A Musical Offering

Composing functions composing music

A Little Haskell

Functions!

```
simple a b = a + b
simple 1 1 --2
```

Lists! Cons-es!

```
l = 1:2:3:[]
--[1,2,3]
1:[2,3]
```

Pattern matching!

```
add2 [] = []
add2 (x:xs) = (2 + x) : (add2 xs)
add2 [1,2,3]
--[3,4,5]
```

A taste of Euterpea

What's a note?

```
concertA = (A, 4) --Tuple
quarterNote = 1/4 --Rational
qnA = note quarterNote concertA
```

Some functions that ship with Euterpea

```
qnA' = a 4 qn
play qnA'
staccatos = (d 4 en) :+: dqnr :+: (fs 4 en)
play (tempo 3 staccatos)
play $ tempo 3 staccatos
play $ instrument ChurchOrgan staccatos
```

Composing music

```
doReMi = c 4 qn :+: d 4 qn :+: e 4 qn
cMaj = c 4 qn :=: e 4 qn :=: g 4 qn
play doReMi
play cMaj
play (doReMi :+: cMaj)
```

Composing music

A familiar tune

```
fJacques =
   g 4 qn :+: a 4 qn :+: b 4 qn :+: g 4 qn :+:
   b 4 qn :+: c 5 qn :+: d 5 hn :+:
   d 5 den :+: e 5 sn :+: d 5 en :+:
   c 5 en :+: b 4 qn :+: g 4 qn :+:
   g 4 qn :+: e 4 qn :+: g 4 hn
play $ instrument Flute $ tempo 2
fJacques
```

Composing music

Another familiar tune

```
ludwigVan =
  (g 1 en :=: g 2 en :=: g 3 en :=: g 4 en) :+:
  (g 1 en :=: g 2 en :=: g 3 en :=: g 4 en) :+:
  (g 1 en :=: g 2 en :=: g 3 en :=: g 4 en) :+:
  (ef 1 dhn :=: ef 2 ddhn :=: ef 3 dhn :=: ef 4 dhn) :+:
 denr :+:
  (f 1 en :=: f 2 en :=: f 3 en :=: f 4 en) :+:
  (f 1 en :=: f 2 en :=: f 3 en :=: f 4 en) :+:
  (f 1 en :=: f 2 en :=: f 3 en :=: f 4 en) :+:
  (d 1 dhn :=: d 2 dhn :=: d 3 dhn :=: d 4 dhn)
play ludwigVan
```

The name of the wind

Type synonyms:

```
type Octave = Int
type Pitch = ( PitchClass, Octave )
type Dur = Rational
data PitchClass = Cff | Cf | C | Dff | Cs |
Df | Css | D --...
```

Haskell infers types, but we can tell it too:

```
qn :: Dur
qn = 1/4
(A, 4) :: Pitch
add2 :: [ Int ] -> [ Int ]
note :: Dur -> Pitch -> Music Pitch
a,b,c,d,e,f,g :: Octave -> Dur -> Music
Pitch
```

A Type of Music

Basic units of music:

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

We can think about these 'constructors' as functions:

```
Note :: Dur -> Pitch -> Primitive
Rest :: Dur -> Primitive
```

Okay definition... but notes are more than pitches (percussion, volume, etc.)

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

Which looks like:

```
Note :: Dur -> a -> Primitive a
Rest :: Primitive a
```

A fancier (recursive!) type

Music is not just single notes

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

Prim :: Primitive a -> Music a
(:+:) :: Music a -> Music a -> Music a
(:=:) :: Music a -> Music a -> Music a
```

Reasoning with types

```
line :: [ Music a ] -> Music a
line [] = rest 0
line (m:ms) = m :+: line ms
line [c 4 qn, e 4 qn, g 4 qn, c 5 qn]
chord :: [ Music a ] -> Music a
chord [c 4 qn, e 4 qn, g 4 qn, c 5 qn]
```

Pattern matching, TNG

```
majChord :: Music Pitch -> Music Pitch
majChord ( Prim ( Note d root)) =
   note d root :=:
   note d (trans 4 root) :=:
   note d (trans 7 root)
majChord _ = error 'only works for
notes!'
```

Pattern matching, TNG

```
majChord :: Music Pitch -> Music Pitch
majChord ( Prim ( Note d r@(pc, oct))) =
   note d r :=:
   note d (trans 4 r) :=:
   note d (trans 7 r) :=:
   note d (pc, o+1))
majChord _ = error 'only works for
notes!'
```

Fancier functions

Haskell ships with some pretty cool functions

```
foldr (+) 0 [1,2,3] --6
scanl (+) 10 [1,2,3] --[10,11,13,16]
foldr1 (+) [1,2,3]
[1,2,3] !! 1 --2
take 2 [1,2,3] --[1,2]
reverse [1,2,3] --[3,2,1]
twice x = x*2
map twice [1,2,3] --[2,4,6]
```

Fancier functions

And their signatures are very interesting. Can you guess them?

```
reverse :: [a] -> [a]
take :: Int -> [a] -> [a]
scanl :: (b -> a -> b) -> b -> [a] -> [b]
(!!) :: [a] -> Int -> a
```

Batteries included

Guess what these functions do:

```
mystery1 :: [Music a] -> Music a
mystery1 ns = foldl1 (:+:) ns
mystery1 [d 4 qn, fs 4 qn]
--Prim (Note (1 % 4) (D,4)) :+: Prim (Note
(1 % 4) (Fs,4))
```

Yep, that's our good old line function, refactored!

How about:

```
mystery2 :: [ Music a ] -> Music a
mystery2 ms = line $ map (transpose 12) ms
mystery1 [d 4 qn, fs 4 qn] :+: mystery2 [d
4 qn, fs 4 qn]
```

Haskell: Curry

Partial application is the default:

```
add a b = a + b
add2 = add 2
add2 40 --42
(add 2) 40
```

Which comes in handy:

```
line = foldr1 (:+:)
chord = foldr1 (:=:)
```

Haskell is lazy

And lazy evaluation is also a default:

```
forever :: Music x -> [ Music x ]
forever m = m : forever m
foreverA o dur = forever $ a o dur
foreverA4qn = foreverA 4 qn
```

No explosions, until...

```
play $ line $ take 2 foreverA4qn
```

Example: scales

How would we build a scale?

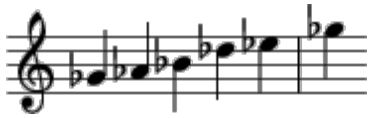
What we want:

```
mkScale [2,2,3,2] (fs 4 qn)
--Fs,Gs,As,Cs,Ds
```

Example: scales

How would we build a scale?

In Music:



Example: scales

How would we build a scale?

In English:

- Take a list of intervals
- Get each abs pitch relative to the root
- (e.g. [0, 0+2, 2+2, 4+3, 7+2]
- Turn those abs pitches into a list of notes

Example: scales

How would we build a scale?

In Haskell:

```
mkScale :: [Int] -> Music Pitch -> [Music
Pitch]
mkScale ints (Prim (Note d p)) =
  map (note qn . pitch) $
  scanl (+) (absPitch p) ints
```

Example: scales

How would we build a scale?

Which we can use to define stuff:

```
pentatonic = mkScale [2,2,3,2]
blackKeys = pentatonic $ fs 4 qn
play $ line blackKeys
```

Example: scales

How would we build a scale?

Or we can go crazy

```
mkScale ints (Prim (Note d p)) =
  map (note qn . pitch) $
  scanl (+) (absPitch p) (cycle ints)

mkChord scale degrees =
  chord $
  map ((scale!!) . (subtract 1))
      degrees
```

Example: scales

How would we build a scale?

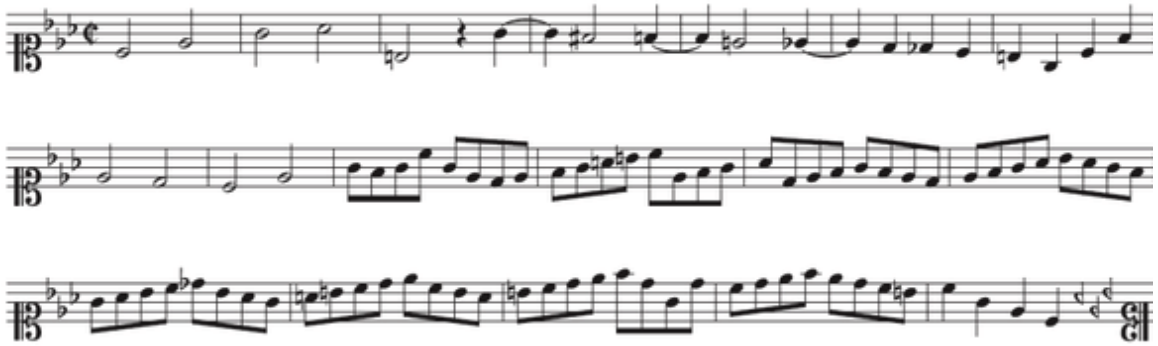
What do these do?

```
cMaj = mkScale [2,2,1,2,2,2,1] (c 4 qn)
play $ line $ take 16 cMaj
play $ mkChord cMaj [1,5,8,9,11]
```

A musical puzzle

Bach's 'Crab Canon'

All the score says is **`Canon 1 a 2`**



Solving the puzzle

Hint: there's a weird extra clef...

Solving the puzzle

We need to play it from both ends!

Solving the puzzle

Let's transcribe it first

```
crabTheme :: Music Pitch
crabTheme = line $
[rest 0, c 4 hn, ef 4 hn,
 g 4 hn, af 4 hn, b 3 hn,
--...
(staccato $ ef 4 qn), c 4 qn]
```

Solving the puzzle

Some helper functions

```
lineToList :: Music a -> [Music a]
lineToList (Prim (Rest 0)) = []
lineToList (n :+: ns) = n : lineToList ns
retrograde :: Music Pitch -> Music Pitch
retrograde = line . reverse . lineToList
staccato :: Music a -> Music a
staccato (Prim (Note d p)) =
  note (d/8) p :+: rest (7*d/8)
```

Solving the puzzle

And play that:

```
crabCanon :: Music Pitch
crabCanon =
  instrument Harpsichord $
  crabTheme :=:
  retrograde crabTheme
```

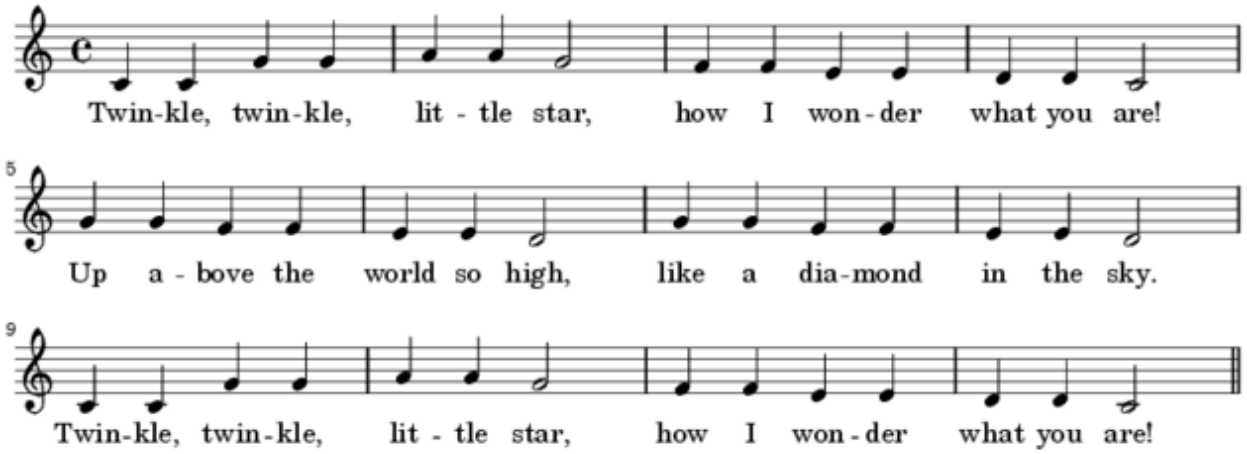
Takeaways

- Rephrase problems in terms of existing solutions
- Find the glue in your language (in Haskell: lazy, partial, high-ordered, functions)
- A clever type system can be a powerful tool, not a boilerplate prison
- Programming can be a tool to approach art, too!

Appendix

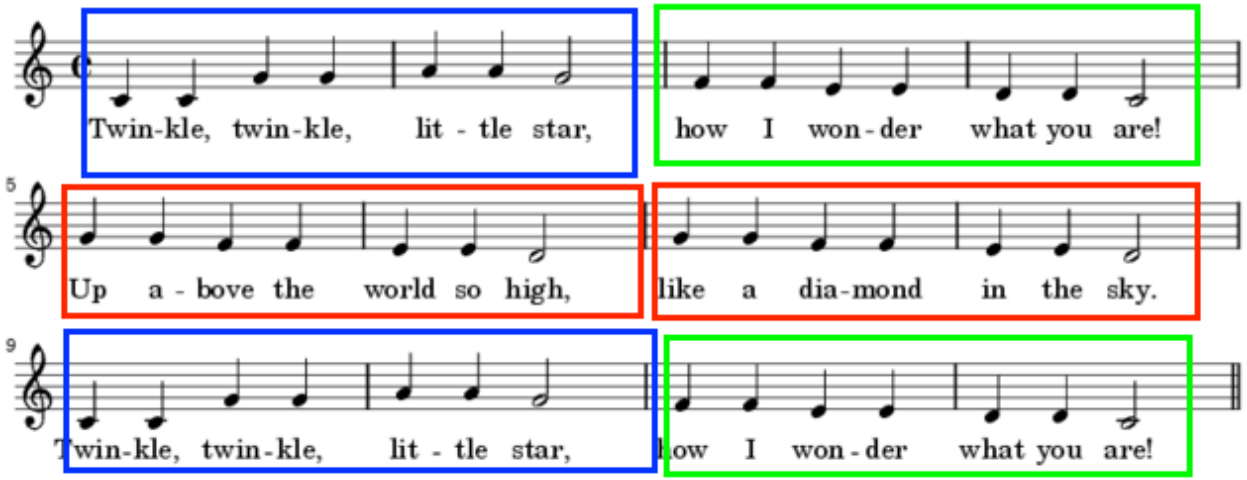
Let's use our knowledge for music!

Remember this tune?



Let's use our knowledge for music!

Remember this tune?



Let's use our knowledge for music!

Remember this tune?

```
pcToQn :: PitchClass -> Music Pitch
pcToQn pc = note qn (pc, 4)

twinkle =
  let m1 = line (map pcToQn
[C,C,G,G,A,A]) :+: g 4 hn
      m2 = line (map pcToQn
[F,F,E,E,D,D]) :+: c 4 hn
      m3 = line (map pcToQn
[G,G,F,F,E,E]) :+: d 4 hn
  in line [m1, m2, m3, m3, m1, m2]
```

Playing with music

```
times 0 m = rest 0; times n m = m :+: times
(n - 1) m
play $ twinkle :=: ((times 2 (rest hn))
:+: twinkle)
```

What if we want to generalize it?

```
canon :: (Int, Dur) -> Music a -> Music a
canon (2, hn) twinkle
```

A function for creating canons:

```
canon :: (Int, Dur) -> Music a -> Music a
canon (voices, delay) mel =
  let range n = take n [0..]
      wait d m n = times n (rest dur) :+: m
  in chord $ map
       (wait delay mel) (range voices)
```

With some interesting results:

```
play $ canon (2, hn) twinkle
play $ canon (2, qn) twinkle
play $ canon (2, en) twinkle
```