LilyPond

The music typesetter

Extending

The LilyPond development team

This file explains how to extend the functionality of LilyPond version 2.18.2.

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

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

For LilyPond version 2.18.2

Table of Contents

1 Scheme tutorial

LilyPond uses the Scheme programming language, both as part of the input syntax, and as internal mechanism to glue modules of the program together. This section is a very brief overview of entering data in Scheme. If you want to know more about Scheme, see http://www.schemers.org.

LilyPond uses the GNU Guile implementation of Scheme, which is based on the Scheme "R5RS" standard. If you are learning Scheme to use with LilyPond, working with a different implementation (or referring to a different standard) is not recommended. Information on guile can be found at http://www.gnu.org/software/guile/. The "R5RS" Scheme standard is located at http://www.schemers.org/Documents/Standards/R5RS/.

1.1 Introduction to Scheme

We begin with an introduction to Scheme. For this brief introduction, we will use the GUILE interpreter to explore how the language works. Once we are familiar with Scheme, we will show how the language can be integrated in LilyPond files.

1.1.1 Scheme sandbox

The LilyPond installation includes the Guile implementation of Scheme. On most systems you can experiment in a Scheme sandbox by opening a terminal window and typing 'guile'. On some systems, notably Windows, you may need to set the environment variable `GUILE_LOAD_PATH` to the directory `../usr/share/guile/1.8` in the LilyPond installation. For the full path to this directory see Section "Other sources of information" in *Learning Manual*. Alternatively, Windows users may simply choose 'Run' from the Start menu and enter 'guile'.

However, a hands-on Scheme sandbox with all of Lilypond loaded is available with this command line:

```
lilypond scheme-sandbox
```

Once the sandbox is running, you will receive a guile prompt:

```
guile>
```

You can enter Scheme expressions at this prompt to experiment with Scheme. If you want to be able to use the GNU readline library for nicer editing of the Scheme command line, check the file '`ly/scheme-sandbox.ly`' for more information. If you already have enabled the readline library for your interactive Guile sessions outside of LilyPond, this should work in the sandbox as well.

1.1.2 Scheme variables

Scheme variables can have any valid scheme value, including a Scheme procedure.

Scheme variables are created with `define`:

```
guile> (define a 2)
guile>
```

Scheme variables can be evaluated at the guile prompt simply by typing the variable name:

```
guile> a
2
guile>
```

Scheme variables can be printed on the display by using the display function:

```
guile> (display a)
2guile>
```

Note that both the value `2` and the guile prompt `guile` showed up on the same line. This can be avoided by calling the newline procedure or displaying a newline character.

```
guile> (display a)(newline)
2
guile> (display a)(display "\n")
2
guile>
```

Once a variable has been created, its value can be changed with `set!`:

```
guile> (set! a 12345)
guile> a
12345
guile>
```

1.1.3 Scheme simple data types

The most basic concept in a language is data typing: numbers, character strings, lists, etc. Here is a list of simple Scheme data types that are often used with LilyPond.

Booleans
: Boolean values are True or False. The Scheme for True is `#t` and False is `#f`.

Numbers
: Numbers are entered in the standard fashion, `1` is the (integer) number one, while `-1.5` is a floating point number (a non-integer number).

Strings
: Strings are enclosed in double quotes:

```
"this is a string"
```

Strings may span several lines:

```
"this
is
a string"
```

and the newline characters at the end of each line will be included in the string.

Newline characters can also be added by including `\n` in the string.

```
"this\nis a\nmultiline string"
```

Quotation marks and backslashes are added to strings by preceding them with a backslash. The string `\a said "b"` is entered as

```
"\\a said \"b\""
```

There are additional Scheme data types that are not discussed here. For a complete listing see the Guile reference guide, http://www.gnu.org/software/guile/manual/html_node/Simple-Data-Types.html.

1.1.4 Scheme compound data types

There are also compound data types in Scheme. The types commonly used in LilyPond programming include pairs, lists, alists, and hash tables.

Pairs

The foundational compound data type of Scheme is the `pair`. As might be expected from its name, a pair is two values glued together. The operator used to form a pair is called `cons`.

```
guile> (cons 4 5)
(4 . 5)
guile>
```

Note that the pair is displayed as two items surrounded by parentheses and separated by whitespace, a period (`.`), and more whitespace. The period is *not* a decimal point, but rather an indicator of the pair.

Pairs can also be entered as literal values by preceding them with a single quote character.

```
guile> '(4 . 5)
(4 . 5)
guile>
```

The two elements of a pair may be any valid Scheme value:

```
guile> (cons #t #f)
(#t . #f)
guile> '("blah-blah" . 3.1415926535)
("blah-blah" . 3.1415926535)
guile>
```

The first and second elements of the pair can be accessed by the Scheme procedures `car` and `cdr`, respectively.

```
guile> (define mypair (cons 123 "hello there")
... )
guile> (car mypair)
123
guile> (cdr mypair)
"hello there"
guile>
```

Note: `cdr` is pronounced "could-er", according Sussman and Abelson, see http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-14.html#footnote_Temp_133

Lists

A very common Scheme data structure is the *list*. Formally, a list is defined as either the empty list (represented as `'()`, or a pair whose `cdr` is a list.

There are many ways of creating lists. Perhaps the most common is with the `list` procedure:

```
guile> (list 1 2 3 "abc" 17.5)
(1 2 3 "abc" 17.5)
```

As can be seen, a list is displayed in the form of individual elements separated by whitespace and enclosed in parentheses. Unlike a pair, there is no period between the elements.

A list can also be entered as a literal list by enclosing its elements in parentheses, and adding a quote:

```
guile> '(17 23 "foo" "bar" "bazzle")
(17 23 "foo" "bar" "bazzle")
```

Lists are a central part of Scheme. In, fact, Scheme is considered a dialect of lisp, where 'lisp' is an abbreviation for 'List Processing'. Scheme expressions are all lists.

Association lists (alists)

A special type of list is an *association list* or *alist*. An alist is used to store data for easy retrieval.

Alists are lists whose elements are pairs. The `car` of each element is called the *key*, and the `cdr` of each element is called the *value*. The Scheme procedure `assoc` is used to retrieve an entry from the alist, and `cdr` is used to retrieve the value:

```
guile> (define my-alist '((1  . "A") (2 . "B") (3 . "C")))
guile> my-alist
((1 . "A") (2 . "B") (3 . "C"))
guile> (assoc 2 my-alist)
(2 . "B")
guile> (cdr (assoc 2 my-alist))
"B"
```

```
guile>
```

Alists are widely used in LilyPond to store properties and other data.

Hash tables

A data structure that is used occasionally in LilyPond. A hash table is similar to an array, but the indexes to the array can be any type of Scheme value, not just integers.

Hash tables are more efficient than alists if there is a lot of data to store and the data changes very infrequently.

The syntax to create hash tables is a bit complex, but you can see examples of it in the LilyPond source.

```
guile> (define h (make-hash-table 10))
guile> h
#<hash-table 0/31>
guile> (hashq-set! h 'key1 "val1")
"val1"
guile> (hashq-set! h 'key2 "val2")
"val2"
guile> (hashq-set! h 3 "val3")
"val3"
```

Values are retrieved from hash tables with `hashq-ref`.

```
guile> (hashq-ref h 3)
"val3"
guile> (hashq-ref h 'key2)
"val2"
guile>
```

Keys and values are retrieved as a pair with `hashq-get-handle`. This is a preferred way, because it will return `#f` if a key is not found.

```
guile> (hashq-get-handle h 'key1)
(key1 . "val1")
guile> (hashq-get-handle h 'frob)
#f
guile>
```

1.1.5 Calculations in Scheme

Scheme can be used to do calculations. It uses *prefix* syntax. Adding 1 and 2 is written as `(+ 1 2)` rather than the traditional $1 + 2$.

```
guile> (+ 1 2)
3
```

Calculations may be nested; the result of a function may be used for another calculation.

```
guile> (+ 1 (* 3 4))
13
```

These calculations are examples of evaluations; an expression like `(* 3 4)` is replaced by its value `12`.

Scheme calculations are sensitive to the differences between integers and non-integers. Integer calculations are exact, while non-integers are calculated to the appropriate limits of precision:

```
guile> (/ 7 3)
7/3
guile> (/ 7.0 3.0)
```

```
2.33333333333333
```

When the scheme interpreter encounters an expression that is a list, the first element of the list is treated as a procedure to be evaluated with the arguments of the remainder of the list. Therefore, all operators in Scheme are prefix operators.

If the first element of a Scheme expression that is a list passed to the interpreter is *not* an operator or procedure, an error will occur:

```
guile> (1 2 3)

Backtrace:
In current input:
  52: 0* [1 2 3]

<unnamed port>:52:1: In expression (1 2 3):
<unnamed port>:52:1: Wrong type to apply: 1
ABORT: (misc-error)
guile>
```

Here you can see that the interpreter was trying to treat 1 as an operator or procedure, and it couldn't. Hence the error is "Wrong type to apply: 1".

Therefore, to create a list we need to use the list operator, or to quote the list so that the interpreter will not try to evaluate it.

```
guile> (list 1 2 3)
(1 2 3)
guile> '(1 2 3)
(1 2 3)
guile>
```

This is an error that can appear as you are working with Scheme in LilyPond.

1.1.6 Scheme procedures

Scheme procedures are executable scheme expressions that return a value resulting from their execution. They can also manipulate variables defined outside of the procedure.

Defining procedures

Procedures are defined in Scheme with define

```
(define (function-name arg1 arg2 ... argn)
 scheme-expression-that-gives-a-return-value)
```

For example, we could define a procedure to calculate the average:

```
guile> (define (average x y) (/ (+ x y) 2))
guile> average
#<procedure average (x y)>
```

Once a procedure is defined, it is called by putting the procedure name and the arguments in a list. For example, we can calculate the average of 3 and 12:

```
guile> (average 3 12)
15/2
```

Predicates

Scheme procedures that return boolean values are often called *predicates*. By convention (but not necessity), predicate names typically end in a question mark:

```
guile> (define (less-than-ten? x) (< x 10))
guile> (less-than-ten? 9)
#t
guile> (less-than-ten? 15)
#f
```

Return values

Scheme procedures always return a return value, which is the value of the last expression executed in the procedure. The return value can be any valid Scheme value, including a complex data structure or a procedure.

Sometimes the user would like to have multiple Scheme expressions in a procedure. There are two ways that multiple expressions can be combined. The first is the `begin` procedure, which allows multiple expressions to be evaluated, and returns the value of the last expression.

```
guile> (begin (+ 1 2) (- 5 8) (* 2 2))
4
```

The second way to combine multiple expressions is in a `let` block. In a let block, a series of bindings are created, and then a sequence of expressions that can include those bindings is evaluated. The return value of the let block is the return value of the last statement in the let block:

```
guile> (let ((x 2) (y 3) (z 4)) (display (+ x y)) (display (- z 4))
... (+ (* x y) (/ z x)))
508
```

1.1.7 Scheme conditionals

if

Scheme has an `if` procedure:

```
(if test-expression true-expression false-expression)
```

test-expression is an expression that returns a boolean value. If *test-expression* returns `#t`, the if procedure returns the value of *true-expression*, otherwise it returns the value of *false-expression*.

```
guile> (define a 3)
guile> (define b 5)
guile> (if (> a b) "a is greater than b" "a is not greater than b")
"a is not greater than b"
```

cond

Another conditional procedure in scheme is `cond`:

```
(cond (test-expression-1 result-expression-sequence-1)
      (test-expression-2 result-expression-sequence-2)
      ...
      (test-expression-n result-expression-sequence-n))
```

For example:

```
guile> (define a 6)
guile> (define b 8)
guile> (cond ((< a b) "a is less than b")
...          ((= a b) "a equals b")
...          ((> a b) "a is greater than b"))
"a is less than b"
```

1.2 Scheme in LilyPond

1.2.1 LilyPond Scheme syntax

The Guile interpreter is part of LilyPond, which means that Scheme can be included in LilyPond input files. There are several methods for including Scheme in LilyPond.

The simplest way is to use a hash mark `#` before a Scheme expression.

Now LilyPond's input is structured into tokens and expressions, much like human language is structured into words and sentences. LilyPond has a lexer that recognizes tokens (literal numbers, strings, Scheme elements, pitches and so on), and a parser that understands the syntax, Section "LilyPond grammar" in *Contributor's Guide*. Once it knows that a particular syntax rule applies, it executes actions associated with it.

The hash mark `#` method of embedding Scheme is a natural fit for this system. Once the lexer sees a hash mark, it calls the Scheme reader to read one full Scheme expression (this can be an identifier, an expression enclosed in parentheses, or several other things). After the Scheme expression is read, it is stored away as the value for an `SCM_TOKEN` in the grammar. Once the parser knows how to make use of this token, it calls Guile for evaluating the Scheme expression. Since the parser usually requires a bit of lookahead from the lexer to make its parsing decisions, this separation of reading and evaluation between lexer and parser is exactly what is needed to keep the execution of LilyPond and Scheme expressions in sync. For this reason, you should use the hash mark `#` for calling Scheme whenever this is feasible.

Another way to call the Scheme interpreter from LilyPond is the use of dollar `$` instead of a hash mark for introducing Scheme expressions. In this case, Lilypond evaluates the code right after the lexer has read it. It checks the resulting type of the Scheme expression and then picks a token type (one of several `xxx_IDENTIFIER` in the syntax) for it. It creates a *copy* of the value and uses that for the value of the token. If the value of the expression is void (Guile's value of `*unspecified*`), nothing at all is passed to the parser.

This is, in fact, exactly the same mechanism that Lilypond employs when you call any variable or music function by name, as `\name`, with the only difference that the name is determined by the Lilypond lexer without consulting the Scheme reader, and thus only variable names consistent with the current Lilypond mode are accepted.

The immediate action of `$` can lead to surprises, Section 1.2.3 [Input variables and Scheme], page 8. Using `#` where the parser supports it is usually preferable. Inside of music expressions, expressions created using `#` *are* interpreted as music. However, they are *not* copied before use. If they are part of some structure that might still get used, you may need to use `ly:music-deep-copy` explicitly.

There are also 'list splicing' operators `$@` and `#@` that insert all elements of a list in the surrounding context.

Now let's take a look at some actual Scheme code. Scheme procedures can be defined in LilyPond input files:

```
#(define (average a b c) (/ (+ a b c) 3))
```

Note that LilyPond comments (`%` and `%{ %}`) cannot be used within Scheme code, even in a LilyPond input file, because the Guile interpreter, not the LilyPond lexer, is reading the Scheme expression. Comments in Guile Scheme are entered as follows:

```
; this is a single-line comment

#!
  This a (non-nestable) Guile-style block comment
  But these are rarely used by Schemers and never in
  LilyPond source code
```

```
!#
```

For the rest of this section, we will assume that the data is entered in a music file, so we add #s at the beginning of each Scheme expression.

All of the top-level Scheme expressions in a LilyPond input file can be combined into a single Scheme expression by the use of the `begin` statement:

```
#(begin
  (define foo 0)
  (define bar 1))
```

1.2.2 LilyPond variables

LilyPond variables are stored internally in the form of Scheme variables. Thus,

```
twelve = 12
```

is equivalent to

```
#(define twelve 12)
```

This means that LilyPond variables are available for use in Scheme expressions. For example, we could use

```
twentyFour = #(* 2 twelve)
```

which would result in the number 24 being stored in the LilyPond (and Scheme) variable `twentyFour`.

The usual way to refer to Lilypond variables, Section 1.2.1 [LilyPond Scheme syntax], page 7, is to call them using a backslash, i.e., `\twentyFour`. Since this creates a copy of the value for most of LilyPond's internal types, in particular music expressions, music functions don't usually create copies of material they change. For this reason, music expressions given with # should usually not contain material that is not either created from scratch or explicitly copied rather than directly referenced.

1.2.3 Input variables and Scheme

The input format supports the notion of variables: in the following example, a music expression is assigned to a variable with the name `traLaLa`.

```
traLaLa = { c'4 d'4 }
```

There is also a form of scoping: in the following example, the `\layout` block also contains a `traLaLa` variable, which is independent of the outer `\traLaLa`.

```
traLaLa = { c'4 d'4 }
\layout { traLaLa = 1.0 }
```

In effect, each input file is a scope, and all `\header`, `\midi`, and `\layout` blocks are scopes nested inside that toplevel scope.

Both variables and scoping are implemented in the GUILE module system. An anonymous Scheme module is attached to each scope. An assignment of the form:

```
traLaLa = { c'4 d'4 }
```

is internally converted to a Scheme definition:

```
(define traLaLa Scheme value of `...')
```

This means that LilyPond variables and Scheme variables may be freely mixed. In the following example, a music fragment is stored in the variable `traLaLa`, and duplicated using Scheme. The result is imported in a `\score` block by means of a second variable `twice`:

```
traLaLa = { c'4 d'4 }

#(define newLa (map ly:music-deep-copy
```

```
  (list traLaLa traLaLa)))
#(define twice
  (make-sequential-music newLa))

\twice
```

This is actually a rather interesting example. The assignment will only take place after the parser has ascertained that nothing akin to `\addlyrics` follows, so it needs to check what comes next. It reads `#` and the following Scheme expression *without* evaluating it, so it can go ahead with the assignment, and *afterwards* execute the Scheme code without problem.

1.2.4 Importing Scheme in LilyPond

The above example shows how to 'export' music expressions from the input to the Scheme interpreter. The opposite is also possible. By placing it after `$`, a Scheme value is interpreted as if it were entered in LilyPond syntax. Instead of defining `\twice`, the example above could also have been written as

```
...
$(make-sequential-music newLa)
```

You can use `$` with a Scheme expression anywhere you could use `\name` after having assigned the Scheme expression to a variable *name*. This replacement happens in the 'lexer', so Lilypond is not even aware of the difference.

One drawback, however, is that of timing. If we had been using `$` instead of `#` for defining `newLa` in the above example, the following Scheme definition would have failed because `traLaLa` would not yet have been defined. For an explanation of this timing problem, Section 1.2.1 [LilyPond Scheme syntax], page 7.

A further convenience can be the 'list splicing' operators `$@` and `#@` for inserting the elements of a list in the surrounding context. Using those, the last part of the example could have been written as

```
...
{ #@newLa }
```

Here, every element of the list stored in `newLa` is taken in sequence and inserted into the list, as if we had written

```
{ #(first newLa) #(second newLa) }
```

Now in all of these forms, the Scheme code is evaluated while the input is still being consumed, either in the lexer or in the parser. If you need it to be executed at a later point of time, check out Section 2.2.3 [Void scheme functions], page 20, or store it in a procedure:

```
#(define (nopc)
  (ly:set-option 'point-and-click #f))

...
#(nopc)
{ c'4 }
```

Known issues and warnings

Mixing Scheme and LilyPond variables is not possible with the '`--safe`' option.

1.2.5 Object properties

Object properties are stored in LilyPond in the form of alist-chains, which are lists of alists. Properties are set by adding values at the beginning of the property list. Properties are read by retrieving values from the alists.

Setting a new value for a property requires assigning a value to the alist with both a key and a value. The LilyPond syntax for doing this is:

```
\override Stem.thickness = #2.6
```

This instruction adjusts the appearance of stems. An alist entry `'(thickness . 2.6)` is added to the property list of the `Stem` object. `thickness` is measured relative to the thickness of staff lines, so these stem lines will be `2.6` times the width of staff lines. This makes stems almost twice as thick as their normal size. To distinguish between variables defined in input files (like `twentyFour` in the example above) and variables of internal objects, we will call the latter 'properties' and the former 'variables.' So, the stem object has a `thickness` property, while `twentyFour` is a variable.

1.2.6 LilyPond compound variables

Offsets

Two-dimensional offsets (X and Y coordinates) are stored as *pairs*. The `car` of the offset is the X coordinate, and the `cdr` is the Y coordinate.

```
\override TextScript.extra-offset = #'(1 . 2)
```

This assigns the pair `(1 . 2)` to the `extra-offset` property of the TextScript object. These numbers are measured in staff-spaces, so this command moves the object 1 staff space to the right, and 2 spaces up.

Procedures for working with offsets are found in '`scm/lily-library.scm`'.

Fractions

Fractions as used by LilyPond are again stored as *pairs*, this time of unsigned integers. While Scheme can represent rational numbers as a native type, musically '`2/4`' and '`1/2`' are not the same, and we need to be able to distinguish between them. Similarly there are no negative 'fractions' in LilyPond's mind. So `2/4` in LilyPond means `(2 . 4)` in Scheme, and `#2/4` in LilyPond means `1/2` in Scheme.

Extents

Pairs are also used to store intervals, which represent a range of numbers from the minimum (the `car`) to the maximum (the `cdr`). Intervals are used to store the X- and Y- extents of printable objects. For X extents, the `car` is the left hand X coordinate, and the `cdr` is the right hand X coordinate. For Y extents, the `car` is the bottom coordinate, and the `cdr` is the top coordinate.

Procedures for working with intervals are found in '`scm/lily-library.scm`'. These procedures should be used when possible to ensure consistency of code.

Property alists

A property alist is a LilyPond data structure that is an alist whose keys are properties and whose values are Scheme expressions that give the desired value for the property.

LilyPond properties are Scheme symbols, such as `'thickness`.

Alist chains

An alist chain is a list containing property alists.

The set of all properties that will apply to a grob is typically stored as an alist chain. In order to find the value for a particular property that a grob should have, each alist in the chain is searched in order, looking for an entry containing the property key. The first alist entry found is returned, and the value is the property value.

The Scheme procedure `chain-assoc-get` is normally used to get grob property values.

1.2.7 Internal music representation

Internally, music is represented as a Scheme list. The list contains various elements that affect the printed output. Parsing is the process of converting music from the LilyPond input representation to the internal Scheme representation.

When a music expression is parsed, it is converted into a set of Scheme music objects. The defining property of a music object is that it takes up time. The time it takes up is called its *duration*. Durations are expressed as a rational number that measures the length of the music object in whole notes.

A music object has three kinds of types:

- music name: Each music expression has a name. For example, a note leads to a Section "NoteEvent" in *Internals Reference*, and `\simultaneous` leads to a Section "SimultaneousMusic" in *Internals Reference*. A list of all expressions available is in the Internals Reference manual, under Section "Music expressions" in *Internals Reference*.
- 'type' or interface: Each music name has several 'types' or interfaces, for example, a note is an `event`, but it is also a `note-event`, a `rhythmic-event`, and a `melodic-event`. All classes of music are listed in the Internals Reference, under Section "Music classes" in *Internals Reference*.
- C++ object: Each music object is represented by an object of the C++ class `Music`.

The actual information of a music expression is stored in properties. For example, a Section "NoteEvent" in *Internals Reference* has `pitch` and `duration` properties that store the pitch and duration of that note. A list of all properties available can be found in the Internals Reference, under Section "Music properties" in *Internals Reference*.

A compound music expression is a music object that contains other music objects in its properties. A list of objects can be stored in the `elements` property of a music object, or a single 'child' music object in the `element` property. For example, Section "SequentialMusic" in *Internals Reference* has its children in `elements`, and Section "GraceMusic" in *Internals Reference* has its single argument in `element`. The body of a repeat is stored in the `element` property of Section "RepeatedMusic" in *Internals Reference*, and the alternatives in `elements`.

1.3 Building complicated functions

This section explains how to gather the information necessary to create complicated music functions.

1.3.1 Displaying music expressions

When writing a music function it is often instructive to inspect how a music expression is stored internally. This can be done with the music function `\displayMusic`

```
{
  \displayMusic { c'4\f }
}
```

will display

```
(make-music
  'SequentialMusic
```

```
  'elements
  (list (make-music
          'NoteEvent
          'articulations
          (list (make-music
                  'AbsoluteDynamicEvent
                  'text
                  "f"))
          'duration
          (ly:make-duration 2 0 1/1)
          'pitch
          (ly:make-pitch 0 0 0))))
```

By default, LilyPond will print these messages to the console along with all the other messages. To split up these messages and save the results of `\display{STUFF}`, redirect the output to a file.

```
lilypond file.ly >display.txt
```

With a combined bit of Lilypond and Scheme magic, you can actually let Lilypond direct just this output to a file of its own:

```
{
  #(with-output-to-file "display.txt"
      (lambda () #{ \displayMusic { c'4\f } #}))
}
```

A bit of reformatting makes the above information easier to read:

```
(make-music 'SequentialMusic
  'elements (list
     (make-music 'NoteEvent
               'articulations (list
       (make-music 'AbsoluteDynamicEvent
 'text
 "f"))
       'duration (ly:make-duration 2 0 1/1)
       'pitch    (ly:make-pitch 0 0 0))))
```

A `{ ... }` music sequence has the name `SequentialMusic`, and its inner expressions are stored as a list in its `'elements` property. A note is represented as a `NoteEvent` object (storing the duration and pitch properties) with attached information (in this case, an `AbsoluteDynamicEvent` with a `"f"` text property) stored in its `articulations` property.

`\displayMusic` returns the music it displays, so it will get interpreted as well as displayed. To avoid interpretation, write `\void` before `\displayMusic`.

1.3.2 Music properties

TODO – make sure we delineate between *music* properties, *context* properties, and *layout* properties. These are potentially confusing.

Let's look at an example:

```
someNote = c'
\displayMusic \someNote
===>
(make-music
  'NoteEvent
  'duration
```

```
  (ly:make-duration 2 0 1/1)
  'pitch
  (ly:make-pitch 0 0 0))
```

The NoteEvent object is the representation of someNote. Straightforward. How about putting c' in a chord?

```
someNote = <c'>
\displayMusic \someNote
===>
(make-music
  'EventChord
  'elements
  (list (make-music
          'NoteEvent
          'duration
          (ly:make-duration 2 0 1/1)
          'pitch
          (ly:make-pitch 0 0 0))))
```

Now the NoteEvent object is the first object of the 'elements property of someNote.

The display-scheme-music function is the function used by \displayMusic to display the Scheme representation of a music expression.

```
#(display-scheme-music (first (ly:music-property someNote 'elements)))
===>
(make-music
  'NoteEvent
  'duration
  (ly:make-duration 2 0 1/1)
  'pitch
  (ly:make-pitch 0 0 0))
```

Then the note pitch is accessed through the 'pitch property of the NoteEvent object,

```
#(display-scheme-music
   (ly:music-property (first (ly:music-property someNote 'elements))
                      'pitch))
===>
(ly:make-pitch 0 0 0)
```

The note pitch can be changed by setting this 'pitch property,

```
#(set! (ly:music-property (first (ly:music-property someNote 'elements))
                          'pitch)
       (ly:make-pitch 0 1 0)) ;; set the pitch to d'.
\displayLilyMusic \someNote
===>
d'
```

1.3.3 Doubling a note with slurs (example)

Suppose we want to create a function that translates input like a into { a(a) }. We begin by examining the internal representation of the desired result.

```
\displayMusic{ a'( a') }
===>
(make-music
  'SequentialMusic
```

```
  'elements
  (list (make-music
          'NoteEvent
          'articulations
          (list (make-music
                  'SlurEvent
                  'span-direction
                  -1))
          'duration
          (ly:make-duration 2 0 1/1)
          'pitch
          (ly:make-pitch 0 5 0))
        (make-music
          'NoteEvent
          'articulations
          (list (make-music
                  'SlurEvent
                  'span-direction
                  1))
          'duration
          (ly:make-duration 2 0 1/1)
          'pitch
          (ly:make-pitch 0 5 0))))
```

The bad news is that the `SlurEvent` expressions must be added 'inside' the note (in its `articulations` property).

Now we examine the input,

```
\displayMusic a'
===>
(make-music
  'NoteEvent
  'duration
  (ly:make-duration 2 0 1/1)
  'pitch
  (ly:make-pitch 0 5 0))))
```

So in our function, we need to clone this expression (so that we have two notes to build the sequence), add a `SlurEvent` to the `'articulations` property of each one, and finally make a `SequentialMusic` with the two `NoteEvent` elements. For adding to a property, it is useful to know that an unset property is read out as `'()`, the empty list, so no special checks are required before we put another element at the front of the `articulations` property.

```
doubleSlur = #(define-music-function (parser location note) (ly:music?)
         "Return: { note ( note ) }.
         `note' is supposed to be a single note."
         (let ((note2 (ly:music-deep-copy note)))
           (set! (ly:music-property note 'articulations)
                 (cons (make-music 'SlurEvent 'span-direction -1)
                       (ly:music-property note 'articulations)))
           (set! (ly:music-property note2 'articulations)
                 (cons (make-music 'SlurEvent 'span-direction 1)
                       (ly:music-property note2 'articulations)))
           (make-music 'SequentialMusic 'elements (list note note2))))
```

1.3.4 Adding articulation to notes (example)

The easy way to add articulation to notes is to merge two music expressions into one context. However, suppose that we want to write a music function that does this. This will have the additional advantage that we can use that music function to add an articulation (like a fingering instruction) to a single note inside of a chord which is not possible if we just merge independent music.

A `$variable` inside the `#{...#}` notation is like a regular `\variable` in classical LilyPond notation. We know that

```
{ \music -. -> }
```

will not work in LilyPond. We could avoid this problem by attaching the articulation to an empty chord,

```
{ << \music <> -. -> >> }
```

but for the sake of this example, we will learn how to do this in Scheme. We begin by examining our input and desired output,

```
%  input
\displayMusic c4
===>
(make-music
  'NoteEvent
  'duration
  (ly:make-duration 2 0 1/1)
  'pitch
  (ly:make-pitch -1 0 0))))
=====
%  desired output
\displayMusic c4->
===>
(make-music
  'NoteEvent
  'articulations
  (list (make-music
          'ArticulationEvent
          'articulation-type
          "accent"))
  'duration
  (ly:make-duration 2 0 1/1)
  'pitch
  (ly:make-pitch -1 0 0))
```

We see that a note (`c4`) is represented as an `NoteEvent` expression. To add an accent articulation, an `ArticulationEvent` expression must be added to the `articulations` property of the `NoteEvent` expression.

To build this function, we begin with

```
(define (add-accent note-event)
  "Add an accent ArticulationEvent to the articulations of `note-event',
  which is supposed to be a NoteEvent expression."
  (set! (ly:music-property note-event 'articulations)
        (cons (make-music 'ArticulationEvent
                'articulation-type "accent")
              (ly:music-property note-event 'articulations)))
```

```
  note-event)
```

The first line is the way to define a function in Scheme: the function name is `add-accent`, and has one variable called `note-event`. In Scheme, the type of variable is often clear from its name. (this is good practice in other programming languages, too!)

```
"Add an accent..."
```

is a description of what the function does. This is not strictly necessary, but just like clear variable names, it is good practice.

You may wonder why we modify the note event directly instead of working on a copy (`ly:music-deep-copy` can be used for that). The reason is a silent contract: music functions are allowed to modify their arguments: they are either generated from scratch (like user input) or are already copied (referencing a music variable with '`\name`' or music from immediate Scheme expressions '`$(...)`' provides a copy). Since it would be inefficient to create unnecessary copies, the return value from a music function is *not* copied. So to heed that contract, you must not use any arguments more than once, and returning it counts as one use.

In an earlier example, we constructed music by repeating a given music argument. In that case, at least one repetition had to be a copy of its own. If it weren't, strange things may happen. For example, if you use `\relative` or `\transpose` on the resulting music containing the same elements multiple times, those will be subjected to relativation or transposition multiple times. If you assign them to a music variable, the curse is broken since referencing '`\name`' will again create a copy which does not retain the identity of the repeated elements.

Now while the above function is not a music function, it will normally be used within music functions. So it makes sense to heed the same contract we use for music functions: the input may be modified for producing the output, and the caller is responsible for creating copies if it still needs the unchanged argument itself. If you take a look at LilyPond's own functions like `music-map`, you'll find that they stick with the same principles.

Where were we? Now we have a `note-event` we may modify, not because of using `ly:music-deep-copy` but because of a long-winded explanation. We add the accent to its `'articulations` list property.

```
(set! place new-value)
```

Here, what we want to set (the 'place') is the `'articulations` property of `note-event` expression.

```
(ly:music-property note-event 'articulations)
```

`ly:music-property` is the function used to access music properties (the `'articulations`, `'duration`, `'pitch`, etc, that we see in the `\displayMusic` output above). The new value is the former `'articulations` property, with an extra item: the `ArticulationEvent` expression, which we copy from the `\displayMusic` output,

```
(cons (make-music 'ArticulationEvent
        'articulation-type "accent")
      (ly:music-property result-event-chord 'articulations))
```

`cons` is used to add an element to the front of a list without modifying the original list. This is what we want: the same list as before, plus the new `ArticulationEvent` expression. The order inside the `'articulations` property is not important here.

Finally, once we have added the accent articulation to its `articulations` property, we can return `note-event`, hence the last line of the function.

Now we transform the `add-accent` function into a music function (a matter of some syntactic sugar and a declaration of the type of its sole 'real' argument).

```
addAccent = #(define-music-function (parser location note-event)
                                     (ly:music?)
```

```
  "Add an accent ArticulationEvent to the articulations of `note-event',
  which is supposed to be a NoteEvent expression."
  (set! (ly:music-property note-event 'articulations)
        (cons (make-music 'ArticulationEvent
                'articulation-type "accent")
              (ly:music-property note-event 'articulations)))
  note-event)
```

We may verify that this music function works correctly,

```
\displayMusic \addAccent c4
```

2 Interfaces for programmers

Advanced tweaks may be performed by using Scheme. If you are not familiar with Scheme, you may wish to read our Chapter 1 [Scheme tutorial], page 1.

2.1 LilyPond code blocks

Creating music expressions in Scheme can be tedious, as they are heavily nested and the resulting Scheme code is large. For some simple tasks this can be avoided by using LilyPond code blocks, which enable common LilyPond syntax to be used within Scheme.

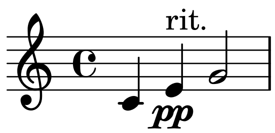
LilyPond code blocks look like

```
#{ LilyPond code #}
```

Here is a trivial example:

```
ritpp = #(define-event-function (parser location) ()
  #{ ^"rit." \pp #}
)

{ c'4 e'4\ritpp g'2 }
```



LilyPond code blocks can be used anywhere where you can write Scheme code. The Scheme reader actually is changed for accommodating LilyPond code blocks and can deal with embedded Scheme expressions starting with `$` and `#`.

The reader extracts the LilyPond code block and generates a runtime call to the LilyPond `parser` to interpret the LilyPond code. Scheme expressions embedded in the LilyPond code are evaluated in the lexical environment of the LilyPond code block, so all local variables and function parameters available at the point the LilyPond code block is written may be accessed. Variables defined in other Scheme modules, like the modules containing `\header` and `\layout` blocks, are not accessible as Scheme variables, i.e. prefixed with `#`, but they are accessible as LilyPond variables, i.e. prefixed with `\`.

If `location` (see Section 2.2 [Scheme functions], page 18) refers to a valid input location (which it usually does inside of music/Scheme functions), all music generated inside the code block has its '`origin`' set to `location`.

A LilyPond code block may contain anything that you can use on the right side of an assignment. In addition, an empty LilyPond block corresponds to a void music expression, and a LilyPond block containing multiple music events gets turned into a sequential music expression.

2.2 Scheme functions

Scheme functions are Scheme procedures that can create Scheme expressions from input written in LilyPond syntax. They can be called in pretty much all places where using `#` for specifying a value in Scheme syntax is allowed. While Scheme has functions of its own, this chapter is concerned with *syntactic* functions, functions that receive arguments specified in LilyPond syntax.

2.2.1 Scheme function definitions

The general form for defining scheme functions is:

```
function =
#(define-scheme-function
     (parser location arg1 arg2 ...)
     (type1? type2? ...)
   body)
```

where

`parser` needs to be literally `parser` in order to give LilyPond code blocks (`#{...#}`) access to the parser.

`location` needs to be literally `location` in order to provide access to the input location object, which is used to provide error messages with file names and line numbers.

`argN` *n*th argument

`typeN?` a Scheme *type predicate* for which *`argN`* must return `#t`. There is also a special form `(`*`predicate? default`*`)` for specifying optional arguments. If the actual argument is missing when the function is being called, the default value is substituted instead. Default values are evaluated at definition time (including LilyPond code blocks!), so if you need a default calculated at runtime, instead write a special value you can easily recognize. If you write the predicate in parentheses but don't follow it with a default value, `#f` is used as the default. Default values are not verified with *predicate?* at either definition or run time: it is your responsibility to deal with the values you specify. Default values that happen to be music expressions are copied while setting `origin` to the `location` parameter.

`body` A sequence of Scheme forms evaluated in order, the last one being used as the return value of the scheme function. It may contain LilyPond code blocks enclosed in hashed braces (`#{...#}`), like described in Section 2.1 [LilyPond code blocks], page 18. Within LilyPond code blocks, use `#` to reference function arguments (eg., '`#arg1`') or to start an inline Scheme expression containing function arguments (eg., '`#(cons arg1 arg2)`'). Where normal Scheme expressions using `#` don't do the trick, you might need to revert to immediate Scheme expressions using `$`, for example as '`$music`'.
If your function returns a music expression, it is given a useful value of `origin`.

Suitability of arguments for the predicates is determined by actually calling the predicate after LilyPond has already converted them into a Scheme expression. As a consequence, the argument can be specified in Scheme syntax if desired (introduced with `#` or as the result of calling a scheme function), but LilyPond will also convert a number of LilyPond constructs into Scheme before actually checking the predicate on them. Currently, those include music, postevents, simple strings (with or without quotes), numbers, full markups and markup lists, score, book, bookpart, context definition and output definition blocks.

For some kinds of expression (like most music not enclosed in braces) LilyPond needs to look further than the expression itself in order to determine its end. If such an expression were considered for an optional argument by evaluating its predicate, LilyPond would not be able to 'backup' when it decides the expression does not fit the parameter. So some forms of music might need to be enclosed in braces to make them acceptable in some circumstances. Some other ambiguities LilyPond sorts out by checking with predicate functions: is '`-3`' a fingering postevent or a negative number? Is `"a" 4` in lyric mode a string followed by a number, or a lyric event of duration `4`? LilyPond tries the argument predicate on successive interpretations until success, with an order designed to minimize inconsistent interpretations and lookahead.

For example, a predicate accepting both music expressions and pitches will consider `c''` to be a pitch rather than a music expression. Immediately following durations or postevents might not work with that interpretation. So it's best to avoid overly permissive predicates like `scheme?` when the application rather calls for more specific argument types.

For a list of available predefined type predicates, see Section "Predefined type predicates" in *Notation Reference*.

See also

Notation Reference: Section "Predefined type predicates" in *Notation Reference*.

Installed Files: '`lily/music-scheme.cc`', '`scm/c++.scm`', '`scm/lily.scm`'.

2.2.2 Scheme function usage

Scheme functions can be called pretty much anywhere where a Scheme expression starting with `#` can be written. You call a scheme function by writing its name preceded by `\`, followed by its arguments. Once an optional argument predicate does not match an argument, LilyPond skips this and all following optional arguments, replacing them with their specified default, and 'backs up' the argument that did not match to the place of the next mandatory argument. Since the backed up argument needs to go somewhere, optional arguments are not actually considered optional unless followed by a mandatory argument.

There is one exception: if you write `\default` in the place of an optional argument, this and all following optional arguments are skipped and replaced by their default. This works even when no mandatory argument follows since `\default` does not need to get backed up. The `mark` and `key` commands make use of that trick to provide their default behavior when just followed by `\default`.

Apart from places where a Scheme value is required, there are a few places where `#` expressions are currently accepted and evaluated for their side effects but otherwise ignored. Mostly those are the places where an assignment would be acceptable as well.

Since it is a bad idea to return values that can be misinterpreted in some context, you should use normal scheme functions only for those cases where you always return a useful value, and use void scheme functions (see Section 2.2.3 [Void scheme functions], page 20) otherwise.

2.2.3 Void scheme functions

Sometimes a procedure is executed in order to perform an action rather than return a value. Some programming languages (like C and Scheme) use functions for either concept and just discard the returned value (usually by allowing any expression to act as statement, ignoring the result). This is clever but error-prone: most C compilers nowadays offer warnings for various non-"void" expressions being discarded. For many functions executing an action, the Scheme standards declare the return value to be unspecified. LilyPond's Scheme interpreter Guile has a unique value `*unspecified*` that it usually (such when using `set!` directly on a variable) but unfortunately not consistently returns in such cases.

Defining a LilyPond function with `define-void-function` makes sure that this special value (the only value satisfying the predicate `void?`) will be returned.

```
noPointAndClick =
#(define-void-function
     (parser location)
     ()
   (ly:set-option 'point-and-click #f))
...
\noPointAndClick   % disable point and click
```

If you want to evaluate an expression only for its side-effect and don't want any value it may return interpreted, you can do so by prefixing it with `\void`:

```
\void #(hashq-set! some-table some-key some-value)
```

That way, you can be sure that LilyPond will not assign meaning to the returned value regardless of where it encounters it. This will also work for music functions such as `\displayMusic`.

2.3 Music functions

Music functions are Scheme procedures that can create music expressions automatically, and can be used to greatly simplify the input file.

2.3.1 Music function definitions

The general form for defining music functions is:

```
function =
#(define-music-function
     (parser location arg1 arg2 ...)
     (type1? type2? ...)
   body)
```

quite in analogy to Section 2.2.1 [Scheme function definitions], page 19. More often than not, *body* will be a Section 2.1 [LilyPond code blocks], page 18.

For a list of available type predicates, see Section "Predefined type predicates" in *Notation Reference*.

See also

Notation Reference: Section "Predefined type predicates" in *Notation Reference*.

Installed Files: 'lily/music-scheme.cc', 'scm/c++.scm', 'scm/lily.scm'.

2.3.2 Music function usage

Music functions may currently be used in several places. Depending on where they are used, restrictions apply in order to be able to parse them unambiguously. The result a music function returns must be compatible with the context in which it is called.

- At top level in a music expression. No restriction apply here.
- As a post-event, explicitly started with a direction indicator (one of `-`, `^`, and `_`).

  In this case, you can't use an *open* music expression as the last argument, one that would end with a music expression able to accept additional postevents.
- As a chord constituent. The returned expression must be of `rhythmic-event` type, most likely a `NoteEvent`.

The special rules for trailing arguments make it possible to write polymorphic functions like `\tweak` that can be applied to different constructs.

2.3.3 Simple substitution functions

Simple substitution functions are music functions whose output music expression is written in LilyPond format and contains function arguments in the output expression. They are described in Section "Substitution function examples" in *Notation Reference*.

2.3.4 Intermediate substitution functions

Intermediate substitution functions involve a mix of Scheme code and LilyPond code in the music expression to be returned.

Some `\override` commands require an argument consisting of a pair of numbers (called a *cons cell* in Scheme).

The pair can be directly passed into the music function, using a `pair?` variable:

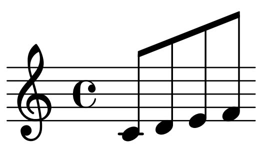
```
manualBeam =
#(define-music-function
     (parser location beg-end)
     (pair?)
   #{
     \once \override Beam.positions = #beg-end
   #})

\relative c' {
  \manualBeam #'(3 . 6) c8 d e f
}
```

Alternatively, the numbers making up the pair can be passed as separate arguments, and the Scheme code used to create the pair can be included in the music expression:

```
manualBeam =
#(define-music-function
     (parser location beg end)
     (number? number?)
   #{
     \once \override Beam.positions = #(cons beg end)
   #})

\relative c' {
  \manualBeam #3 #6 c8 d e f
}
```



Properties are maintained conceptually using one stack per property per grob per context. Music functions may need to override one or several properties for the duration of the function, restoring them to their previous value before exiting. However, normal overrides pop and discard the top of the current property stack before pushing to it, so the previous value of the property is lost when it is overridden. When the previous value must be preserved, prefix the `\override` command with `\temporary`, like this:

```
\temporary \override ...
```

The use of `\temporary` causes the (usually set) `pop-first` property in the override to be cleared, so the previous value is not popped off the property stack before pushing the new value onto it. When a subsequent `\revert` pops off the temporarily overriden value, the previous value will re-emerge.

In other words, calling `\temporary \override` and `\revert` in succession on the same property will have a net effect of zero. Similarly, pairing `\temporary` and `\undo` on the same music containing overrides will have a net effect of zero.

Here is an example of a music function which makes use of this. The use of `\temporary` ensures the values of the `cross-staff` and `style` properties are restored on exit to whatever values they had when the `crossStaff` function was called. Without `\temporary` the default values would have been set on exit.

```
crossStaff =
#(define-music-function (parser location notes) (ly:music?)
  (_i "Create cross-staff stems")
  #{
  \temporary \override Stem.cross-staff = #cross-staff-connect
  \temporary \override Flag.style = #'no-flag
  #notes
  \revert Stem.cross-staff
  \revert Flag.style
#})
```

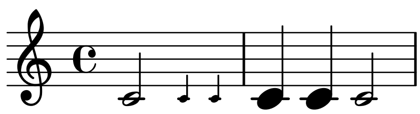
2.3.5 Mathematics in functions

Music functions can involve Scheme programming in addition to simple substitution,

```
AltOn =
#(define-music-function
     (parser location mag)
     (number?)
   #{
     \override Stem.length = #(* 7.0 mag)
     \override NoteHead.font-size =
       #(inexact->exact (* (/ 6.0 (log 2.0)) (log mag)))
   #})

AltOff = {
  \revert Stem.length
  \revert NoteHead.font-size
}

\relative c' {
  c2 \AltOn #0.5 c4 c
  \AltOn #1.5 c c \AltOff c2
}
```

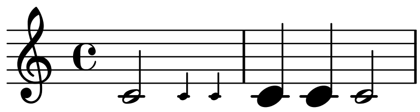


This example may be rewritten to pass in music expressions,

```
withAlt =
#(define-music-function
     (parser location mag music)
     (number? ly:music?)
```

```
   #{
     \override Stem.length = #(* 7.0 mag)
     \override NoteHead.font-size =
       #(inexact->exact (* (/ 6.0 (log 2.0)) (log mag)))
     #music
     \revert Stem.length
     \revert NoteHead.font-size
   #})

\relative c' {
  c2 \withAlt #0.5 { c4 c }
  \withAlt #1.5 { c c } c2
}
```



2.3.6 Functions without arguments

In most cases a function without arguments should be written with a variable,

```
dolce = \markup{ \italic \bold dolce }
```

However, in rare cases it may be useful to create a music function without arguments,

```
displayBarNum =
#(define-music-function
     (parser location)
     ()
   (if (eq? #t (ly:get-option 'display-bar-numbers))
       #{ \once \override Score.BarNumber.break-visibility = ##f #}
       #{#}))
```

To actually display bar numbers where this function is called, invoke `lilypond` with

```
lilypond -d display-bar-numbers FILENAME.ly
```

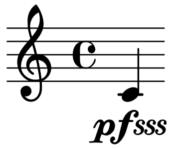
2.3.7 Void music functions

A music function must return a music expression. If you want to execute a function only for its side effect, you should use `define-void-function`. But there may be cases where you sometimes want to produce a music expression, and sometimes not (like in the previous example). Returning a `void` music expression via `#{ #}` will achieve that.

2.4 Event functions

To use a music function in the place of an event, you need to write a direction indicator before it. But sometimes, this does not quite match the syntax of constructs you want to replace. For example, if you want to write dynamics commands, those are usually attached without direction indicator, like `c'\pp`. Here is a way to write arbitrary dynamics:

```
dyn=#(define-event-function (parser location arg) (markup?)
         (make-dynamic-script arg))
\relative c' { c\dyn pfsss }
```



You could do the same using a music function, but then you always would have to write a direction indicator before calling it, like `c-\dyn pfsss`.

2.5 Markup functions

Markups are implemented as special Scheme functions which produce a `Stencil` object given a number of arguments.

2.5.1 Markup construction in Scheme

Markup expressions are internally represented in Scheme using the `markup` macro:

```
(markup expr)
```

To see a markup expression in its Scheme form, use the `\displayScheme` command:

```
\displayScheme
\markup {
  \column {
    \line { \bold \italic "hello" \raise #0.4 "world" }
    \larger \line { foo bar baz }
  }
}
```

Compiling the code above will send the following to the display console:

```
(markup
  #:line
  (#:column
   (#:line
    (#:bold (#:italic "hello") #:raise 0.4 "world")
    #:larger
    (#:line
     (#:simple "foo" #:simple "bar" #:simple "baz")))))
```

To prevent the markup from printing on the page, use '`\void \displayScheme` *markup*'. Also, as with the `\displayMusic` command, the output of `\displayScheme` can be saved to an external file. See Section 1.3.1 [Displaying music expressions], page 11.

This example demonstrates the main translation rules between regular LilyPond markup syntax and Scheme markup syntax. Using `#{ ... #}` for entering in LilyPond syntax will often be most convenient, but we explain how to use the `markup` macro to get a Scheme-only solution.

LilyPond	**Scheme**
`\markup markup1`	`(markup markup1)`
`\markup { markup1 markup2 ... }`	`(markup markup1 markup2 ... )`
`\markup-command`	`#:markup-command`
`\variable`	`variable`
`\center-column { ... }`	`#:center-column ( ... )`
`string`	`"string"`
`#scheme-arg`	`scheme-arg`

The whole Scheme language is accessible inside the `markup` macro. For example, You may use function calls inside `markup` in order to manipulate character strings. This is useful when defining new markup commands (see Section 2.5.3 [New markup command definition], page 26).

Known issues and warnings

The markup-list argument of commands such as `#:line`, `#:center`, and `#:column` cannot be a variable or the result of a function call.

```
(markup #:line (function-that-returns-markups))
```

is invalid. One should use the `make-line-markup`, `make-center-markup`, or `make-column-markup` functions instead,

```
(markup (make-line-markup (function-that-returns-markups)))
```

2.5.2 How markups work internally

In a markup like

```
\raise #0.5 "text example"
```

`\raise` is actually represented by the `raise-markup` function. The markup expression is stored as

```
(list raise-markup 0.5 (list simple-markup "text example"))
```

When the markup is converted to printable objects (Stencils), the `raise-markup` function is called as

```
(apply raise-markup
       \layout object
       list of property alists
       0.5
       the "text example" markup)
```

The `raise-markup` function first creates the stencil for the `text example` string, and then it raises that Stencil by 0.5 staff space. This is a rather simple example; more complex examples are in the rest of this section, and in '`scm/define-markup-commands.scm`'.

2.5.3 New markup command definition

This section discusses the definition of new markup commands.

Markup command definition syntax

New markup commands can be defined using the `define-markup-command` Scheme macro, at top-level.

```
(define-markup-command (command-name layout props arg1 arg2 ...)
    (arg1-type? arg2-type? ...)
    [ #:properties ((property1 default-value1)
                    ...) ]
  ...command body...)
```

The arguments are

`command-name`
: the markup command name

`layout`
: the 'layout' definition.

`props`
: a list of associative lists, containing all active properties.

`argi`
: *i*th command argument

`argi-type?`
: a type predicate for the i*th* argument

If the command uses properties from the `props` arguments, the `#:properties` keyword can be used to specify which properties are used along with their default values.

Arguments are distinguished according to their type:

- a markup, corresponding to type predicate `markup?`;
- a list of markups, corresponding to type predicate `markup-list?`;
- any other scheme object, corresponding to type predicates such as `list?`, `number?`, `boolean?`, etc.

There is no limitation on the order of arguments (after the standard `layout` and `props` arguments). However, markup functions taking a markup as their last argument are somewhat special as you can apply them to a markup list, and the result is a markup list where the markup function (with the specified leading arguments) has been applied to every element of the original markup list.

Since replicating the leading arguments for applying a markup function to a markup list is cheap mostly for Scheme arguments, you avoid performance pitfalls by just using Scheme arguments for the leading arguments of markup functions that take a markup as their last argument.

Markup commands have a rather complex life cycle. The body of a markup command definition is responsible for converting the arguments of the markup command into a stencil expression which is returned. Quite often this is accomplished by calling the `interpret-markup` function on a markup expression, passing the *layout* and *props* arguments on to it. Those arguments are usually only known at a very late stage in typesetting. Markup expressions have their components assembled into markup expressions already when `\markup` in a LilyPond expression or the `markup` macro in Scheme is expanded. The evaluation and typechecking of markup command arguments happens at the time `\markup`/`markup` are interpreted.

But the actual conversion of markup expressions into stencil expressions by executing the markup function bodies only happens when `interpret-markup` is called on a markup expression.

On properties

The `layout` and `props` arguments of markup commands bring a context for the markup interpretation: font size, line width, etc.

The `layout` argument allows access to properties defined in `paper` blocks, using the `ly:output-def-lookup` function. For instance, the line width (the same as the one used in scores) is read using:

```
(ly:output-def-lookup layout 'line-width)
```

The `props` argument makes some properties accessible to markup commands. For instance, when a book title markup is interpreted, all the variables defined in the `\header` block are automatically added to `props`, so that the book title markup can access the book title, composer, etc. It is also a way to configure the behaviour of a markup command: for example, when a command uses font size during processing, the font size is read from `props` rather than having a `font-size` argument. The caller of a markup command may change the value of the font size property in order to change the behaviour. Use the `#:properties` keyword of `define-markup-command` to specify which properties shall be read from the `props` arguments.

The example in next section illustrates how to access and override properties in a markup command.

A complete example

The following example defines a markup command to draw a double box around a piece of text.

Firstly, we need to build an approximative result using markups. Consulting the Section "Text markup commands" in *Notation Reference* shows us the `\box` command is useful:

```
\markup \box \box HELLO
```

Now, we consider that more padding between the text and the boxes is preferable. According to the `\box` documentation, this command uses a `box-padding` property, which defaults to 0.2. The documentation also mentions how to override it:

```
\markup \box \override #'(box-padding . 0.6) \box A
```

Then, the padding between the two boxes is considered too small, so we override it too:

```
\markup \override #'(box-padding . 0.4) \box
     \override #'(box-padding . 0.6) \box A
```

Repeating this lengthy markup would be painful. This is where a markup command is needed. Thus, we write a `double-box` markup command, taking one argument (the text). This draws the two boxes, with some padding.

```
#(define-markup-command (double-box layout props text) (markup?)
  "Draw a double box around text."
  (interpret-markup layout props
    #{\markup \override #'(box-padding . 0.4) \box
            \override #'(box-padding . 0.6) \box { #text }#}))
```

or, equivalently

```
#(define-markup-command (double-box layout props text) (markup?)
  "Draw a double box around text."
  (interpret-markup layout props
    (markup #:override '(box-padding . 0.4) #:box
            #:override '(box-padding . 0.6) #:box text)))
```

`text` is the name of the command argument, and `markup?` its type: it identifies it as a markup. The `interpret-markup` function is used in most of markup commands: it builds a stencil, using `layout`, `props`, and a markup. In the second case, this markup is built using the `markup` scheme macro, see Section 2.5.1 [Markup construction in Scheme], page 25. The transformation from `\markup` expression to scheme markup expression is straight-forward.

The new command can be used as follow:

```
\markup \double-box A
```

It would be nice to make the `double-box` command customizable: here, the `box-padding` values are hard coded, and cannot be changed by the user. Also, it would be better to distinguish the padding between the two boxes, from the padding between the inner box and the text. So we will introduce a new property, `inter-box-padding`, for the padding between the two boxes. The `box-padding` will be used for the inner padding. The new code is now as follows:

```
#(define-markup-command (double-box layout props text) (markup?)
  #:properties ((inter-box-padding 0.4)
                (box-padding 0.6))
  "Draw a double box around text."
  (interpret-markup layout props
    #{\markup \override #`(box-padding . ,inter-box-padding) \box
               \override #`(box-padding . ,box-padding) \box
               { #text } #}))
```

Again, the equivalent version using the markup macro would be:

```
#(define-markup-command (double-box layout props text) (markup?)
  #:properties ((inter-box-padding 0.4)
                (box-padding 0.6))
  "Draw a double box around text."
  (interpret-markup layout props
    (markup #:override `(box-padding . ,inter-box-padding) #:box
            #:override `(box-padding . ,box-padding) #:box text)))
```

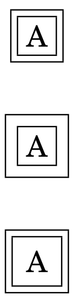
Here, the `#:properties` keyword is used so that the `inter-box-padding` and `box-padding` properties are read from the `props` argument, and default values are given to them if the properties are not defined.

Then, these values are used to override the `box-padding` properties used by the two `\box` commands. Note the backquote and the comma in the `\override` argument: they allow you to introduce a variable value into a literal expression.

Now, the command can be used in a markup, and the boxes padding be customized:

```
#(define-markup-command (double-box layout props text) (markup?)
  #:properties ((inter-box-padding 0.4)
                (box-padding 0.6))
  "Draw a double box around text."
  (interpret-markup layout props
    #{\markup \override #`(box-padding . ,inter-box-padding) \box
              \override #`(box-padding . ,box-padding) \box
              { #text } #}))

\markup \double-box A
\markup \override #'(inter-box-padding . 0.8) \double-box A
\markup \override #'(box-padding . 1.0) \double-box A
```



Adapting builtin commands

A good way to start writing a new markup command, is to take example on a builtin one. Most of the markup commands provided with LilyPond can be found in file ‘`scm/define-markup-commands.scm`’.

For instance, we would like to adapt the `\draw-line` command, to draw a double line instead. The `\draw-line` command is defined as follow (documentation stripped):

```
(define-markup-command (draw-line layout props dest)
  (number-pair?)
  #:category graphic
  #:properties ((thickness 1))
  "...documentation..."
  (let ((th (* (ly:output-def-lookup layout 'line-thickness)
               thickness))
        (x (car dest))
```

```
        (y (cdr dest)))
    (make-line-stencil th 0 0 x y)))
```

To define a new command based on an existing one, copy the definition, and change the command name. The `#:category` keyword can be safely removed, as it is only used for generating LilyPond documentation, and is of no use for user-defined markup commands.

```
(define-markup-command (draw-double-line layout props dest)
  (number-pair?)
  #:properties ((thickness 1))
  "...documentation..."
  (let ((th (* (ly:output-def-lookup layout 'line-thickness)
               thickness))
        (x (car dest))
        (y (cdr dest)))
    (make-line-stencil th 0 0 x y)))
```

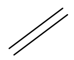
Then, a property for setting the gap between two lines is added, called `line-gap`, defaulting e.g. to 0.6:

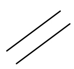
```
(define-markup-command (draw-double-line layout props dest)
  (number-pair?)
  #:properties ((thickness 1)
                (line-gap 0.6))
  "...documentation..."
  ...
```

Finally, the code for drawing two lines is added. Two calls to `make-line-stencil` are used to draw the lines, and the resulting stencils are combined using `ly:stencil-add`:

```
#(define-markup-command (my-draw-line layout props dest)
  (number-pair?)
  #:properties ((thickness 1)
                (line-gap 0.6))
  "..documentation.."
  (let* ((th (* (ly:output-def-lookup layout 'line-thickness)
                thickness))
         (dx (car dest))
         (dy (cdr dest))
         (w (/ line-gap 2.0))
         (x (cond ((= dx 0) w)
                  ((= dy 0) 0)
                  (else (/ w (sqrt (+ 1 (* (/ dx dy) (/ dx dy))))))))
         (y (* (if (< (* dx dy) 0) 1 -1)
               (cond ((= dy 0) w)
                     ((= dx 0) 0)
                     (else (/ w (sqrt (+ 1 (* (/ dy dx) (/ dy dx))))))))))
     (ly:stencil-add (make-line-stencil th x y (+ dx x) (+ dy y))
                     (make-line-stencil th (- x) (- y) (- dx x) (- dy y)))))

\markup \my-draw-line #'(4 . 3)
\markup \override #'(line-gap . 1.2) \my-draw-line #'(4 . 3)
```





2.5.4 New markup list command definition

Markup list commands are defined with the `define-markup-list-command` Scheme macro, which is similar to the `define-markup-command` macro described in Section 2.5.3 [New markup command definition], page 26, except that where the latter returns a single stencil, the former returns a list of stencils.

In a similar vein, `interpret-markup-list` is used instead of `interpret-markup` for converting a markup list into a list of stencils.

In the following example, a `\paragraph` markup list command is defined, which returns a list of justified lines, the first one being indented. The indent width is taken from the `props` argument.

```
#(define-markup-list-command (paragraph layout props args) (markup-list?)
   #:properties ((par-indent 2))
   (interpret-markup-list layout props
     #{\markuplist \justified-lines { \hspace #par-indent #args } #}))
```

The version using just Scheme is more complex:

```
#(define-markup-list-command (paragraph layout props args) (markup-list?)
   #:properties ((par-indent 2))
   (interpret-markup-list layout props
     (make-justified-lines-markup-list (cons (make-hspace-markup par-indent)
                                             args))))
```

Besides the usual `layout` and `props` arguments, the `paragraph` markup list command takes a markup list argument, named `args`. The predicate for markup lists is `markup-list?`.

First, the function gets the indent width, a property here named `par-indent`, from the property list `props`. If the property is not found, the default value is `2`. Then, a list of justified lines is made using the built-in markup list command `\justified-lines`, which is related to the `make-justified-lines-markup-list` function. A horizontal space is added at the beginning using `\hspace` (or the `make-hspace-markup` function). Finally, the markup list is interpreted using the `interpret-markup-list` function.

This new markup list command can be used as follows:

```
\markuplist {
  \paragraph {
    The art of music typography is called \italic {(plate) engraving.}
    The term derives from the traditional process of music printing.
    Just a few decades ago, sheet music was made by cutting and stamping
    the music into a zinc or pewter plate in mirror image.
  }
  \override-lines #'(par-indent . 4) \paragraph {
    The plate would be inked, the depressions caused by the cutting
    and stamping would hold ink.  An image was formed by pressing paper
    to the plate.  The stamping and cutting was completely done by
    hand.
  }
}
```

2.6 Contexts for programmers

2.6.1 Context evaluation

Contexts can be modified during interpretation with Scheme code. The syntax for this is

`\applyContext` *function*

function should be a Scheme function that takes a single argument: the context in which the `\applyContext` command is being called. The following code will print the current bar number on the standard output during the compile:

```
\applyContext
  #(lambda (x)
    (format #t "\nWe were called in barnumber ~a.\n"
     (ly:context-property x 'currentBarNumber)))
```

2.6.2 Running a function on all layout objects

The most versatile way of tuning an object is `\applyOutput` which works by inserting an event into the specified context (Section "ApplyOutputEvent" in *Internals Reference*). Its syntax is

```
\applyOutput Context proc
```

where *proc* is a Scheme function, taking three arguments.

When interpreted, the function *proc* is called for every layout object found in the context *Context* at the current time step, with the following arguments:

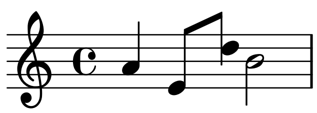
- the layout object itself,
- the context where the layout object was created, and
- the context where `\applyOutput` is processed.

In addition, the cause of the layout object, i.e., the music expression or object that was responsible for creating it, is in the object property `cause`. For example, for a note head, this is a Section "NoteHead" in *Internals Reference* event, and for a stem object, this is a Section "Stem" in *Internals Reference* object.

Here is a function to use for `\applyOutput`; it blanks note-heads on the center-line and next to it:

```
#(define (blanker grob grob-origin context)
   (if (and (memq 'note-head-interface (ly:grob-interfaces grob))
            (< (abs (ly:grob-property grob 'staff-position)) 2))
       (set! (ly:grob-property grob 'transparent) #t)))

\relative c' {
  a'4 e8 <<\applyOutput #'Voice #blanker a c d>> b2
}
```



To have *function* interpreted at the `Score` or `Staff` level use these forms

```
\applyOutput #'Score #function
\applyOutput #'Staff #function
```

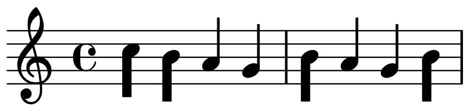
2.7 Callback functions

Properties (like `thickness`, `direction`, etc.) can be set at fixed values with `\override`, e.g.

```
\override Stem.thickness = #2.0
```

Properties can also be set to a Scheme procedure,

```
\override Stem.thickness = #(lambda (grob)
    (if (= UP (ly:grob-property grob 'direction))
        2.0
        7.0))
c b a g b a g b
```



In this case, the procedure is executed as soon as the value of the property is requested during the formatting process.

Most of the typesetting engine is driven by such callbacks. Properties that typically use callbacks include

`stencil` The printing routine, that constructs a drawing for the symbol

`X-offset` The routine that sets the horizontal position

`X-extent` The routine that computes the width of an object

The procedure always takes a single argument, being the grob.

That procedure may access the usual value of the property, by first calling the function that is the usual callback for that property, which can by found in the Internals Reference or the file 'define-grobs.scm':

```
\relative c'' {
  \override Flag #'X-offset = #(lambda (flag)
    (let ((default (ly:flag::calc-x-offset flag)))
      (* default 4.0)))
  c4. d8 a4. g8
}
```

If routines with multiple arguments must be called, the current grob can be inserted with a grob closure. Here is a setting from `AccidentalSuggestion`,

```
`(X-offset .
  ,(ly:make-simple-closure
    `(,+
        ,(ly:make-simple-closure
           (list ly:self-alignment-interface::centered-on-x-parent))
      ,(ly:make-simple-closure
           (list ly:self-alignment-interface::x-aligned-on-self)))))
```

In this example, both `ly:self-alignment-interface::x-aligned-on-self` and `ly:self-alignment-interface::centered-on-x-parent` are called with the grob as argument. The results are added with the `+` function. To ensure that this addition is properly executed, the whole thing is enclosed in `ly:make-simple-closure`.

In fact, using a single procedure as property value is equivalent to

```
(ly:make-simple-closure (ly:make-simple-closure (list proc)))
```

The inner `ly:make-simple-closure` supplies the grob as argument to *proc*, the outer ensures that result of the function is returned, rather than the `simple-closure` object.

From within a callback, the easiest method for evaluating a markup is to use grob-interpret-markup. For example:

```
my-callback = #(lambda (grob)
                 (grob-interpret-markup grob (markup "foo")))
```

2.8 Inline Scheme code

TODO: after this section had been written, LilyPond has improved to the point that finding a *simple* example where one would *have* to revert to Scheme has become rather hard.

Until this section gets a rewrite, let's pretend we don't know.

The main disadvantage of \tweak is its syntactical inflexibility. For example, the following produces a syntax error (or rather, it did so at some point in the past).

```
F = \tweak font-size #-3 -\flageolet

\relative c'' {
  c4^\F c4_\F
}
```

Using Scheme, this problem can be avoided. The route to the result is given in Section 1.3.4 [Adding articulation to notes (example)], page 15, especially how to use \displayMusic as a helping guide.

```
F = #(let ((m (make-music 'ArticulationEvent
                          'articulation-type "flageolet")))
       (set! (ly:music-property m 'tweaks)
             (acons 'font-size -3
                    (ly:music-property m 'tweaks)))
       m)

\relative c'' {
  c4^\F c4_\F
}
```

Here, the tweaks properties of the flageolet object m (created with make-music) are extracted with ly:music-property, a new key-value pair to change the font size is prepended to the property list with the acons Scheme function, and the result is finally written back with set!. The last element of the let block is the return value, m itself.

2.9 Difficult tweaks

There are a few classes of difficult adjustments.

- One type of difficult adjustment involves the appearance of spanner objects, such as slurs and ties. Usually, only one spanner object is created at a time, and it can be adjusted with the normal mechanism. However, occasionally a spanner crosses a line break. When this happens, the object is cloned. A separate object is created for every system in which the spanner appears. The new objects are clones of the original object and inherit all properties, including \overrides.

  In other words, an \override always affects all pieces of a broken spanner. To change only one part of a spanner at a line break, it is necessary to hook into the formatting process. The after-line-breaking callback contains the Scheme procedure that is called after the line breaks have been determined and layout objects have been split over different systems.

  In the following example, we define a procedure my-callback. This procedure

  - determines if the spanner has been split across line breaks
  - if yes, retrieves all the split objects
  - checks if this grob is the last of the split objects
  - if yes, it sets extra-offset.

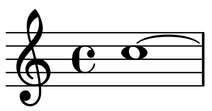
  This procedure is installed into Section "Tie" in *Internals Reference*, so the last part of the broken tie is repositioned.

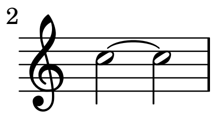
```
#(define (my-callback grob)
   (let* (
          ;; have we been split?
          (orig (ly:grob-original grob))

          ;; if yes, get the split pieces (our siblings)
          (siblings (if (ly:grob? orig)
                        (ly:spanner-broken-into orig)
                        '())))

     (if (and (>= (length siblings) 2)
               (eq? (car (last-pair siblings)) grob))
         (ly:grob-set-property! grob 'extra-offset '(-2 . 5)))))

\relative c'' {
  \override Tie.after-line-breaking =
  #my-callback
  c1 ~ \break
  c2 ~ c
}
```





When applying this trick, the new `after-line-breaking` callback should also call the old one, if such a default exists. For example, if using this with `Hairpin`, `ly:spanner::kill-zero-spanned-time` should also be called.

- Some objects cannot be changed with `\override` for technical reasons. Examples of those are `NonMusicalPaperColumn` and `PaperColumn`. They can be changed with the `\overrideProperty` function, which works similar to `\once \override`, but uses a different syntax.

```
\overrideProperty
Score.NonMusicalPaperColumn       % Grob name
  . line-break-system-details     % Property name
  . next-padding                  % Optional subproperty name
  #20                             % Value
```

Note, however, that `\override`, applied to `NonMusicalPaperColumn` and `PaperColumn`, still works as expected within `\context` blocks.

3 LilyPond Scheme interfaces

This chapter covers the various tools provided by LilyPond to help Scheme programmers get information into and out of the music streams.

TODO – figure out what goes in here and how to organize it

Appendix A GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its

Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix B LilyPond index