has a really weird DNS hosting setup, there are no technical barriers to having names like lsr.lilypond.org, frog.lilypond.org, or news.lilypond.org. Is this something that we want to do?

  (prep: 1 hours+2 weeks. discuss: 5 hours)
- **Authorship in source files**: Our documentation currently does not attempt to track individual authors of each file, while our source code makes a confused and jumbled attempt to track this. A number of guidelines for F/OSS projects explicitly recommends _not_ tracking this in individual files, since the code repository will track that for you.

  (prep: 2 hours. discuss: 15 hours)
- **Clarity for sponsorships**: We currently do not advertize bounties and sponsorships on the webpage. How much advertising do we want, and what type? Should we change the "structure" / "framework" for bounties?

  (prep: 2 hours. discuss: 10 hours)
- **Separate branches for active development**: it might be good to have *everybody* working on separate branches. This complicates the git setup, but with sufficient logic in lily-git.tcl, we can probably make it transparent to newbies. However, we'd need a reliable person to handle all the required merging and stuff.

(prep: 2 hours. discuss: 10 hours)

- **Precise definition of Critical issues**: at the moment, a stable release is entirely dependent on the number of Critical issues, but there's some questions about precisely what a "Critical issue" should be. We should clarify this, in conjunction with a general discussion about how often we want to have stable releases, how permissive we want to be about patches, etc etc.

  (prep: 1 hour. discuss: 5 hours)

- **When do we add regtests?**: There is a discrepancy between our stated policy on adding regtests, and our actual practice in handling bugs and patches. Clarify.

  There is also a wider question how to organize the regtests, such as where to put interesting-console-output regtests, including stuff like lilypond-book and midi2ly in a sensible manner, and possibly including regtests for currently-broken functionality.

  (prep: 2 hours. discuss: 5 hours)

13.5 Grand LilyPond Input Syntax Standardization (GLISS)

Summary

- Start: sortly after 2.14 comes out, which is currently estimated to happen in January 2011.
- Length: 6-12 months. We're not going to rush this.
- Goal: define an input which we commit to being machine-updateable for the forseeable future. Any future patches which change the syntax in a non-convert-ly-able format will be rejected. (subject to the limitations, below) Once this is finished, we will release lilypond 3.0.

The Problem

One of the biggest complaints people have with lilypond – other than silly thing like "there's no gui" – is the changing syntax. Now, inventing a language or standards is difficult. If you set it in stone too soon, you risk being stuck with decisions which may limit matters. If you keep on updating the syntax, interaction with older data (and other programs!) becomes complex.

Scope and Limitations

- tweaks will not be included. Anything with \override, \set, \overrideProperty, \tweak, \revert, \unset... including even those command names themselves... is still fair game for NOT_SMART convert-ly updates.
- other than that, everything is on the table. Is it a problem to have the tagline inside \header? What should the default behavior of \include be? When we abolish \times, do we move to \tuplet 3:2 or \tuplet 2/3 or what (for typical triplets in 4/4 time)?
- we need to get standards for command names. This will help users remember them, and reduce the options for future names (and potential renamings later on). \commandOn and \commandOff seem to work well (should we *always* have an Off command?), but what about the "command" part? Should it be \nounVerbOn, or \verbNounOn ? Or \verbNotesWithExtraInformationOn ?
- we need standards for the location of commands. Ligature brackets, I'm looking at you. (non-postfix notation must die!)
- this Grand Project doesn't affect whether we have a 2.16 or not. The main problem will be deciding what to do (with a bit of messiness anticipated for \tuplet); we should definitely release a 2.16 before merging _any_ of these changes.

- we obviously can't /guarantee/ that we'll /never/ make any non-convert-ly changes in the basic format. But we *can* guarantee that such changes would force lilypond 4.0, and that we would only do so for overwhelmingly good reasons.

Workflow

- We're going to have lots and lots of emails flying around. The vast majority won't really fit into either -devel or -user, so we'll use a list devoted to syntax issues.
- Once we have a serious proposal that gained general acceptance from the separate syntax mailing list, I'll bring it to -devel. We're not going to make any changes without discussing it on -devel, but if we're going to have huge threads about English grammar and silly ideas, and I don't want to clutter up -devel. Once whatever chaotic silliness on the syntax list is settled down, I'll bring the ideas to -devel.
- as with GDP, I'll moderate the discussion. Not as with mailist moderation, but rather by introducing issues at specific times. We don't want a free-for-all discussion of all parts of the syntax at once; nothing will get resolved.
- Whenever possible, we'll decide on policies at the highest level of abstraction. For example, consider \numericTimeSignature, \slurUp, \xNotesOn, \startTextSpan, and \verylongfermata. One of them starts with the name of the notation first (slur). One has an abbreviation (x instead of cross). One has the verb at the end (On), another has it at the beginning (start). The adjective can come at the beginning (numeric, x) or end (Up). Most are in camelCase, but one isn't (verylongfermata).
- Instead of arguing about each individual command, we'll decide on abstract questions. Should each command begin the notation-noun, or the verb? Should all commands be in camelCase, or should we make everything other than articulations in camelCase but make articulations all lower-case? Are abbreviations allowed?
- Once we've answered such fundamental questions, most of the syntax should fall into place pretty easily. There might be a few odd questions left ("is it a span, or a spanner?"), but those can be settled fairly quickly.

Implementation

Nothing until the project is finished, then we declare the next stable release (2.16.0 or 2.18.0 ?) to be the final 2.x version, release it, then apply all the GLISS syntax changes and start testing a beta for 3.0 a week or two later.

Discussion

Don't respond to any of the specifics yet. Yes, we all have our pet irritations (like "what's up with \paper and \layout?!"). There will be plenty of time to discuss them once GLISS starts.

That said, we have a list of specific items that people really wanted to have written down. See Section 13.5.1 [Specific GLISS issues], page 134.

13.5.1 Specific GLISS issues

- add regtests for every piece of syntax (not one-command-per-file, but making a few files which, between them, use every single piece of syntax.) This is a great test for convert-ly.
- should GLISS cover suggested conventions? (indentation, one-bar-per-line, etc – the kind of stuff we list for the lilypond formatting in the docs ?)
- how much (if any) syntactic sugar should we add? i.e.

```
  \instrumentName #'foo
% instead of
  \set Staff.instrumentName
```

  ? Carl: maybe yes, Neil: no. (for example, it fails for pianostaff)

- the values that are used as arguments to common used overrides. Sometimes they are a symbol (e.g. #'around), sometimes a predefined variable referring to a Scheme value or object (e.g. #LEFT, #all-visible). The main trouble is that for novice users it is not clear when there should be an apostrophe and when not.
- When do we need -\command and when is it just \command ?
- Command-line options to the lilypond binary. -dfoo counts as a tweak; we won't be trying to pin those down.
- ```
  \layout {
    \context { \Score
  % vs.
  \layout {
    \context {
      \Score
  ```
- If would be pedagogically simpler to realize this difference if the syntax was separate if you define a context from scratch (as is the case with \RemoveEmptyStaffContext) or if it's defined by adding onto an existing context. For example, a syntax like

  ```
  \context{
   % Copy the current settings of the Staff context:
   \use Staff
   % do whatever additional settings
  }
  %%% could be used to distinguish from
  \context{
   % Take settings from a variable:
   \Variable
   % do whatever additional settings
  }

  %%% and

  \context{
   % Start from scratch:
   \type ...
   \name ...
   \consists ...
   ...
  }
  ```
- Capitalization of identifiers: \VoiceOne ?
- ```
  %%% Allow
  { music expression } * 4
  %%% instead of
  \repeat unfold 4 { music expression }
  ```
  ? patch here: http://lists.gnu.org/archive/html/lilypond-devel/2010-04/msg00467.html
- Personally, I find it easier to understand when there's a repeated 8 in the half-bar position; it's much easier to see that you have two groups of 4:

  ```
  c8 c c c c8 c c c
  %%% instead of one group of eight:
  c8 c c c c c c c
  ```

- trivially simple bar-lines:

  c1 | c1 |

  encourage, allow, or discourage, or disallow?
- indentation of \\ inside a {} construct.
- barline checks at the end of line should be preceded by at least 2 spaces? barline checks should line up if possible (i.e. if you can use less than 4, 8, X empty spaces before a barline check to make them line up?)
- Why doesn't \transpose respect \relative mode?
- on \score vs. \new Score

  But in the light of a consistent syntax and semantic, I see no reason (from the users POV) to disallow it. After all, the real top-level context is a \book {}, isn't it, and I don't see a point in disallowing a \new Score construct just like \new Staff.

  From a syntactical POV, I see the following pros for \new Score: - You can write \with { ... } for every other context but \Score, which (for consistency) should also work with \new Score. - When there's a \new Foo Bar, there's also a \context Foo Bar, which makes the same as a parallel instantiation of all Bar's. - [Quoting Rune from http://www.mail-archive.com/lilypond-devel@gnu.org/msg14713.html "I know that the \score-statement is a syntactical construct, but I think it would be nice to hide this fact from the users. I think we could make the use of score-block much more intuitive if changing the syntax to \new \Score and adding an implicit sequential-statement to the score."
- Discussion on http://code.google.com/p/lilypond/issues/detail?id=1322 about \new vs. \context.
- Let users add their own items to the parser? comment 11 on: http://code.google.com/p/lilypond/issues/detail?id=1322
- should engravers be pluralized (note_heads_engraver) or not (note_head_engraver) ?
- should we allow numbers in identifier names? Issue: http://code.google.com/p/lilypond/issues/detail?id=1
- should we officially allow accented characters? in general, how do we feel about utf-8 stuff?
- for the sake of completeness/simplicity, what about *disallowing* the "one-note" form of a music expression? i.e. only allowing stuff like

  ```
  \transpose c d { e1 }
  \transpose c d << e1 >>
  ```

  and never allowing

  ```
  \transpose c d e1
  ```

13.6 Unsorted policies

Language-specific mailing lists

A translator can ask for an official lilypond-xy mailing list once they've finished all "priority 1" translation items.

Performing yearly copyright update ("grand-replace")

At the start of each year, copyright notices for all source files should be refreshed by running the following command from the top of the source tree:

```
make grand-replace
```

Internally, this invokes the script '`scripts/build/grand-replace.py`', which performs a regular expression substitution for old-year -> new-year wherever it finds a valid copyright notice.

Note that snapshots of third party files such as '`texinfo.tex`' should not be included in the automatic update; '`grand-replace.py`' ignores these files if they are listed in the variable `copied_files`.

Push git access

Git access is given out when a contributor has a significant record of patches being accepted without problems. If existing developers are tired of pushing patches for a contributor, we'll discuss giving them push access. Unsolicited requests from contributors for access will almost always be turned down.

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
http://fsf.org/

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

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful 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, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

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. (Thus, 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. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

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 Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document 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, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. 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, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include 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, PostScript or PDF 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.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

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 (or copies in media that commonly have printed covers) 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 computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. 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 fewer than five), unless they release you from this requirement.

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", Preserve 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. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements 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 to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

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, and that you preserve all their Warranty Disclaimers.

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 "Acknowledgements", 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, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which 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 half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket 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, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

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. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

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 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

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.