LilyPond

The music typesetter

Contributor's Guide

The LilyPond development team

Copyright © 1999–2008 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

Table of Contents

1 Starting with git 1
- 1.1 Getting the source code 1
  - 1.1.1 Git introduction 1
  - 1.1.2 Main source code 1
  - 1.1.3 Website source code 1
  - 1.1.4 Documentation translations source code 1
  - 1.1.5 Other branches 1
  - 1.1.6 Other locations for git 1
  - 1.1.7 Git user configuration 2
- 1.2 Updating the source code 2
  - 1.2.1 Importance of updating 2
  - 1.2.2 Updating command 2
  - 1.2.3 Resolving conflicts 2
- 1.3 Sharing your changes 2
  - 1.3.1 Producing a patch 2
  - 1.3.2 Committing directly 3
- 1.4 Advanced git stuff 3
  - 1.4.1 Introduction to Git concepts 3
  - 1.4.2 Git commands for managing several branches 4
  - 1.4.3 Working on LilyPond sources with several branches 5
  - 1.4.4 Git log 5
  - 1.4.5 Applying git patches 6
- 1.5 Git on Windows 6
  - 1.5.1 Background to nomenclature 6
  - 1.5.2 Installing git 6
  - 1.5.3 Initialising Git 6
  - 1.5.4 Git GUI 7
  - 1.5.5 Personalising your local git repository 7
  - 1.5.6 Checking out a branch 8
  - 1.5.7 Updating files from remote/origin/master 8
  - 1.5.8 Editing files 8
  - 1.5.9 Sending changes to remote/origin/master 9
  - 1.5.10 Resolving merge conflicts 9
  - 1.5.11 Other actions 9

2 Compiling LilyPond 11
- 2.1 Compiling from source 11
- 2.2 Concurrent Stable and Development Versions 11

3 Documentation work 12
- 3.1 Introduction to documentation work 12
- 3.2 Documentation suggestions 12
- 3.3 Texinfo introduction and usage policy 13
  - 3.3.1 Texinfo introduction 13
  - 3.3.2 Documentation files 13
  - 3.3.3 Sectioning commands 14
  - 3.3.4 LilyPond formatting 14

3.3.5 Text formatting ... 15
3.3.6 Syntax survey ... 16
3.3.7 Other text concerns ... 17
3.4 Documentation policy ... 18
3.4.1 Books ... 18
3.4.2 Section organization ... 19
3.4.3 Checking cross-references ... 20
3.4.4 General writing ... 20
3.4.5 Technical writing style ... 20
3.5 Tips for writing docs ... 21
3.6 Updating doc with `convert-ly` ... 22
3.7 Translating the documentation ... 22
3.7.1 Getting started with documentation translation ... 22
Translation requirements ... 22
Which documentation can be translated ... 22
Starting translation in a new language ... 22
3.7.2 Documentation translation details ... 23
Files to be translated ... 23
Translating the Learning Manual and other Texinfo documentation ... 24
Translating the Notation Reference and Application Usage ... 25
Translating the Documentation index '`index.html.in`' ... 26
3.7.3 Documentation translation maintenance ... 26
Check state of translation ... 26
Updating documentation translation ... 26
3.7.4 Translations management policies ... 27
Maintaining without updating translations ... 27
Managing documentation translation with Git ... 29
3.7.5 Technical background ... 29
3.7.6 Translation status ... 30

4 Website work ... 31
4.1 Introduction to website work ... 31
4.2 Translating the website ... 31

5 LSR work ... 32
5.1 Introduction to LSR ... 32
5.2 Adding and editing snippets ... 32
5.3 Approving snippets ... 32
5.4 LSR to Git ... 33
5.5 Fixing snippets in LilyPond sources ... 33
5.6 Updating LSR to a new version ... 34

6 Issues ... 35
6.1 Introduction to issues ... 35
6.2 Issue classification ... 35
6.3 Adding issues to the tracker ... 35

7 Programming work ... 36

7.1 Overview of LilyPond architecture ... 36
7.2 LilyPond programming languages ... 36
7.2.1 C++ ... 36
7.2.2 GNU Bison ... 36
7.2.3 GNU Make ... 37
7.2.4 GUILE or Scheme ... 37
7.2.5 MetaFont ... 37
7.2.6 PostScript ... 37
7.2.7 Python ... 37
7.3 Programming without compiling ... 37
7.3.1 Modifying distribution files ... 37
7.3.2 Desired file formatting ... 37
7.4 Finding functions ... 38
7.4.1 Using the ROADMAP ... 38
7.4.2 Using grep to search ... 38
7.4.3 Using git grep to search ... 38
7.4.4 Searching on the git repository at Savannah ... 38
7.5 Code style ... 38
7.5.1 Handling errors ... 39
7.5.2 Languages ... 39
7.5.3 Filenames ... 39
7.5.4 Indentation ... 39
7.5.5 Classes and Types ... 39
7.5.6 Members ... 40
7.5.7 Macros ... 40
7.5.8 Broken code ... 40
7.5.9 Naming ... 40
7.5.10 Messages ... 40
7.5.11 Localization ... 40
7.6 Debugging LilyPond ... 42
7.6.1 Debugging overview ... 42
7.6.2 Compiling with debugging information ... 42
7.6.3 Typical gdb usage ... 42
7.6.4 Typical .gdbinit files ... 42
7.6.5 Using Guile interactively with LilyPond ... 43

8 Release work ... 44

8.1 Development phases ... 44
8.2 Minor release checklist ... 44
8.3 Major release checklist ... 44
8.4 Making a release ... 45

1 Starting with git

To complete or present in another form the introduction to Git usage in this chapter, it may be a good idea to look for Git documentation at http://git-scm.com/documentation,

1.1 Getting the source code

1.1.1 Git introduction

The source code is kept in a Git respository. This allows us to track changes to files, and for multiple people to work on the same set of files efficiently.

Note: These instructions assume that you are using the command-line version of Git 1.5 or higher. Windows users should skip to Section 1.5 [Git on Windows], page 6.

1.1.2 Main source code

To get the main source code and documentation,

```
mkdir lilypond; cd lilypond
git init-db
git remote add -f -t master -m master origin git://git.sv.gnu.org/lilypond.git/
git checkout -b master origin/master
```

1.1.3 Website source code

To get the website (including translations),

```
mkdir lilypond-web ; cd lilypond-web
git init-db
git remote add -f -t web -m web origin git://git.sv.gnu.org/lilypond.git/
git checkout -b web origin/web
```

1.1.4 Documentation translations source code

To translate the documentation (*not* the website),

```
mkdir lilypond-translation; cd lilypond-translation
git init-db
git remote add -f -t lilypond/translation -m lilypond/translation origin git://git.sv.gnu.org/lilypond.git/
git checkout -b lilypond/translation origin/lilypond/translation
```

1.1.5 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.

- `gub`: This stores the Grand Unified Binary, our cross-platform building tool. For more info, see http://lilypond.org/gub. The git location is:

  `http://github.com/janneke/gub`
- `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.

1.1.6 Other locations for git

If you have difficulty connecting to most of the repositories listed in earlier sections, try:

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

Using HTTP protocol is slowest, so it is not recommended unless both SSH and Git protocols fail, which happens e.g. if you connect to internet through a router that filters out Git and/or SSH connections.

1.1.7 Git user configuration

To configure git to automatically use your name and email address for commits and patches,

```
git config --global user.name "MYNAME"
git config --global user.email MYEMAIL@EXAMPLE.NET
```

1.2 Updating the source code

1.2.1 Importance of updating

In a large project like LilyPond, contributors sometimes edit the same file at the same time. As long as everybody updates their version of the file with the most recent changes (*pulling*), there are generally no problems with this multiple-person editing. However, boring problems can arise if you do not pull before attempting commit, e.g. you may encounter a conflict; in this case, see Section 1.2.3 [Resolving conflicts], page 2.

1.2.2 Updating command

Whenever you are asked to pull, it means you should update your local copy of the repository with the changes made by others on the remote `git.sv.gnu.org` repository:

```
git pull -r
```

1.2.3 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.

1.3 Sharing your changes

1.3.1 Producing a patch

Once you have finished editing your files, checked that your changes meet the Section 7.5 [Code style], page 38, and/or Section 3.4 [Documentation policy], page 18, properly set up your name and email in Section 1.1.7 [Git user configuration], page 2, and checked that the entire thing compiles, you may:

```
git commit -a
git format-patch origin
```

The commit should include a brief message describing the change. This consists of a one-line summary describing the change, and if necessary a blank line followed by several lines giving the details:

```
Did household chores.

I hung up the wet laundry and then washed the car.  I also
vacuumed the floors, rinsed the dirty dishes, fed the cat, and
recalibrated the temporal flux machine.
```

If the change is to the documentation only then the one-line summary should be prefixed with “Docs:”.

If you added a file to the source code, you must add it to git with:

```
git add FILENAME
```

(and possibly modify the '`GNUmakefile`')

These commands will produce one or more files named '`0001-xyz`', '`0002-abc`', etc. in the top directory of the git tree. Send an email to `lilypond-devel@gnu.org` with these files attached, and a developer will review and apply the patches to the main repository.

1.3.2 Committing directly

Most contributors do not have permission to commit directly. If you do, make sure you have set up your name and email in Section 1.1.7 [Git user configuration], page 2, then edit '`.git/config`': change the line

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

into

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

where *user* is your login name on Savannah.

If you have not already done so, you should generate and upload a SSH key: open `https://savannah.gnu.org/my/` in your browser, then go to 'Preferences' then to something like 'Edit SSH Keys', and follow the instructions on that page.

You may then:

```
git push origin
```

1.4 Advanced git stuff

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.

1.4.1 Introduction to Git concepts

A bit of Git vocabulary will be explained below. The following is just introduction material; for better understanding of Git concepts, you are invited to read further documentation, especially Git Community Book at `http://book.git-scm.com/`.

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`, `web` or `lilypond/translation`; if you do not know or remember, see Section 1.1 [Getting the source code], page 1 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 acutally 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 Section 1.1 [Getting the source code], page 1.

```
git remote add -f -t 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.

1.4.2 Git commands for managing several branches

Listing branches and remotes

You can get the exact path or URL of all remotes with 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 1.2.3 [Resolving conflicts], page 2.

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.

1.4.3 Working on LilyPond sources with several 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 origin
```

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 current branch into current branch. For example, if your current branch is `master` — which is the case if you got the sources with the commands described in Section 1.1.2 [Main source code], page 1 and did not issue any `git checkout` command — `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 -l -s -n . 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.

1.4.4 Git log

The commands above don't only bring you the latest version of the sources, but also the full history of revisions (revisons, 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.

1.4.5 Applying git patches

Well-formed git patches should be committed with

`git am`

Patches created without `git format-patch` should be committed with

`git apply`

1.5 Git on Windows

1.5.1 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 respository for LilyPond files. Editors hold local copies 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 are given a unique reference number called the 'Committish' which identifies them 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 complete or as a 'diff' or 'patch' (which send just the differences from master).

1.5.2 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.

1.5.3 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. You will need to have space for around 150Mbytes.

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 -f -t 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, here we call it 'lily-local' - you may wish to make up your own.

Then, finally, type

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

to create the lily-local branch containing the local copies of the master files. You will be advised your local branch has been set up to track the remote branch.

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 Git/Documentation/user.

Terminate the Git bash shell by typing `exit`.

1.5.4 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 two panels on the left contain the names of files which you are in the process of editing (Unstaged Changes), and files you have finished editing and have staged ready for committing (Staged Changes). At this stage 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.

The final 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.

1.5.5 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.

1.5.6 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 lily-local and one tracking branch called origin/master. The latter is your local copy of the remote/origin/master branch in the master LilyPond repository. The lily-local 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.

1.5.7 Updating files from remote/origin/master

Before starting the editing of a file, ensure your local branches contain the latest version in remote/origin/master 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. This will not affect any files you have modified in your local branch.

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

```
Merge -> Local Merge
```

and if necessary select the local branch into which the merge is to be made.

Note that a merge cannot be completed if there are any local uncommitted changes on the lily-local branch.

This will update all the files in that 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).

1.5.8 Editing files

First ensure your lily-local 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 at editing any time.

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

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 lily-local 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 left. 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.

1.5.9 Sending changes to remote/origin/master

If you do not have write access to remote/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 n and the commit message.

1.5.10 Resolving merge conflicts

As soon as you have committed a changed file your local branch has diverged from origin/master, and will remain diverged until your changes have been committed in remote/origin/master and Fetched back into your origin/master. Similarly, if a new commit has been made to remote/origin/master by someone else and Fetched, your lily-local 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 others.

If the diagram at top left of the resulting window does not show your branch's tag on the same node as the remote/origins/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 remote/origin/master and Fetched, or if files modified and committed by others have been Fetched since you last Merged origin/master into your lily-local 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 remote/origin/master for you has added some changes of his own before committing your changes to remote/origin/master, or if someone else has changed the same file since you last fetched the file from remote/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]

1.5.11 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.

2 Compiling LilyPond

2.1 Compiling from source

TODO (see AU 1 for now)

2.2 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

- how to build with debug info
- other compilation tricks for developers

3 Documentation work

3.1 Introduction to documentation work

Our documentation tries to adhere to our Section 3.4 [Documentation policy], page 18. 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.

3.2 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. Here is an example of a perfect documentation report:

```
To: lilypond-devel@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 maillist 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 mailist; 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!

3.3 Texinfo introduction and usage policy

3.3.1 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.

3.3.2 Documentation files

The user manuals lives in '`Documentation/user/`'. In particular, the files '`lilypond-learning.ly`' (LM), '`lilypond.itely`' (NR), '`music-glossary.tely`' (MG), 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); see

the "main" file for each manual to determine the filename of the specific chapter you wish to modify.

Developer manuals live in '`Documentation/devel`'. Currently there is only one; '`contrib-guide.texi`'.

Although snippets are part of documentation, they are not (directly) part of the manuals. For information about how to modify them, see Chapter 5 [LSR work], page 32.

3.3.3 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.
- Sectioning commands (`@node` and `@section`) must not appear inside an `@ignore`. Separate those commands with a space, ie `@n ode`.

3.3.4 LilyPond formatting

- Use two spaces for indentation in lilypond examples. (no tabs)
- All text strings should be prefaced with #. LilyPond does not strictly require this, but it is helpful to get users accustomed to this scheme construct. ie `\set Staff.instrumentName = #"cello"`
- All engravers should have double-quotes around them:

  `\consists "Spans_arpeggio_engraver"`

  Again, LilyPond does not strictly require this, but it is a useful standard to follow.
- Examples should end with a complete bar if possible.
- If possible, only write one bar per line. The notes on each line should be an independent line – tweaks should occur on their own line if possible. Bad:

  ```
  \override textscript #'padding = #3 c1^"hi"
  ```

  Good:

  ```
  \override textscript #'padding = #3
  c1^"hi"
  ```
- Most LilyPond input should be produced with:

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

  or

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

  If you want to use \layout{} or define variables, use

  `@lilypond[verbatim,quote]`

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

excepted in Templates, where 'doctitle' may be omitted.

- Avoid long stretches of input code. Noone is going to read them in print. Please create a smaller example. (the smaller example does not need to be minimal, however)
- 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.
- Add extra spaces around `{ }` marks; ie

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

- If you only have one bar per line, omit bar checks. If you put more than one bar per line (not recommended), then include bar checks.
- If you want to work on an example outside of the manual (for easier/faster processing), use this header:

```
\paper {
  #(define dump-extents #t)
  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.

3.3.5 Text formatting

- Lines should be less than 72 characters long. (I personally recommend writing with 66-char lines, but don't bother modifying existing material.)
- 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`.
- 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 variable@tie{}@var{a} ...
  ```

- To get consistent indentation in the DVI output it is better to avoid the @verbatim environment. Use the @example environment instead if possible, but without extraneous indentation. For example, this

```
@example
  foo {
    bar
  }
@end example
```

should be replaced with

```
@example
foo {
  bar
}
@end example
```

where '@example' starts the line (without leading spaces).

- Do not compress the input vertically; this is, do not use

```
    Beginning of logical unit
    @example
    ...
    @end example
    continuation of logical unit
```

but instead do

```
Beginning of logical unit

@example
...
@end example

@noindent
continuation of logical unit
```

This makes it easier to avoid forgetting the '@noindent'. Only use @noindent if the material is discussing the same material; new material should simply begin without anything special on the line above it.

- in @itemize use @item on a separate line like this:

```
@itemize
@item
Foo

@item
Bar
```

Do not use @itemize @bullet.

- To get LilyPond version, use @version{} (this does not work inside LilyPond snippets). If you write "@version{}" (enclosed with quotes), or generally if @version{} is not followed by a space, tere will be an ugly line break in PDF output unless you enclose it with

```
@w{ ... }

  e.g.

@w{"@version{}"}
```

3.3.6 Syntax survey

- @c - single line comments "@c NOTE:" is a comment which should remain in the final version. (gp only command ;)

- @ignore ... @end ignore - multi-line comment
- @cindex - General index. Please add as many as you can. Don't capitalize the first word.
- @funindex - is for a \lilycommand.
- @example ... @end ignore - example text that should be set as a blockquote. Any {} must be escaped with @{ }@
- @itemize @item A @item B ... @end itemize - for bulleted lists. Do not compress vertically like this.
- @code{} - typeset in a tt-font. Use for actual lilypond code or property/context names. If the name contains a space, wrap the entire thing inside @w{@code{ }}.
- @notation{} - refers to pieces of notation, e.g. "@notation{cres.}". Also use to specific lyrics ("the @notation{A - men} is centered"). Only use once per subsection per term.
- @q{} - Single quotes. Used for 'vague' terms.
- @qq{} - Double quotes. Used for actual quotes ("he said") or for introducing special input modes.
- @tie{} - 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"
- @var - Use for variables.
- @warning{} - produces a "Note: " box. Use for important messages.
- @bs - Generates a backslash inside @warning. Any '\' used inside @warning (and @q or @qq) must be written as '@bs{}' (texinfo would also allow \\, but this breaks with PDF output).
- @ref{} - normal references (type the exact node name inside the {}).
- @ruser{} - link to the NR.
- @rlearning{} - link to the LM.
- @rglos{} - link to the MG.
- @rprogram{} - link to the AU.
- @rlsr{} - link to a Snippet section.
- @rinternals{} - link to the IR.
- @uref{} - link to an external url.

3.3.7 Other text concerns

- References must occur at the end of a sentence, for more information see @ref{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

```
. (ie finish the previous sentence/paragaph)
: (ie `in this example:')
, (ie `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.

3.4 Documentation policy

3.4.1 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.

3.4.2 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}.

    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 FIXME?
- Do not include any real info in second-level sections (ie 1.1 Pitches). A first-level section may have introductory material, but other than that all material goes into third-level sections (ie 1.1.1 Writing Pitches).

3.4.3 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 look for check-xrefs and fix-xrefs targets in 'make help' output. 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/.

3.4.4 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, ie

  ```
  @cindex time signature
  ```

  is preferred instead of "Time signature", Only use capital letters for musical terms which demand them, like D.S. al Fine.

  For scheme functions, only include the final part, i.e.,

  ```
  @funindex modern-voice-cautionary
       and NOT
  @funindex #(set-accidental-style modern-voice-cautionary)
  ```

- Preferred terms:
  - In general, use the American spellings. The internal lilypond property names use this spelling.
  - List of specific terms:

    ```
    canceled
    simultaneous    NOT concurrent
    measure: the unit of music
    bar line: the symbol delimiting a measure   NOT barline
    note head   NOT notehead
    chord construct   NOT chord (when referring to <>)
    ```

3.4.5 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.

3.5 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 5.2 [Adding and editing snippets], page 32.

3.6 Updating doc with `convert-ly`

cd into Documentation and run

```
find . -name '*.itely' | xargs convert-ly -e
```

This also updates translated documentation.

3.7 Translating the documentation

3.7.1 Getting started with documentation translation

First, get the sources from the Git repository, see Section 1.1.4 [Documentation translations source code], page 1.

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.

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 Section 2.1 [Compiling from source], page 11.

Which documentation can be translated

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

- documentation index (HTML);
- user manual and program usage – Texinfo source, PDF and HTML output; Info output might be added if there is enough demand for it;
- the News document.

The following pieces of documentation should be added soon, by descending order of priority:

- automatically generated documentation: markup commands, predefined music functions;
- the Snippets List;
- the examples page;
- 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`'.

Before starting the real translation work, it is recommended to commit changes you made so far to Git, so e.g. you are able to get back to this state of the sources easily if needed; see Section 1.3 [Sharing your changes], page 2.

3.7.2 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. later,
- 6. optional.

Files marked with priority 3, 4 or 5 may be submitted individually. Word counts (excluding LilyPond snippets) are given for each file.

```
-1- Documentation index and Tutorial
429   user/lilypond-learning.tely
6343  user/tutorial.itely
23    user/dedication.itely
432   user/macros.itexi
171   index.html.in
161   translations.template.html.in
6438  po/lilypond-doc.pot (translate to po/MY_LANGUAGE.po)
---   ../lilypond-texi2html.init (section TRANSLATIONS)
13997 total

-2- Introduction and beginning of Application Usage
411   user/preface.itely
3855  user/introduction.itely
407   user/lilypond-program.tely
193   user/install.itely (partial translation)
1149  user/setup.itely
2827  user/running.itely
8842  total

-3- Learning manual
10318 user/fundamental.itely -- Fundamental concepts
14816 user/tweaks.itely -- Tweaking output
4550  user/working.itely -- Working on LilyPond files
498   user/templates.itely -- Templates
30182 total

-4- Notation reference
724   user/lilypond.tely
```

```
91    user/notation.itely -- Musical notation
3145  user/pitches.itely
4772  user/rhythms.itely
1393  user/expressive.itely
555   user/repeats.itely
1456  user/simultaneous.itely
1708  user/staff.itely
905   user/editorial.itely
2376  user/text.itely
76    user/specialist.itely -- Specialist notation
2725  user/vocal.itely
1516  user/chords.itely
702   user/piano.itely
810   user/percussion.itely
826   user/guitar.itely
66    user/strings.itely
242   user/bagpipes.itely
4487  user/ancient.itely
6024  user/input.itely -- Input syntax
2164  user/non-music.itely -- Non-musical notation
8663  user/spacing.itely -- Spacing issues
11747 user/changing-defaults.itely -- Changing defaults
5187  user/programming-interface.itely -- Interfaces for programmers
1182  user/notation-appendices.itely -- Notation manual tables
252   user/cheatsheet.itely -- Cheat sheet
63794 total

-5- Application usage
3248  user/lilypond-book.itely -- LilyPond-book
1171  user/converters.itely -- Converting from other formats
4419  total

-6- Appendices whose translation is optional
310   user/literature.itely
960   user/scheme-tutorial.itely (needs to be revised first)
1270  total
```

Translating the Learning Manual and other Texinfo documentation

Any title which comes with one of the following commands must not be translated directly in the Texinfo source

```
@node                                                  @majorheading
@chapter       @unnumbered          @appendix          @chapheading
@section       @unnumberedsec       @appendixsec       @heading
@subsection    @unnumberedsubsec    @appendixsubsec    @subheading
@subsubsection @unnumberedsubsubsec @appendixsubsubsec @subsubheading
@ref  @rglos  @ruser  @rlearning  @rprogram  @rlsr
```

The same applies to first argument of `@rmanual named` commands; however, the second argument *Bar baz* of `@ref{Foo,Bar baz,,info-file}` and `@rmanual named{Foo,Bar baz}` should be translated.

`@uref`'s names are to be translated.

In any section which looks like

```
@menu
* node1:: thing1
* node2:: thing2
...
@end menu
```

the node names *nodeN* are *not* to be translated, whereas extra title information *thingN* is.

Every node name or section title must from now on be translated separately in a '`.po`' file (just as well as 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 '`user/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 26.

When you encounter

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

in the source, open '`input/lsr/`*filename*`.ly`', translate the `texidoc` header field it contains, enclose it with `texidoc`*MY-LANGUAGE* `= "` and `"`, and write it into '`input/texidocs/`*filename*`.texidoc`' – please keep possibly existing translations in other languages! Additionnally, 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, '`input/texidocs/`*filename*`.texidoc`' may contain

```
doctitlees = "Spanish title baz"
texidoces = "
Spanish translation blah
"
doctitlede = "German title bar"
texidocde = "German translation foo
"
```

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

Index entries (`@cindex` and so on) should be translated.

Finally, please carefully apply every rule exposed in Section 3.3 [Texinfo introduction and usage policy], page 13, and Section 3.4 [Documentation policy], page 18. If one of these rules conflicts with a rule specific to your language, please ask the Translation meister and/or the Documentation Editors on `lilypond-devel@gnu.org`.

Translating the Notation Reference and Application Usage

Copy '`user/lilypond.tely`' (or '`user/lilypond-program.tely`', respectively) into '*MY-LANGUAGE*`/user`', then translate this file and run `skeleton-update` – see [Updating documentation translation], page 26. Your are now ready to translate the Notation Reference (Application Usage, respectively) exactly like the Learning Manual.

Translating the Documentation index 'index.html.in'

Unlike almost all HTML pages in this documentation, links in this page are not tweaked by 'postprocess_html.py', so links should be manually edited to link to existing translations.

3.7.3 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.

Check state of translation

First pull from Git, 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/user/foo.itely check-translation
```

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
```

Global state of the translation is recorded in 'Documentation/translations.html.in', 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 27.

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/user/foo.itely update-translation
```

For each file to be udpated, 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.

Texinfo skeleton files, i.e. '.itely' files not yet translated, containing only the Texinfo structure 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/user/every.itely`*' with music snippets from '*`user/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.

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.

See also

[Maintaining without updating translations], page 27.

3.7.4 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, transloators may be (and that) involved in these porjects too.

it is possible – and even recommended – to perform some maintaining that keeps translated documentation usable and eases future translation updating. The rationale below the tasks list motivates this plan. 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 ouptput to a file because of overwhelming output, or call check-translation.py on individual files, see [Check state of translation], page 26.
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 sucessful 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 succesfully.
- `master` Git branch may be merged into `lilypond/translation` whenever `make` and `make doc` are succesful (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 27.

3.7.5 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/`’:

- ‘`html-gettext.py`’ – translate node names, section titles and cross references in HTML files generated by `makeinfo`,
- ‘`texi-gettext.py`’ – gettext node names, section titles and references before calling `texi2pdf`,
- ‘`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

3.7.6 Translation status

4 Website work

4.1 Introduction to website work

Short answer: don't do it yet. We're completely revamping the website.

4.2 Translating the website

5 LSR work

5.1 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.

5.2 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 3.3.4 [LilyPond formatting], page 14.

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 '`input/new`' and a reference should be added to the manual.

Snippets created or updated in '`input/new`' should be copied to '`input/lsr`' by invoking at top of the source tree

```
scripts/auxiliar/makelsr.py
```

Be sure that `make doc` runs successfully before submitting a patch, to prevent breaking compilation.

Formatting snippets in 'input/new'

When adding a file to this directory, please start the file with

```
\version "2.x.y"
\header {
  lsrtags = "rhythms,expressive-marks"  % use existing LSR tags other than
%   'docs'; see makelsr.py for the list of tags used to sort snippets.
  texidoc = "This code demonstrates ..."  % this will be formated by Texinfo
  doctitle = "Snippet title"  % please put this at the end so that
    the '% begin verbatim' mark is added correctly by makelsr.py.
}
```

and name the file '`snippet-title.ly`'.

5.3 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 3.3.4 [LilyPond formatting], page 14.
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**.

5.4 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 '`input/new`' and the other from LSR with the same file name, the one from '`input/new`' will be copied by `makelsr.py`.

5.5 Fixing snippets in LilyPond sources

In case some snippet from '`input/lsr`' cause 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 '`input/lsr/`*foo*`.ly`' to make the documentation build succesfully.
2. Determine where it comes from by looking at its first line, e.g. run

   ```
   head -1 input/lsr/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 5.2 [Adding and editing snippets], page 32. 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 5.4 [LSR to Git], page 33. 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 '`input/new`', then run `makelsr.py`.
4. **In case the snippet comes from** '`input/new`', apply in '`input/new/`*foo*`.ly`' the same fix you did in '`input/lsr/`*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.

5.6 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 '`input/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 '`input/new/`' were included, then delete those snippets from '`input/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
```

6 Issues

6.1 Introduction to issues

First, “issue” 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.

Second, the classification of what counts as a bug vs. feature request, and the priorities assigned to bugs, are a matter of concern **for developers only**. If you are curious about the classification, read on, but don’t complain that your particular issue is higher priority or counts as a bug rather than a feature request.

6.2 Issue classification

6.3 Adding issues to the tracker

7 Programming work

7.1 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 translators or engravers (the distinction is unclear).

Translators are defined in C++ files named *-engraver.cc. In *-engraver.cc, a C++ class of Engraver type is created. The Engraver is also declared as a translator. Much of the work of translating is handled by Scheme functions, which is one of the keys to LilyPond's exceptional flexibility.

7.2 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.

7.2.1 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.

7.2.2 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.

7.2.3 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.

7.2.4 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 Learning Manual, see Section "Scheme tutorial" in *Learning Manual*.

7.2.5 MetaFont

MetaFont is used to create the music fonts used by LilyPond. A MetaFont tutorial is available at the METAFONT tutorial page.

7.2.6 PostScript

PostScript is used to generate graphical output. A brief PostScript tutorial is available online. The PostScript Lanugage Reference is available online in PDF format.

7.2.7 Python

Python is used for XML2ly and is used for buillding the documentation and the website.

Python documentation is available at python.org.

7.3 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.

7.3.1 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.

7.3.2 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 3.3 [Texinfo introduction and usage policy], page 13.

7.4 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.

7.4.1 Using the ROADMAP

The file ROADMAP is located in the main directory of the lilypond source. ROADMAP lists all of the directories in the LilPond 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.

7.4.2 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.

7.4.3 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.

7.4.4 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.

7.5 Code style

7.5.1 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.

7.5.2 Languages

C++ and Python are preferred. Python code should use PEP 8.

7.5.3 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.*’

7.5.4 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
                 ))
```

7.5.5 Classes and Types

```
This_is_a_class
```

7.5.6 Members

Member variable names end with an underscore:

```
Type Class::member_
```

7.5.7 Macros

Macro names should be written in uppercase completely.

7.5.8 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.

We reject broken-in-advance on principle.

7.5.9 Naming

7.5.10 Messages

Messages need to follow Localization.

7.5.11 Localization

This document provides some guidelines for programmers write 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.
- Don't localize/gettextify:
  - 'programming_error ()'s
  - 'programming_warning ()'s
  - debug strings
  - output strings (PostScript, TeX, etc.)
- Messages to be localised 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. The 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 modularise too much; words frequently cannot be translated without context. It is probably safe to treat most occurences 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

7.6 Debugging LilyPond

The most commonly used tool for debugging LilyPond is the GNU debugger gdb. Use of gdb is described in this section.

7.6.1 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 allows the display of a stack trace, which shows the sequence in which functions are called and the arguments to the various function calls.

7.6.2 Compiling with debugging information

In order to use a debugger with LilyPond, it is necessary to compile LilyPond with debugging information. This is accomplished by ...

TODO – get good description here, or perhaps add debugging compile to AU1.1 as it comes to CG and just use a reference here.

TODO – Test the following to make sure it is true.

If you want to be able to set breakpoints in Scheme functions, it is necessary to compile guile with debugging information. This is done by ...

TODO – get compiling description for guile here.

7.6.3 Typical gdb usage

7.6.4 Typical .gdbinit files

The behavior of gdb can be readily customized through the use of *.gdbinit* files. 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 scm_error
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
```

7.6.5 Using Guile interactively with LilyPond

In order to experiment with Scheme programming in the LilyPond environment, it is convenient 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))
```

Second, 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. When the guile prompt appears, the LilyPond active module must be set as the current guile module:

```
guile> (set-current-module lilypond-module)
```

Proper operation of these commands can be demonstrated by typing the name of a LilyPond public scheme function to see if it's properly 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.

8 Release work

8.1 Development phases

There are 2.5 states of development for LilyPond.

- **Stable phase**: Starting from the release of a new major version `2.x.0`, the following patches **MAY NOT** be merged with master:
  - Any change to the input syntax. If a file compiled with a previous `2.x` version, then it must compile in the new version.
  - 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.
- **Development phase**: Any commits are fine. Readers may be familiar with the term "merge window" from following Linux kernel news.
- **Release prep phase**: FIXME: I don't like that name.

  A new git branch `stable/2.x` is created, and a major release is made in two weeks.
  - `stable/2.x branch`: Only translation updates and important bugfixes are allows.
  - `master`: Normal "stable phase" development occurs.

  If we discover the need to change the syntax or build system, we will apply it and re-start the release prep phase.

This marks a radical change from previous practice in LilyPond. However, this setup is not intended to slow development – as a rule of thumb, the next development phase will start within a month of somebody wanting to commit something which is not permitted during the stable phase.

8.2 Minor release checklist

A "minor release" means an update of `y` in `2.x.y`.

email brief summary to info-lilypond

8.3 Major release checklist

A "major release" means an update of `x` in `2.x.0`.

Before release:

* write release notes. note: stringent size requirements for various websites, so be brief.

* write preface section for manual.

* submit pots for translation : send url of tarball to translation@iro.umontreal.ca, mentioning lilypond-VERSION.pot

* Check reg test

* Check all 2ly scripts.

* Run convert-ly on all files, bump parser minimum version.

* Make FTP directories on lilypond.org

* website: - Make new table in download.html

- add to documentation list

- revise examples tour.html/howto.html

- add to front-page quick links

- change all links to the stable documentation

- doc auto redirects to v2.LATEST-STABLE

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

8.4 Making a release

- Build with GUB, and check the regtests.
- Upload the tarballs and sh scripts.
- (if major) Branch MASTER to stable/2.x.
- Make announcement.