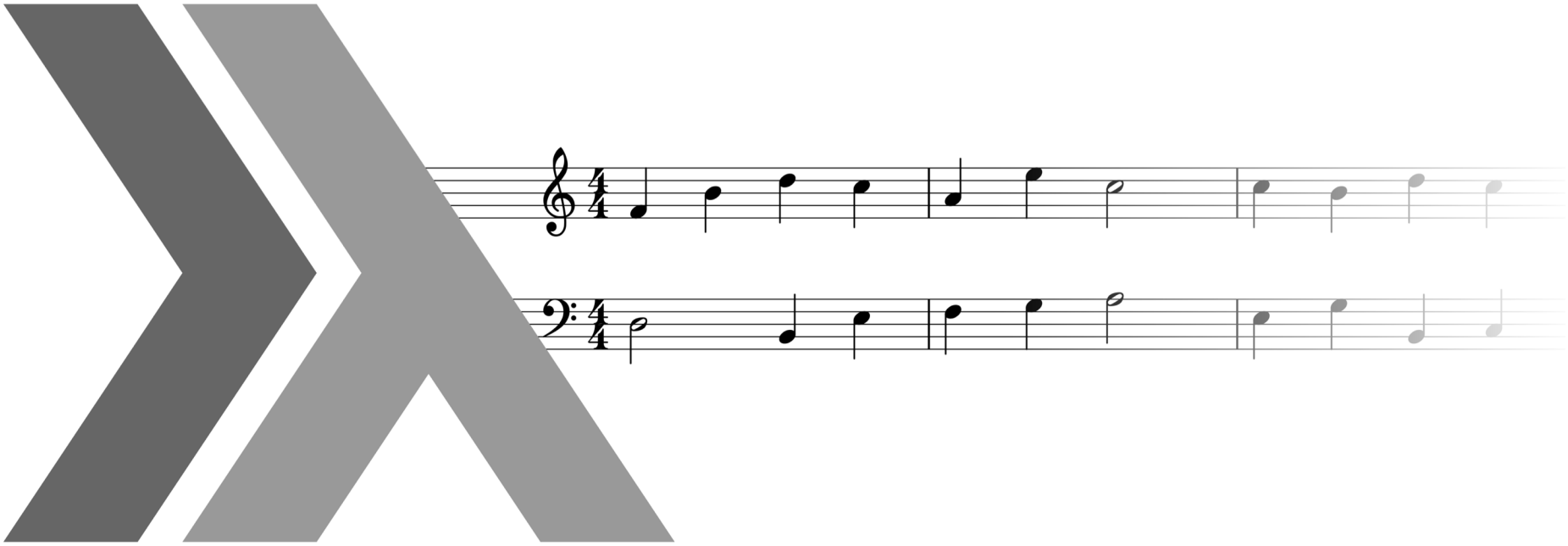
Making Algorithmic Music with Euterpea*

* with a brief invasion of a non-Haskell framework at the end (but it's still functionally-themed).



Donya Quick

Stevens Institute of Technology

About Me & What I Do

- Current research:
  - MUSICA project for human-computer interaction through natural language and music.
    - Modeling interactive jazz.
    - Modeling natural language for musical features.
- Other projects:
  - Kulitta: a framework for automated music composition.
    - I still work on this periodically but lately have been taking a step back from it to explore other strategies.
  - Euterpea (developer/maintainer): Haskell library for music representation originally created by Paul Huduak.
- I also write non-algorithmic music.

About Euterpea

- Available versions:
  - GitHub: 2.0.8* (development) – what I'm using today
  - Hackage: 2.0.7
- Go to http://www.euterpea.com for setup info.
  - Easiest way to get it:
    - Download Haskell Platform
    - Run: `cabal update`
    - Run: `cabal install Euterpea`
      (will give you version 2.0.7)
- Mac/Linux users: you need a software synth installed and running before you use Euterpea!
- Euterpea is not on Stack at this time.

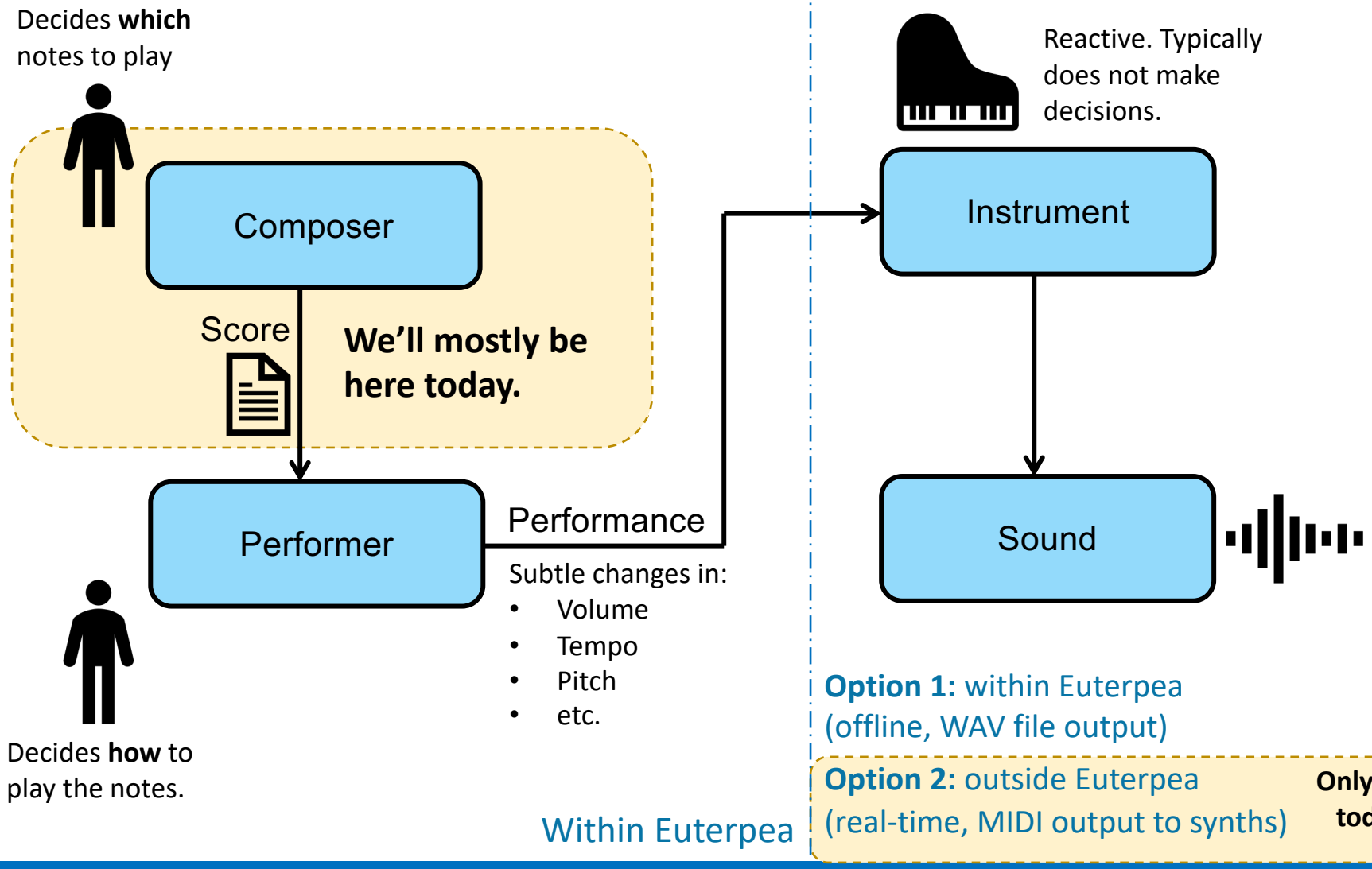
*2.0.8 has a tentative bug fix for something, but most users will be fine with 2.0.7.

What We're Looking at Today

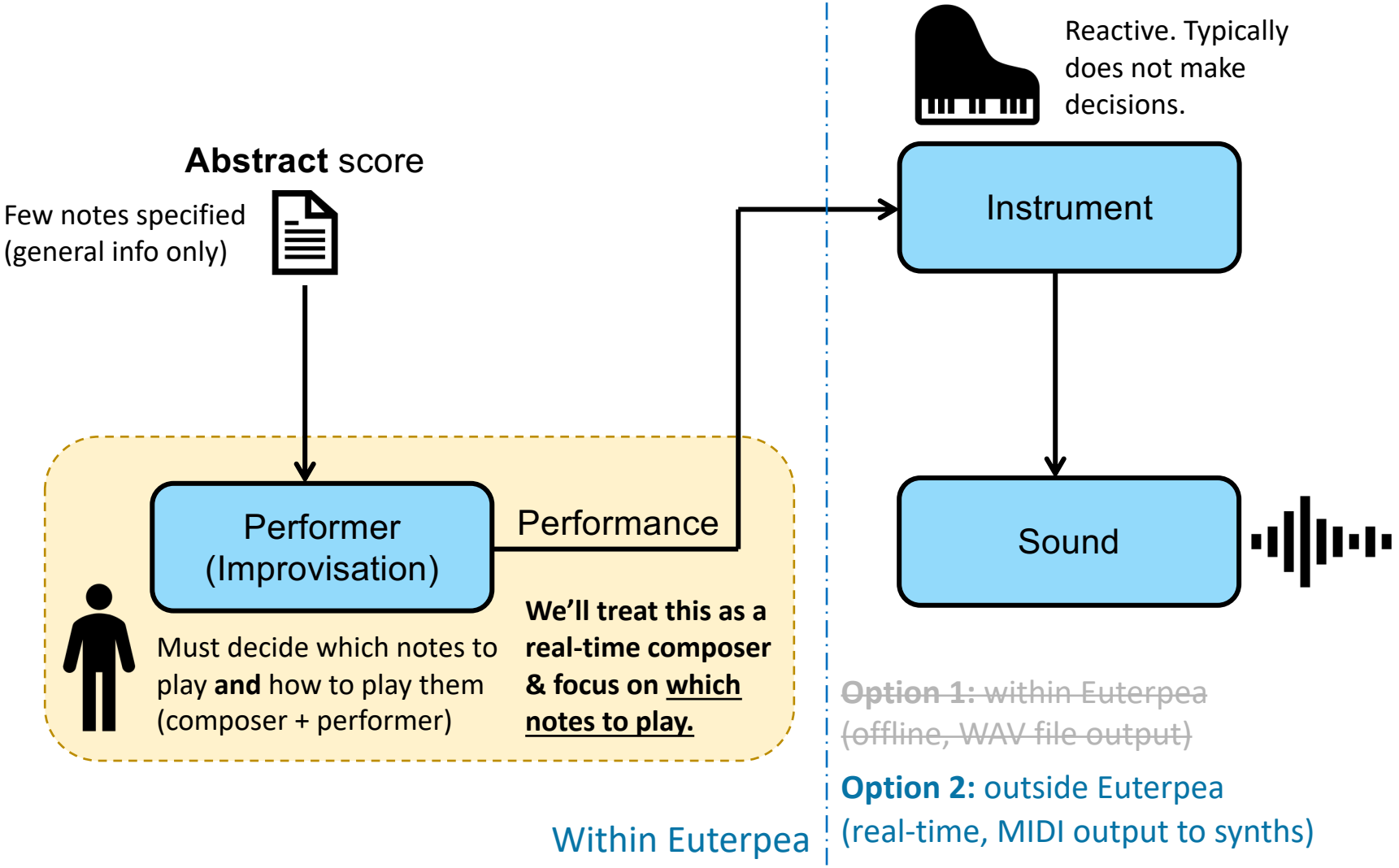
- A brief introduction to Euterpea
- Some simple stochastic music
- Modeling jazz improvisation

In other words, we're going to look at…

Composition vs. Performance



Improvisation



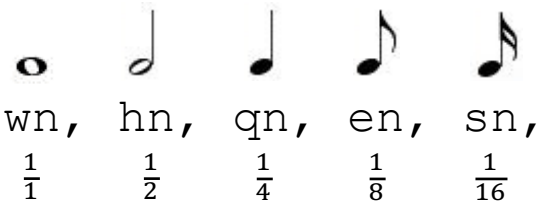
Euterpea at a Glance

```
type AbsPitch = Int
type Volume = Int
type Dur = Rational
```

0-127

0-127

with shorthands: `wn, hn, qn, en, sn, …`




```
data Primitive a = Note Dur a | Rest Dur
```

Holds pitch/volume information

```
data Music a = Prim (Primitive a)
    | (Music a) :+: (Music a)
    | (Music a) :=: (Music a)
    | …
```

combine in sequence

combine in parallel

```
note :: Dur -> a -> Music a
rest :: Dur -> a -> Music a
line :: [Music a] -> Music a
```

creates a single Note

combines everything sequentially

Some Playable instances of `Music a`:

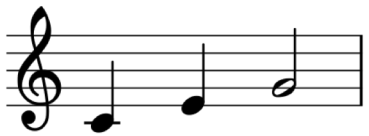
`Pitch` is a tuple of pitch class & octave. We won't be using this very much today.

```
Music Pitch
Music AbsPitch
Music (AbsPitch, Volume)
```

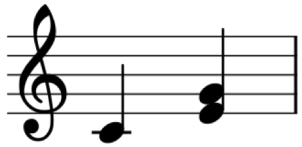
Requires Euterpea ≥2.0.5 from July 3, 2018

Making Some Notes (`Music Pitch`)

```
m1 :: Music Pitch
m1 = c 4 qn :+: e 4 qn :+: g 4 hn
```



```
m2 :: Music Pitch
m2 = c 4 qn :+: (e 4 qn :=: g 4 qn)
```



From GHCi, run: `play m1`
(and similarly for `m2`)

Durations:

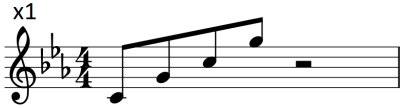
Making a Bunch More Notes (Music Pitch)

```
x1 = c 4 en :+: g 4 en :+: c 5 en :+: g 5 en
x2 = x1 :+: transpose 3 x1
x3 = x2 :+: x2 :+: invert x2 :+: retro x2
x4 = forever x3 :=: forever (tempo (2/3) x3)
```

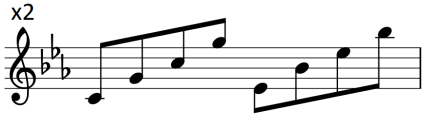
And with some more work…

Custom performance algorithm → Adjust tempo & use a good synth → Jam on it ("Blue Lambda")

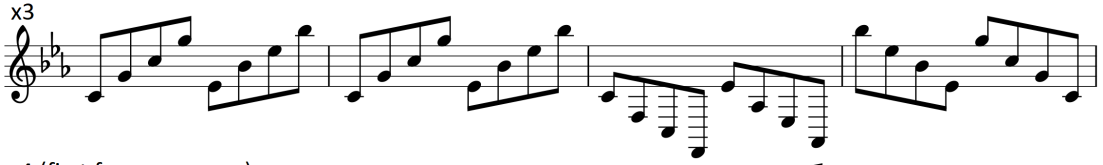
x1



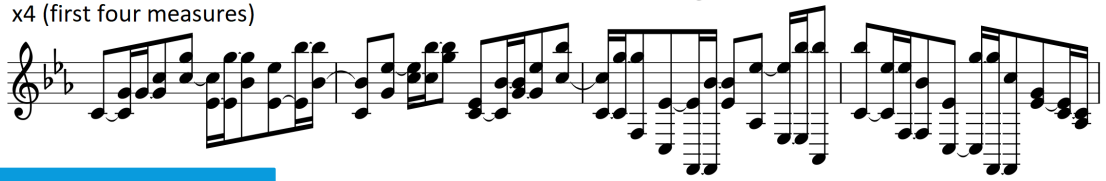
x2



x3



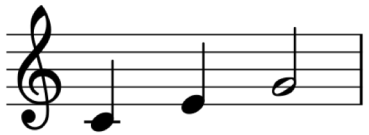
x4 (first four measures)



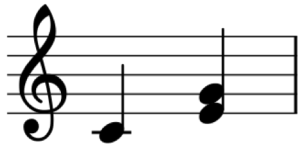
And if you put x4 through a crazy ring-modulator synth instead:

Making Some Notes with Numbers (`Music AbsPitch`)

```
m1 :: Music AbsPitch
m1 = note qn 60 :+: note qn 64 :+: note hn 67
```



```
m2 :: Music AbsPitch
m2 = note qn 60 :+: (note qn 64 :+: note qn 67)
```



Durations:

Pitch number translation:

60 = C4

64 = E4

67 = G4

We'll be using `AbsPitch` instead of `Pitch` from here on out. Be aware that some functions (like `invert`) don't work with `Music AbsPitch`. However, `Music AbsPitch` is much better for working with lists of numbers.

Random Numbers to Music

Code as would be typed into GHCi:

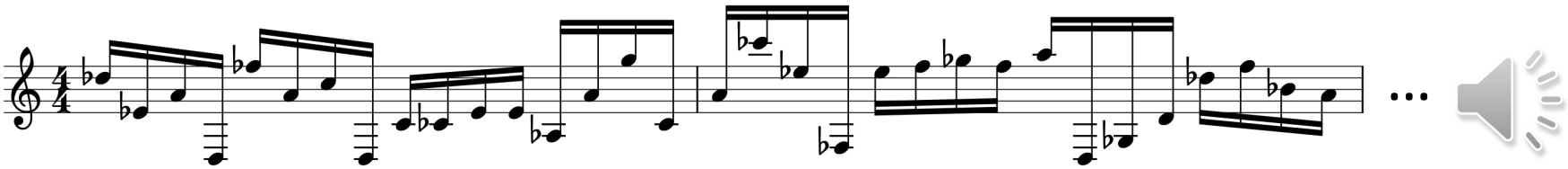
```
1 import Euterpea
2 import System.Random
3 g0 = mkStdGen 5
4 nums = randomRs (50::AbsPitch, 85) g0
5 m = line $ map (note sn) nums
6 play m
```

Random seed

Important for avoiding ambiguous type errors

`m :: Music AbsPitch`

(use **Ctrl+C** to stop infinite playback)



Randomness with Tonality & Volume

```
choose :: [a] -> StdGen -> (a, StdGen)
choose [] g = error "Nothing to choose from!"
choose xs g =
    let (i,g') = next g
    in  (xs !! (i `mod` length xs), g')
```

Key strategy: choosing from lists/sets to start introducing structure.

Pitches to pick from

Durations to pick from

Volume threshold

Current random generator

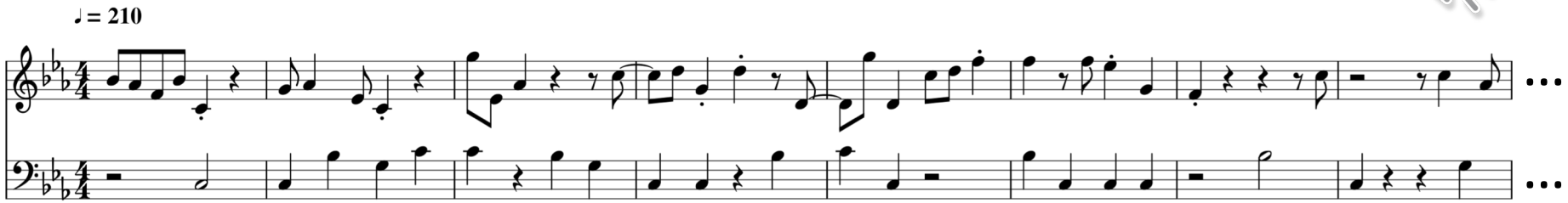
```
randomMel :: … -> Music (AbsPitch, Volume)
randomMel pitches durs thresh g0 =
    let (p, g1) = choose pitches g0     -- pick the next pitch
        (d, g2) = choose durs g1        -- pick the next duration
        (v, g3) = randomR (0,127) g2    -- pick the next volume
        x = if v < thresh then rest d   -- allow volume to dictate note vs. rest
            else note d (p, v)
    in  x :+: randomMel pitches durs thresh g3
```

The note or rest we just built, followed by…

A recursive call to build more notes (infinitely)

Adding a Bassline

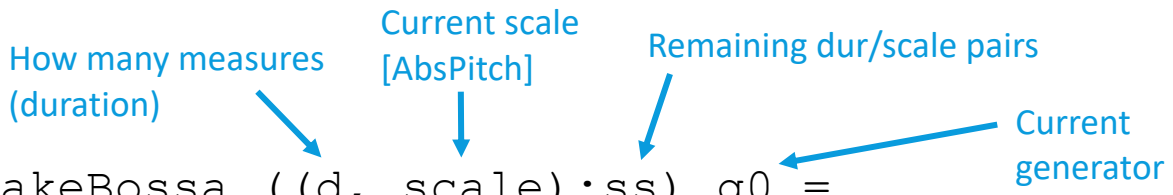
```
pitches1, pitches2 :: [AbsPitch[
pitches1 = [60,62,63,65,67,68,70,72,74,75,77,79]
pitches2 = [36,36,43,43,46,48]

mel1, mel2, duet :: Music (AbsPitch, Volume)
mel1 = randomMel pitches1 [qn,en,en,en] 40 0 (mkStdGen 500)
mel2 = randomMel pitches2 [hn,qn,qn,qn] 20 20 (mkStdGen 501)

duet = tempo 1.75 (
    instrument Vibraphone mel1 :=:
    instrument AcousticBass mel2)
```



It doesn't just have to sound jazzy. This is one uses the same strategy, but with analog synths, and adds some chords:

"Aquatic Level"

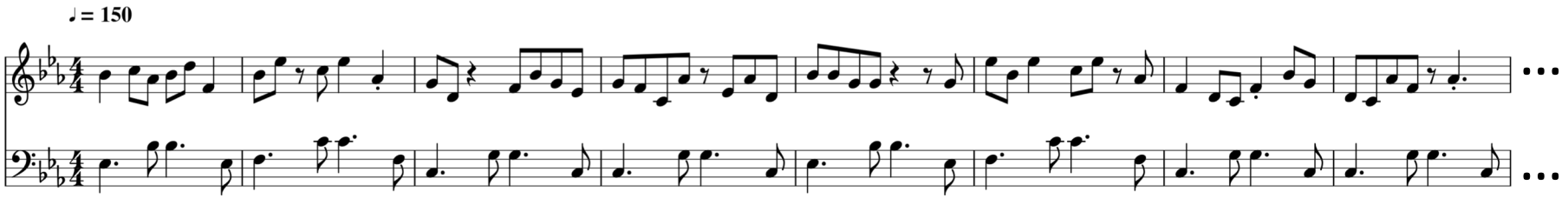
Key Changes & Some Bossa Nova




```
makeBossa ((d, scale):ss) g0 =
    let s1 = map (+60) scale        -- range for melody
        br = 36 + (scale !! 0)      -- root for bass
        bf = 36 + (scale !! 4)      -- fifth for bass
        (g1,g2) = split g0          -- split the generator
        mel = cut d (randomMel s1 [qn,en,en,en] 40 g1)
        bpat = note dqn (br,100) :+: note en (bf,80) :+:
               note dqn (bf,100) :+: note en (br,80)
        bass = cut d (forever bpat)
    in  (instrument Marimba mel :=:
         instrument AcousticBass bass) :+:
         makeBossa ss g2
makeBossa [] g0 = rest 0
```

Key Changes & Some Bossa Nova

```
efMaj, fMin, cMin :: Scale
efMaj = [3,5,7,8,10,12,14]
fMin = [5,7,8,10,12,13,15]
cMin = [0,2,3,5,7,8,10]

myKeys :: [(Dur, Scale)]
myKeys = [(1, efMaj), (1, fMin), (2, cMin)] ++ myKeys
bossa = tempo 1.25 $ makeBossa myKeys (mkStdGen 123)
```



And if we randomize the keys and add some chords…

Modeling Improvisation

Each part is basically a function…

…from a **harmonic context**…

…to notes played.

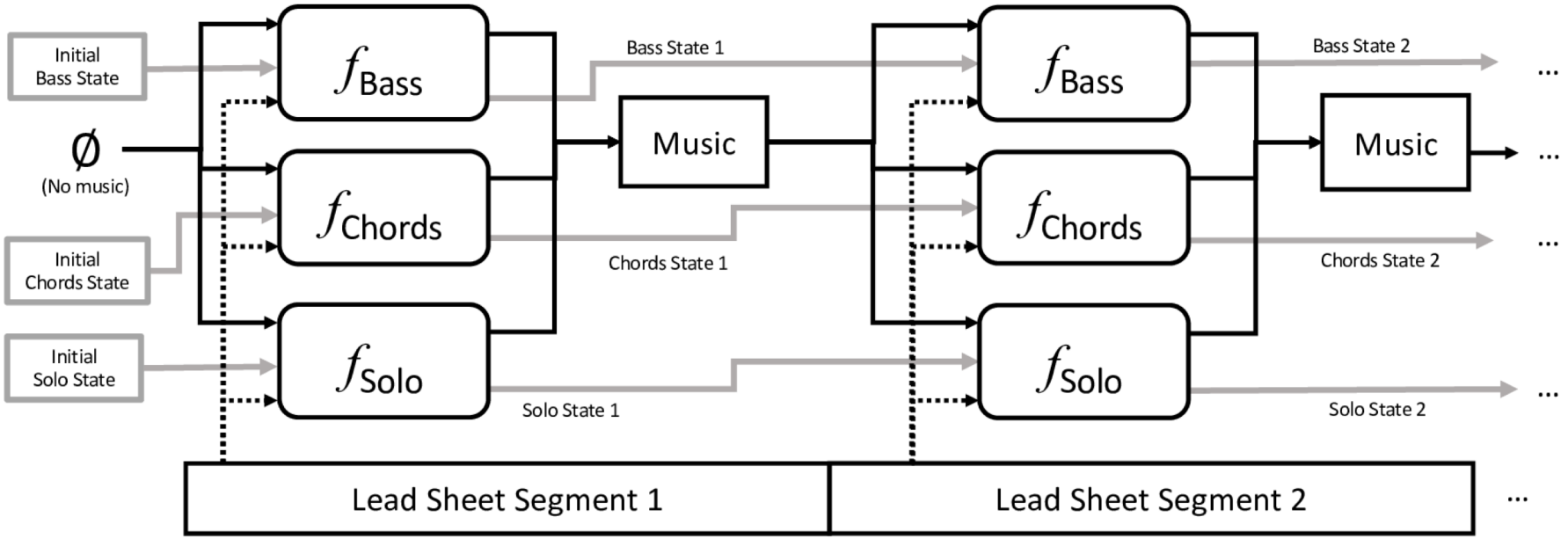
So can we simply model jazz like this?

```
type Context = ...
type JazzFun a = Context -> Music a
```

Yes!

Well…mostly. There are some other important kinds of contexts to deal with, but they are still contexts. The function bodies will also be complex.

Modeling Improvisation



Some kinds of context:

- The current lead sheet segment (harmonic context)
- Each musician's current internal state
- Notes emitted by all performers so far

Jazzkell implements this model:
github.com/donya/jazzkell

Two Examples Using This Approach

The main difference between these and what we just built is consideration of additional context and more finessed decision-making.

- Hypnotize
  - Randomly generated lead sheet
  - Parts must adapt to the lead sheet on the fly.
  - Implementation included in Jazzkell's examples folder

- Ugly Purse Dog
  - Ugly Purse Dog (Noun): a small, scruffy, angry animal that lives inside a luxury handbag on the NYC subways.
  - Mix of fixed and algorithmically generated lead sheets
  - Fixed parts follow the repeating pattern: Cm, Cm, AbM, BbM
  - Generated in large sections and put together in a DAW
  - Also exists as an interactive system (Python version of model)

Full compositions on `soundcloud.com/donyaquick`

Stateful Solos at a Glance

- Solos can be smoothed by:
  - Storing the last solo note played as part of the soloist's internal state.
  - Picking the next notes according to the following constraints:
    - Within the current scale with X probability (chromatic with 1-X probability)
    - Within a distance threshold of the last solo note (i.e. nearby)
    - Not repeating the last solo note (or probabilistically avoiding repeats)
- Soloist A can respond to soloist B by:
  - Storing observed output from soloist B's lead sheet segments
  - Using one or both of the following generative strategies:
    - Partial recombinance of the observed material (+ some novel material)
    - Pattern extraction in combination with pattern-based generation (Harder to do in real-time than recombinatory strategies)

One Way to Do a Walking Bass

- Note: this description assumes 4/4 for simplicity
- Given the current lead sheet segment, S1, the next segment, S2, and the current bass pitch, p0 (root of S1's scale):
  - Choose pitch p4 as an instance of the root of S2's scale.
  - Choose pitch p2 = f(p0,p4)
  - Choose pitch p1 = f(p0,p2)
  - Choose pitch p3 = f(p2,p4)
  - Return: map (note qn) [p0, p1, p2, p3]
  - In the next iteration, p0=p4 (saved in internal state).
- f(pitch1, pitch2) =
  - If pitch1 and pitch2 are far apart, choose a pitch between them.
  - If pitch1 and pitch 2 are 2 semitones apart, pick the pitch between.
  - Otherwise, pick a pitch that is near p2 (above or below).

Interactivity (Python version)

- Effectively a Python port of the Haskell algorithms.
  - All of the generative algorithms were prototyped in Haskell first and then translated (Python is also functional, so it was fairly direct)
- Why Python?
  - There are some stability issues in this type of scenario involving Euterpea's MIDI back-end dependencies.
  - The MUSICA project is heavily Python based.
    (Python is still a big thing in anything AI-related)
- Interactive version of "Ugly Purse Dog" 
  - Human: piano
  - Computer: everything else. Marimba borrows from the piano's material.

Thank You!

- Contact info: donyaquick@gmail.com | dquick@stevens.edu
- My site: www.donyaquick.com
- Euterpea site: www.euterpea.com
- Music on SoundCloud: www.soundcloud.com/donyaquick
  - Relevant playlists: Algo-Jazz & Made with Euterpea