CS50 Seminar: Type-ing Music with Haskell and LilyPond

Connor Harris and Stephen Krewson

November 4, 2015

Table of contents

1. Euterpea
2. LilyPond

Acknowledgements

Thank you to Mark Santolucito and Donya Quick. And, of course, to the person who introduced me (and the larger world) to Haskell, Paul Hudak. (SK)
Thanks to the entire LilyPond community for creating a wonderful resource, and especially to Simon Albrecht for answering many of my questions on the lilypond-user mailing list; and more generally to everyone who indulged my interests for music and programming. (CH)

Prerequisites

- Install Haskell Platform
- "cabal update" and "cabal install Euterpea"
- Configure a MIDI synthesizer if your machine doesn't have one
- Pick a SoundFont to allow the MIDI synth to realize the array of sound patches specified in General MIDI standard (cf. typeface vs. font)
- Install LilyPond
- Ensure midi2ly Python script will execute by adding {path to LilyPond}/usr/bin to your system's PATH. You may need to add .PY to your PATHEXT as well.

Resources

The Haskell School of Music textbook can be downloaded at `http://haskell.cs.yale.edu/publications/`. This is the definitive text for Euterpea. I have taken some screenshots from HSoM and some from "A History of Haskell: Being Lazy with Class" (2007). Available at `http://haskell.cs.yale.edu/wp-content/uploads/2011/02/history.pdf`.
For a great illustrated tutorial check out *Learn You a Haskell for Great Good!* at `http://learnyouahaskell.com/introduction`.

Computer Music

- Algorithmic composition
- Digital synthesis
- Digital sampling

FM Synthesis

Musical Instrument Digital Interface (1983)

- Data transfer protocol
- Guide for organizing sounds (patch map)
- Hardware interface

General MIDI

MIDI MANUFACTURERS ASSOCIATION

Google™ Custom Search Search

Home | Learn About MIDI | About Us | Specifications | Career Center | Public Forum | Store

General MIDI Level 1 Sound Set

General MIDI's most recognized feature is the defined list of sounds or "patches". However, General MIDI does not actually define the way the sound will be reproduced, only the name of that sound.

Though this can obviously result in wide variations in performance from the same song data on different GM sound sources, the authors of General MIDI felt it important to allow each manufacturer to have their own ideas and express their personal aesthetics when it comes to picking the exact timbres for each sound.

Each manufacturer must insure that their sounds provide an acceptable representation of song data written for General MIDI. Guidelines for developing GM compatible sound sets and song data are available through the MMA.

- The names of the instruments indicate what sort of sound will be heard when that instrument number (MIDI Program Change or "PC#") is selected on the GM1 synthesizer.
- These sounds are the same for all MIDI Channels except Channel 10, which has only percussion sounds and some sound "effects". (See "GM1 Percussion Key Map")

General MIDI Level 1 Instrument Families

The General MIDI Level 1 instrument sounds are grouped by families. In each family are 8 specific instruments.

PC#	Family Name	PC#	Family Name
1-8	Piano	65-72	Reed
9-16	Chromatic Percussion	73-80	Pipe
17-24	Organ	81-88	Synth Lead
25-32	Guitar	89-96	Synth Pad
33-40	Bass	97-104	Synth Effects
41-48	Strings	105-112	Ethnic
49-56	Ensemble	113-120	Percussive
57-64	Brass	121-128	Sound Effects

General MIDI Level 1 Instrument Patch Map

Note: While GM1 does not define the actual characteristics of any sounds, the names in parentheses after each of the synth leads, pads, and sound effects are, in particular, intended only as guides).

PC#	Instrument Name	PC#	Instrument Name
1.	Acoustic Grand Piano	65.	Soprano Sax
2.	Bright Acoustic Piano	66.	Alto Sax
			Tenor Sax

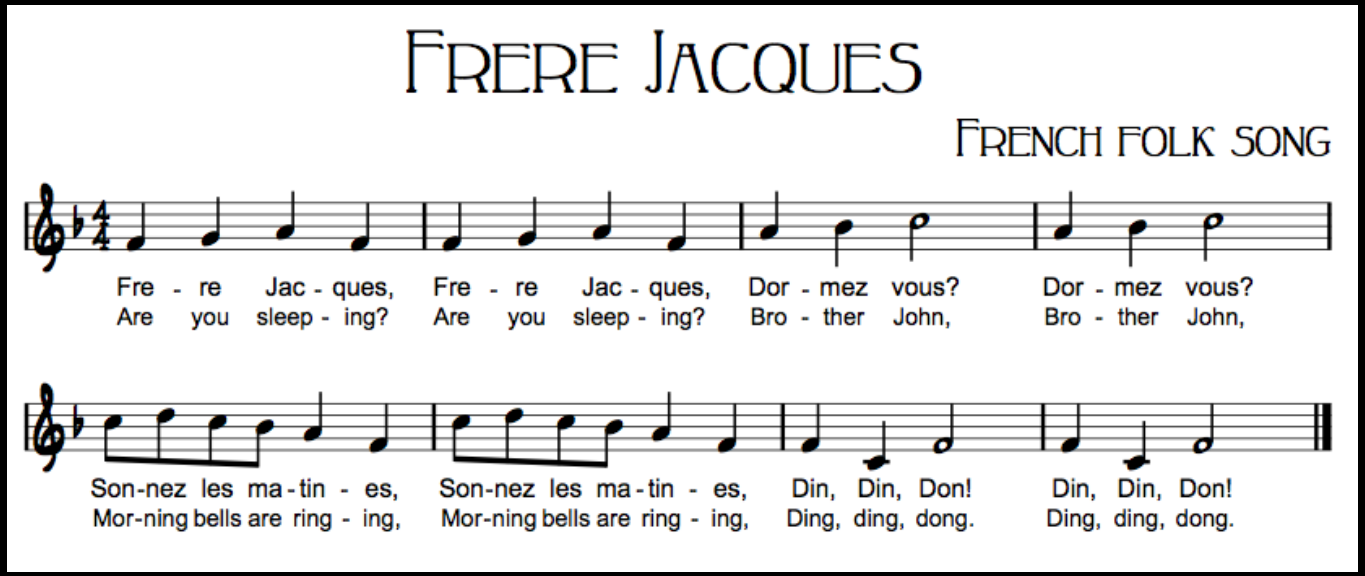
MIDI Drum Map

General MIDI Percussion Map

Channel 10

Key / Perc. sound	Key / Perc. sound	Key / Perc. sound
35. Acoustic Bass Drum	51. Ride Cymbal 1	67. High Agogo
36. Bass Drum 1	52. Chinese Cymbal	68. Low Agogo
37. Side Stick	53. Ride Bell	69. Cabasa
38. Acoustic Snare	54. Tambourine	70. Maracas
39. Hand Clap	55. Splash Cymbal	71. Short Whistle
40. Electric Snare	56. Cowbell	72. Long Whistle
41. Low Floor Tom	57. Crash Cymbal 2	73. Short Guiro
42. Closed Hi-Hat	58. Vibraslap	74. Long Guiro
43. High Floor Tom	59. Ride Cymbal 2	75. Claves
44. Pedal Hi-Hat	60. High Bongo	76. Hi Woodblock
45. Low Tom	61. Low Bongo	77. Low Woodblock
46. Open Hi-Hat	62. Mute Hi Conga	78. Mute Cuica
47. Low-Mid Tom	63. Open Hi Conga	79. Open Cuica
48. High-Mid Tom	64. Low Conga	80. Mute Triangle
49. Crash Cymbal 1	65. High Timbale	81. Open Triangle
50. High Tom	66. Low Timbale	

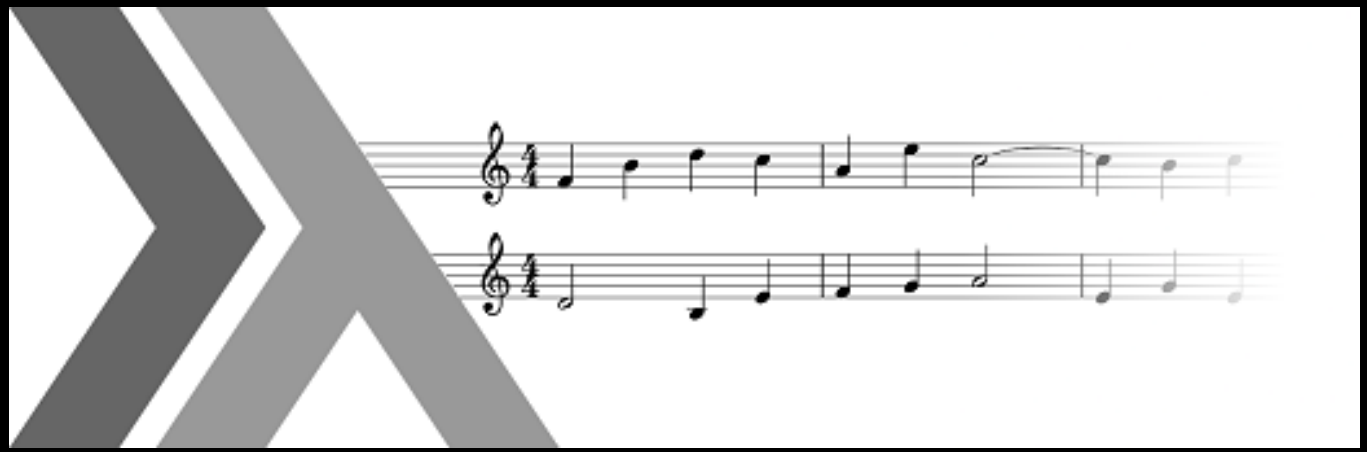
Composability: the case for Haskell scores



Composability: the case for Haskell controllers

Haskell (and Euterpea) Features

An apology: this is just a *sketch* of some aspects of Haskell and its suitability for computer music. Check out the code online. Fire up GHCi and :l[oad] some Haskell files. Use :browse and especially :t[ype]. If I achieve one thing, I hope it's showing you how fun exploring Haskell can be!



Lazy

- Non-strict evaluation strategy or "call-by-need." That is, delay evaluation until a term is required. Permits infinite objects!
- Accumulating over (potentially infinite) lists. Associativity and strictness. What can we get away with NOT computing? Consider (1 - (2 - (3 - 0))) vs. (((0 - 1) - 2) - 3). Consider (1 + (2 + (3 + 0))) vs. (((0 + 1) + 2) + 3)
- foldr (+/-) 0 [1..3]
- foldl (+/-) 0 [1..3]

```
1  p5 = stepSequence (Just 127) 27 $ take gN $ foldr
     (:) [] $ [3,8,21] ++ [1,6..]
```

Pure

Strong static typing. Functions are mathematical in the sense that f :: Int $->$ Int is guaranteed to not access or change any mutable variables or perform I/0. (f 3) will ALWAYS return the same value.

Type Classes

Solved overloading, inspired a type-system "laboratory". Think about the notes as laid out on a keyboard. What operations should musical values support?

```
data PitchClass = Cff | Cf | C | Dff | Cs | Df | Css | D | Eff | Ds
                | Ef | Fff | Dss | E | Ff | Es | F | Gff | Ess | Fs
                | Gf | Fss | G | Aff | Gs | Af | Gss | A | Bff | As
                | Bf | Ass | B | Bs | Bss
  deriving (Eq, Ord, Show, Read, Enum, Bounded)
data Primitive a = Note Dur a
                 | Rest Dur
  deriving (Show, Eq, Ord)
data Music a =
    Prim (Primitive a)          -- primitive value
  | Music a :+: Music a         -- sequential composition
  | Music a :=: Music a         -- parallel composition
  | Modify Control (Music a)    -- modifier
  deriving (Show, Eq, Ord)
data Control =
    Tempo      Rational             -- scale the tempo
  | Transpose  AbsPitch             -- transposition
  | Instrument InstrumentName       -- instrument label
  | Phrase     [PhraseAttribute]    -- phrase attributes
  | Player     PlayerName           -- player label
```

Currying

- A function of two arguments, f x y, can be represented as (f x) y. That is, a function of one argument that returns a function of one argument! This allows mapping (f x) over a list of y's.
- Consider the intermediate form [[youSleeping, youSleeping],[brotherJohn, brotherJohn],[morningBells, morningBells],[dingDong, dingDong]] (remember that each of these phrases is itself a list, necessitating two concats)

```
1 song = concat $ concatMap (replicate 2) [youSleeping,
    brotherJohn, morningBells, dingDong]
```

Lambda Expressions

Suppose we have some operation that, unlike replicate, is not defined in the standard library. Well we can define it *anonymously* and then can map it in curried fashion. Here's a small helper function defined within drumMachine that uses lambda syntax. Note that this kind of filter is simply the mapping of a Boolean function over a list (what does it do, btw?).

```
1    f n p = (\x -> x `mod` n == p)
```

Infix Operators

- Math style (vs. prefix)
- Can define custom operators! (:+:, :=:, /=:)
- quot x xs becomes a ‘quot‘ b
- Helpful $ infix operator (low precedence)

```
1 dingDong = [(c 4 qn :=: e 3 qn), (f 3 qn :=: g 3 qn),
      (c 4 hn :=: e 3 hn)]
```

Sections

Infix operators are themselves "first class" functions! Remember partial application. Consider map (+2) [1,2,3] or even map (2+) [1,2,3]

```
1   p8 = stepSequence Nothing 41 $ take gN $ iterate
      (+3) 2
```

List Comprehensions

How does this emulate the classic 16-step sequencer, adding EQ (well, at least volume) to the particular ‘part‘ being generated?

```
1 stepSequence v p xs = addVolume v' $ line [if x `elem`
      xs then perc (toEnum p) qn else qnr | x <- [1..gN]]
```

Algebraic Types

A data type created by combining other data types either as combinations (products) or alternations (sums). Note how the Maybe type is a sum of either (Just a) or (Nothing). We use this to assign our drum machine optional track-specific volumes (and a default value).

5.1 Algebraic types

Here is a simple declaration of an algebraic data type and a function accepting an argument of the type that illustrates the basic features of algebraic data types in Haskell.

```
data Maybe a = Nothing | Just a

mapMaybe :: (a->b) -> Maybe a -> Maybe b
mapMaybe f (Just x) = Just (f x)
mapMaybe f Nothing  = Nothing
```

```
1 stepSequence :: Maybe Volume -> Int -> [Int] -> Music
      (Pitch, Volume)
```

Pattern Matching

Extremely powerful and readable when you can define a base case and then a typical case. You can use underscore to placehold for any value.

```
1 stepSequence _ _ [] = rest 0
2 stepSequence v p xs = addVolume v' $ line [if x `elem`
      xs then perc (toEnum p) qn else qnr | x <- [1..gN]]
3   where v' = fromMaybe 75 v
```

Code Walkthrough: Frere Jacques

- There are x phrases, each repeated y times.
- How do we implement a round? (many ways to do this!)
- Note the GM patch names. Think of the instruments and volumes as modifications or annotations of the underlying musical notes.

```
1 frereJacques =
2   let
3   part1 = instrument Helicopter $ addVolume 127 (line
      song)
4   part2 = instrument Pad3Polysynth $ addVolume 127
      (line (rest 2 : song))
5   part3 = instrument Lead1Square $ addVolume 70 (line
      (rest 4 : song))
6   part4 = instrument VoiceOohs $ addVolume 127 (line
      (rest 6 : song))
7   in (part1 :=: part2 :=: part3 :=: part4)
```

Code Walkthrough: Drum Machine

Big idea. Make a bunch of lists ("parts"). Use a bunch of techniques to generate values in the range [1..r], where r is some global value for the number of "steps" (quarter notes scaled to 4x time at 120 bpm). Assign each list A_j to a percussion sound. As we step through r starting at 1, if $r_i \in A_j$, then that sample gets triggered.

Code Walkthrough: Drum Machine

```
1  drumMachine :: Music (Pitch, Volume)
2  drumMachine = let r = [1..gN]
3    f n p = (\x -> x `mod` n == p)
4    p1 = stepSequence (Just 127) 1 $ filter (f 4 1) r
5    p2 = stepSequence Nothing 7 $ filter (f 4 3) r
6    p3 = stepSequence (Just 127) 0 [7,10,14,23,26,30]
7    p4 = stepSequence Nothing 4 $ filter (f 8 5) r
8    p5 = stepSequence (Just 127) 27 $ take gN $ foldr
       (:) [] $ [3,8,21] ++ [1,6..]
9    p6 = stepSequence Nothing 28 $ map (`mod` gN) $
       scanr (+) 1 [1..5]
10   p7 = stepSequence Nothing 29 $ zipWith (*) [...]
11   p8 = stepSequence Nothing 41 $ take gN $ iterate
       (+3) 2
12   p9 = stepSequence Nothing 35 $ filter (not . (f 4
       1)) r
13   p10 = stepSequence Nothing 3 [7,15,25,28,32]
14 in tempo 4 $ instrument Percussion $ repeatM $ chord
       [p1,p2,p3,p4,p5,p6,p7,p8,p9,p10]
```

Time permitting...

- drumMachine deliberately combines a kitchen sink of filtering and list generation techniques. The goal is not concision or efficiency but demonstration of Haskell's expressive power.
- folds and scans and higher-order thinking
- Musical equivalents for higher-order functions (takeM, repeatM, etc.)
- Infinite musical values! How long will drumMachine run?

```
1 map' f = foldr (\x xs -> f x : xs) []
```

On to LilyPond!

This is all free, open-source software. So we can glue together some scripts on the command line to generate a score for our song!

```
1 midi2ly song.midi
2 lilypond song-midi.ly
```

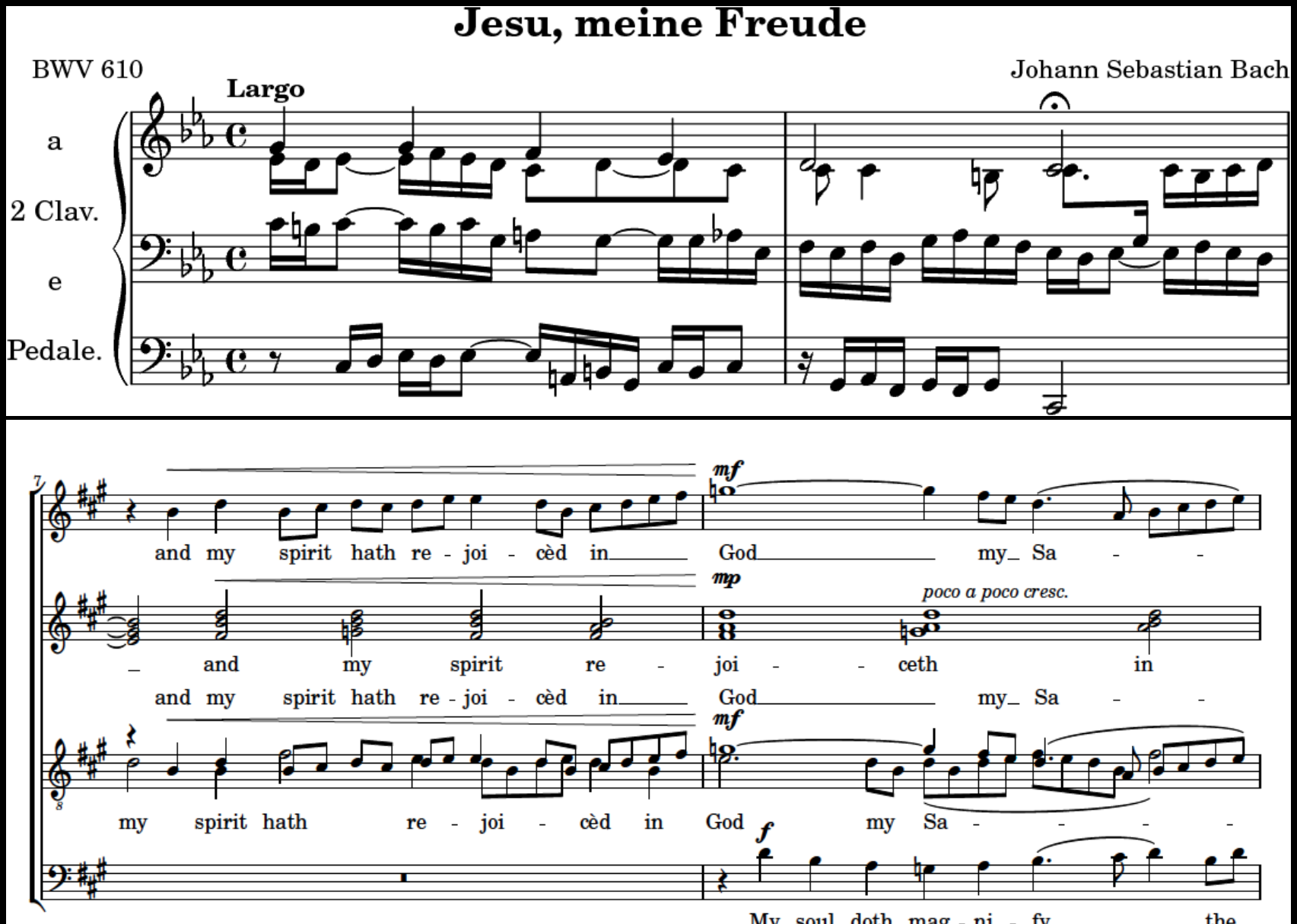
What is LilyPond?

- Declarative programming language for music typesetting . . .
- . . . and program that compiles the language into beautiful PDF scores.
- Analogy: LilyPond is to Finale, Sibelius, MuseScore, etc. as LaTeX is to Microsoft Word

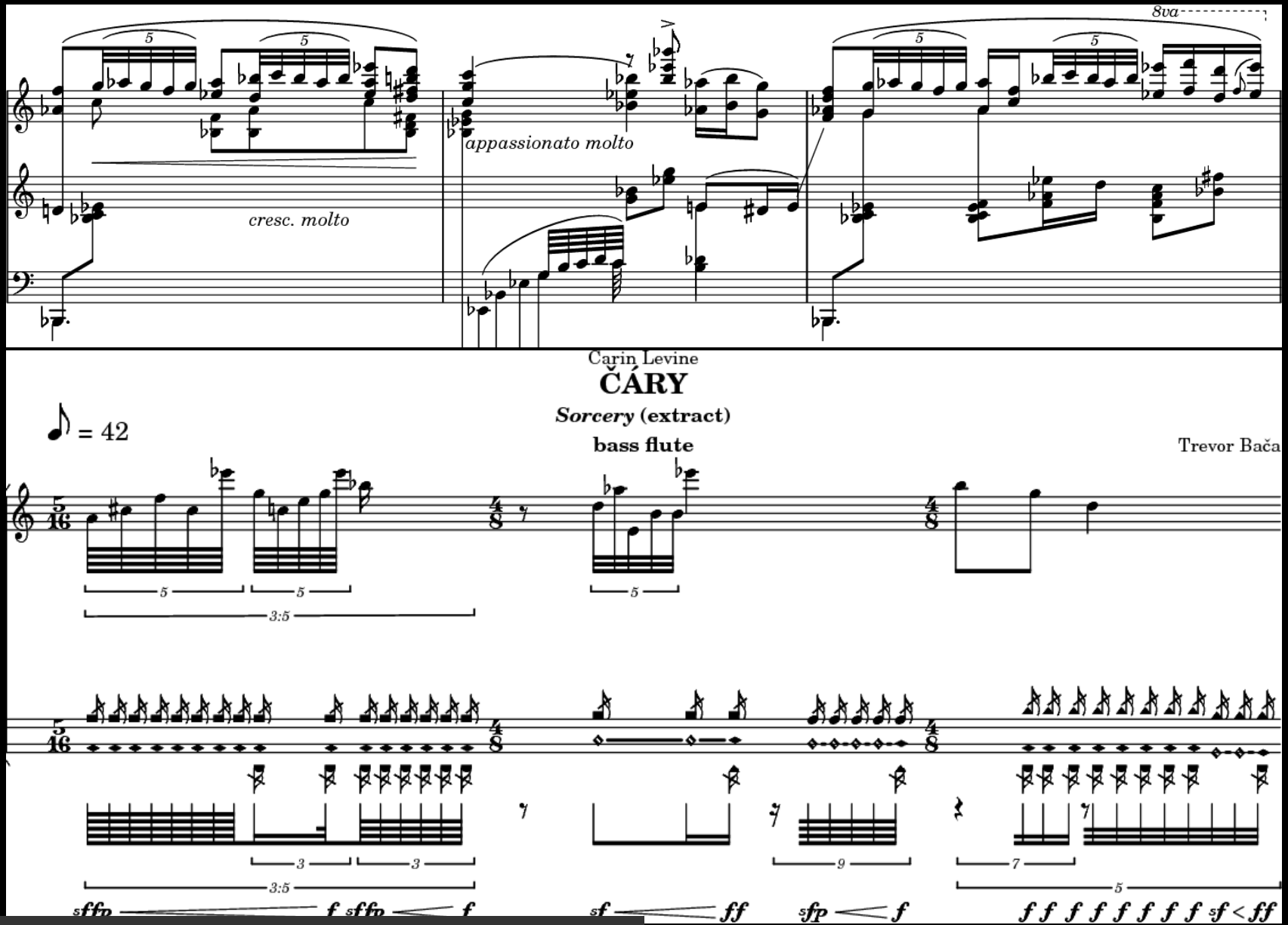
Why use LilyPond?

- Computer graphics is hard—don't write your own typesetting software.
- Finale, Sibelius, etc. have hard-to-read binary file formats, can't be used programmatically
- Easy integrability with LaTeX
- Looks beautiful!
- Good skill for life, not just CS50

Simple examples . . .



. . . and fancier ones



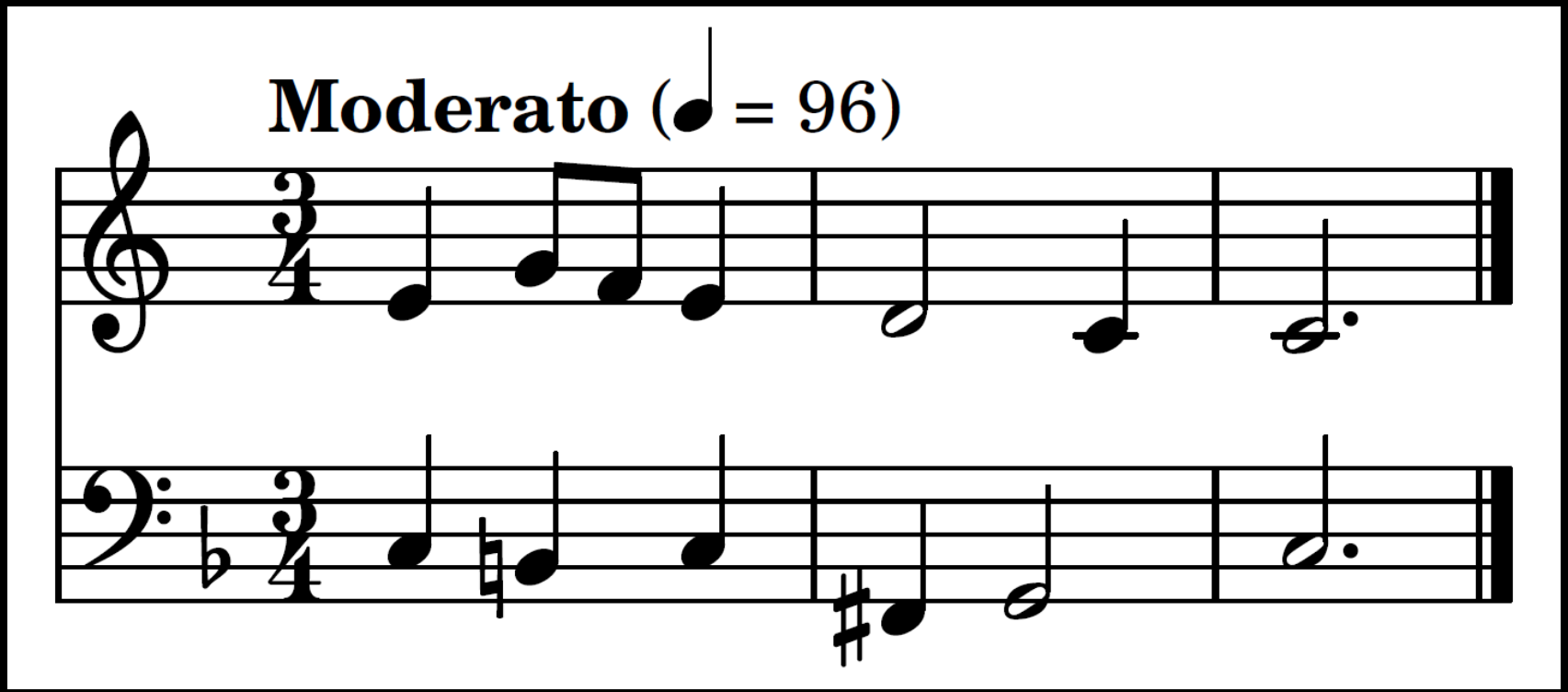
LilyPond object hierarchy, oversimplified

- Notes (and expression marks, text, etc.) are contained in a `Voice` context ...
- ... which can be contained hierarchically in other contexts (e.g. `Staff`, `PianoStaff`, `Score`) ...
- ... printed by attached engravers (e.g. `Note_heads_engraver` to `Voice`, `Clef_engraver` to `Staff`, `Metronome_mark_engraver` to `Score`)
- Extensively customizable! Modifiable context attributes, embedded scripting language (Scheme)

Code sample . . .

```
1 partOne = \relative c' {\time 3/4 \tempo "Moderato"
      4=96 e4 g8 f e4 | d2 c4 | c2. \bar "|." }
2 partTwo = \relative c { \key f \major c4 b c | fis, g2
      | c2. }
3 firstStaff = \new Staff { \partOne }
4 secondStaff = \new Staff {\clef bass \partTwo}
5 \score{ << \firstStaff \secondStaff>> }
```

. . . and its output



MusicXML

- XML spec for encoding music scores
- Most programs can write and read MusicXML files
- May be easier for your program to write than LilyPond files (depending on your internal object model and use of LilyPond's customization features)
- Any language worth its salt has an XML library
- Easier for users to modify with program of their choice
- Convertible to LilyPond format with `musicxml2ly` (bundled with LilyPond package)

Frescobaldi

- Free, open-source cross-platform editing environment for LilyPond, à la TeXstudio or TeXworks
- Split-screen PDF viewer
- Autocompletion and syntax highlighting for easier editing
- Template constructor for
- Free open-source download at `frescobaldi.org`

Other resources

- LilyPond manuals (`lilypond.org/manuals`), including a tutorial, syntax reference and code samples ("snippets")
- LilyPond internals reference (`lilypond.org/internals`): useful for making extensive customizations not covered by the snippets
- MusicXML tutorial (`musicxml.com/tutorial`): aimed at programmers who want their programs to output MusicXML files
- *Structure and Interpretation of Computer Programs* (`mitpress.mit.edu/sicp/`): introductory CS book using Scheme (the LilyPond embedded scripting language); also the second-greatest CS textbook ever written; you won't need more than the first few sections.