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 2
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
1.2.4 Technical notes 2
1.3 Sharing your changes 3
1.3.1 Producing a patch 3
1.3.2 Committing directly 3
1.4 Other interesting Git commands 3
1.4.1 Git log 3
1.4.2 Applying git patches 4
1.5 Git on Windows 4
1.5.1 Background to nomenclature 4
1.5.2 Installing git 4
1.5.3 Initialising Git 4
1.5.4 Git GUI 5
1.5.5 Personalising your local git repository 5
1.5.6 Checking out a branch 5
1.5.7 Updating files from remote/origin/master 6
1.5.8 Editing files 6
1.5.9 Sending changes to remote/origin/master 6
1.5.10 Resolving merge conflicts 7
1.5.11 Other actions 7

2 Compiling 8
2.1 move AU 1 here 8

3 Documentation work 9
3.1 Introduction to documentation work 9
3.2 Texinfo crash course 9
3.2.1 Sectioning commands 9
3.2.2 LilyPond formatting 10
3.2.3 Text formatting 11
3.2.4 Syntax survey 12
3.2.5 Other text concerns 13
3.3 Documentation policy 13
3.3.1 Books 13
3.3.2 Section organization 14

3.3.3 Checking cross-references ... 15
3.3.4 General writing ... 15
3.3.5 Technical writing style ... 16
3.4 Tips for writing docs ... 16
3.5 Updating doc with convert-ly ... 17
3.6 Translating the documentation ... 17

4 Website work ... 18
4.1 Introduction to website work ... 18
4.2 Translating the website ... 18

5 LSR work ... 19
5.1 Introduction to LSR ... 19
5.2 Adding snippets ... 19
5.3 Approving snippets ... 19
5.4 LSR to git ... 19

6 Issues ... 20
6.1 Introduction to issues ... 20
6.2 Issue classification ... 20
6.3 Adding issues to the tracker ... 20

7 Programming work ... 21
7.1 Introduction to programming ... 21
7.2 Programming without compiling ... 21
7.2.1 Modifying distribution files ... 21
7.2.2 Desired file formatting ... 21
7.3 Finding functions ... 21
7.3.1 Using the ROADMAP ... 21
7.3.2 Using grep to search ... 22
7.3.3 Using git grep to search ... 22
7.3.4 Searching on the git repository at Savannah ... 22
7.4 Code style ... 22
7.4.1 Handling errors ... 22
7.4.2 Languages ... 22
7.4.3 Filenames ... 22
7.4.4 Indentation ... 23
7.4.5 Classes and Types ... 23
7.4.6 Members ... 23
7.4.7 Macros ... 23
7.4.8 Broken code ... 23
7.4.9 Naming ... 23
7.4.10 Messages ... 23
7.4.11 Localization ... 23
7.5 Debugging LilyPond ... 25
7.5.1 Debugging overview ... 25
7.5.2 Compiling with debugging information ... 26
7.5.3 Typical gdb usage ... 26
7.5.4 Typical .gdbinit files ... 26

8 Release work 27
8.1 Development phases 27
8.2 Minor release checklist 27
8.3 Major release checklist 27

1 Starting with git

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 (generally) without any problems.

> **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 4.

1.1.2 Main source code

To get the main source code and documentation,

FIXME: test this!!!

```
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),

FIXME: test this!!!

```
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),

FIXME: change!!!

```
mkdir lilypond-translate; cd lilypond-translate
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
```

1.1.5 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:

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

> **Note:** The `git://` and `ssh://` URLs are intended for advanced git users.

1.1.7 Git user configuration

To configure git to automatically use your name and email address for 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 ("pull"ing), there are generally no problems with this multiple-person editing. However, serious problems can arise if you do not pull before attempting commit.

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 origin
```

1.2.3 Resolving conflicts

Occasionally an update may result in conflicts – this happens when you and somebody else hae 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.

```
TODO
```

1.2.4 Technical notes

TODO: I'm not going to bother with this section. -gp

Let's explain a bit of Git vocabulary. The `git pull origin` command is just a shortcut for this command:

```
git pull git://git.sv.gnu.org/lilypond.git/ MY-BRANCH:origin/MY-BRANCH
```

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'n'paste the committish.

A branch is a tree (in the mathematical or computer science sense) of commits, and the topmost commit of this branch is called a head.

The "git fetch" command above has created a branch called `origin/web` 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' or 'git fetch' with this branch reference as argument, e.g. by using .git/remotes/web remote file when running 'git fetch web'.

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

This README tries to explain most of Git commands needed for translating the web site. However, you are invited to read further documentation to make git more familiar to you; for instance, take a look at `http://git.or.cz/gitwiki/`, especially GitDocumentation and GitGlossary; a good alternative to reading the wiki is reading the first two chapters of Git User's Manual at `http://www.kernel.org/pub/software/scm/git/docs/user-manual.html`

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 7.4 [Code style], page 22 and/or Section 3.3 [Documentation policy], page 13, and checked that the entire thing compiles, you may

```
git commit -a
git-format-patch HEAD
```

Send an email to `lilypond-devel@gnu.org` with the diff as an attachment.

1.3.2 Committing directly

Most contributors do not have permission to commit directly. If you do, edit '`.git/config`' to contain

```
FIXME?  Is anything needed, or did the previous commands set it
up?
```

You may then `git push`.

1.4 Other interesting Git commands

1.4.1 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.

1.4.2 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 install.

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.

In the git bash shell 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.

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 -n
```

where n an integer, normally 1. 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

2.1 move AU 1 here

3 Documentation work

3.1 Introduction to documentation work

Our documentation tries to adhere to our Section 3.3 [Documentation policy], page 13. 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 Texinfo crash course

The language is called texinfo; you can see its manual here: http://www.gnu.org/software/texinfo/manual/

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.

3.2.1 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.

3.2.2 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.

3.2.3 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{}"}
```

3.2.4 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).

3.2.5 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.

3.3 Documentation policy

3.3.1 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.

3.3.2 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).

3.3.3 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/.

3.3.4 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 <>)
```

3.3.5 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.

3.4 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.

3.5 Updating doc with convert-ly

cd into Documentation and run

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

(This also updates translated docs.)

3.6 Translating the documentation

4 Website work

4.1 Introduction to website work

4.2 Translating the website

5 LSR work

5.1 Introduction to LSR

5.2 Adding snippets

5.3 Approving snippets

5.4 LSR to git

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

de

7 Programming work

7.1 Introduction to programming

FIXME – decide what goes in here and put it here. I'm not sure what should be here – CDS

7.2 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.

7.2.1 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.

7.2.2 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

7.3 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 inefective. This section describes a process for finding interesting code.

7.3.1 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.

7.3.2 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 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 most likely directories to grep for function names are scm/ for scheme files, ly/ for lilypond input (*.ly) files, and lily/ for C++ files.

7.3.3 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.

7.3.4 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.

7.4 Code style

7.4.1 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.

7.4.2 Languages

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

7.4.3 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.*'

7.4.4 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
       ))
```

7.4.5 Classes and Types

```
This_is_a_class
```

7.4.6 Members

Member variable names end with an underscore:

```
Type Class::member_
```

7.4.7 Macros

Macro names should be written in uppercase completely.

7.4.8 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.

7.4.9 Naming

7.4.10 Messages

Messages need to follow Localization.

7.4.11 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, ...)'. Eg:

```
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, we'll 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 mustn't 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's 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'd prefer something like (2) or (3).

- Do not run make po/po-update with GNU gettext < 0.10.35

7.5 Debugging LilyPond

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

7.5.1 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.

7.5.2 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.

7.5.3 Typical gdb usage

7.5.4 Typical .gdbinit files

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

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