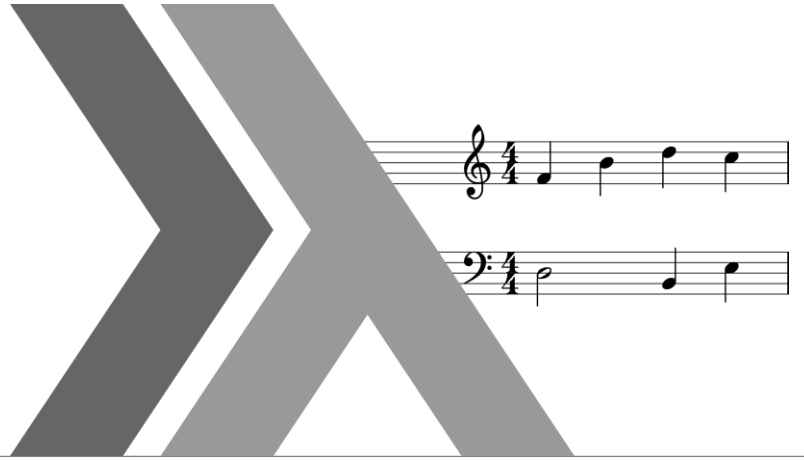
Euterpea's Music Types



DONYA QUICK

http://www.euterpea.com

Prerequisites

- Have Haskell and Euterpea installed.
  - Need to set up for the first time?
    Go to the Euterpea website: http://www.euterpea.com
    or watch the *Getting Started with Euterpea* tutorial video.
- Verify that MIDI is working.
  - Mac/Linux users: remember to start your synthesizer *before* starting GHC/GHCi.
  - Easy test: open GHCi and then run

```
import Euterpea
play $ c 4 qn
```

In This Tutorial

- Euterpea's types for pitch, duration, and volume.
- The basics of Euterpea's `Music` data type.
- Functions for quickly creating musical values.
- How to define a melody and some chords.
- Common problems people run into when working with Euterpea's music types for the first time.

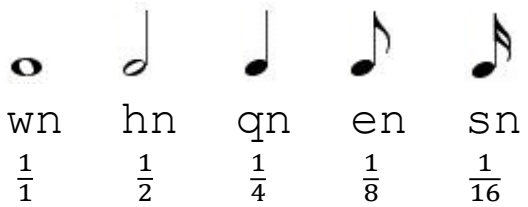
Pitch, Duration, and Volume Types

Pitch:

- `type AbsPitch = Int`
  MIDI standard absolute pitches (also called *pitch numbers*) from 0-127, where C4 = 60.
- `data PitchClass = C | Cs | ...`
  Symbolic representation for pitch classes. (More on this type shortly)
- `data Octave = Int`
  Octaves are integers from -1 to 9.
- `type Pitch = (PitchClass, Octave)`
  Tuple representation for pitches. C4 = (C,4).

Duration:

- `type Dur = Rational`
  Lossless representation for durations. 1 = 1 measure in 4/4 (4 beats).
- Predefined shorthands: `wn`, `hn`, `qn`, etc. (add `d` in front for "dotted", like `dqn`)



Volume:

- `type Volume = Int`
  MIDI standard volumes from 0-127

A note about `Rational`: you can define like regular fractions, using values like 3, 2.5, and 1/4. However, when the values are printed to the screen, they show up using different notation: 3%1, 5%2, and 1%4. However, you can't use that notation to define them – it's simply how they print.

The `PitchClass` Data Type

```
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 (Show, Eq, Ord, Read, Enum, Bounded)
```

Naming scheme: f = flat, ff = double flat, s = sharp, ss = double sharp.

C is `C`, F# is `Fs`, A♭ is `Af`, and so on.

Common Problem: Comparing Pitch Classes

- Beware of comparing equality of Pitch Classes directly!
  - The derived Eq (support for comparison with ==) instance does not support enharmonic equivalence. For example:

    `C == C` ➔ True ✓

    `C == Df` ➔ False ✗

    `Es == F` ➔ False ✗

  - To safely compare equality of pitch classes outside of an octave context…

```
pcEq :: PitchClass -> PitchClass -> Bool
pcEq x y = (pcToInt x `mod` 12) == (pcToInt y `mod` 12)
```

Using `pcToInt` converts a `PitchClass` to an `Int`, but does not force it to the range [0-11]. So, `pcToInt Cf = -1`. While this makes sense when the octave of a note is known, if you want complete handling of enharmonic equivalence over *just* the pitch classes, you need to take the values modulo 12.

This is the backquote character (`), not a single quote or apostrophe ('). Backquote is usually on the same key as tilde (~).

The `Pitch` Type

```
type Pitch = (PitchClass, Octave)
```

- The `Pitch` type is a tuple of a
  - a `PitchClass`
  - an `Octave` (type synonym for `Int`).
- Convert to/from AbsPitch using:

```
pitch :: AbsPitch -> Pitch
absPitch :: Pitch -> AbsPitch
```

- **Beware of using regular comparison operators!!!** (==, <, >, etc.)
  - Always convert values to `AbsPitch` before comparing.

Common Problem: Comparing Pitches

- The enharmonic equivalence issue with `PitchClass` still holds for `Pitch`. For example:

  `(C,4) == (Dff,4)` ➔ False ✗

  `(Es,4) == (F,4)` ➔ False ✗

- Tuples compare ***left to right*** in Haskell. So, the `PitchClass` components are are compared *before* the `Octave` components are examined. For example:

  `(C,4) < (G,3)` ➔ True ✗ (C4 is a higher pitch than G3)

  `(B,3) < (C,4)` ➔ False ✗ (B3 is a lower pitch than C4)

  `(D,4) > (C,5)` ➔ True ✗ (D4 is a lower pitch than C5)

How To Safely Compare Pitches

- Solution: always convert Pitch values to AbsPitch *before* doing comparison operations. For example:

```
absPitch (C,4) == absPitch(Dff,4)
absPitch (B,3) < absPitch(C,4)
```

➔ True ✓
➔ False ✓

Haskell note: infix operators like `(==)` and `(<)` are always applied *after* regular functions in expressions containing both. In these examples, this means that `absPitch` will get applied to all of the `Pitch` values before the comparisons take place.

Coding tip: If you're ever unsure in what order things will be applied, you can always use extra parentheses just to be safe.

Music in Euterpea

- Music in Euterpea is represented a **tree**.
- Two kinds of leaf nodes: **notes** and **rests**.
  - These basic building blocks are also called **primitives**.
- Musical ways to put music values together:
  - **Sequential composition** – play one then the other
  - **Parallel composition**– start playing both at the same time (but end times may differ)

Musical Primitives

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

- Notes have a duration (`Dur`) and an `a`.
  - `a` is a type variable for **polymorphism**.
  - The `a` value in a Note will hold pitch information (and sometimes more).
  - All Notes in a music value must have the same type for `a`!
  - We'll look at what types `a` can be later in this tutorial.
- Rests have only a duration.

Larger Musical Structures

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

sequential composition

parallel composition

for tempo changes, etc. (not covered here)

- Notes and rests (Primitives) are wrapped by a **Prim** node.
- The `(:+:)` and `(:=:)` constructors are **infix**.
  - `x :+: y` is the same as `(:+:) x y`

Functions for Building Musical Structures

- `c, cs, df, d, … , bf, b, bs :: Octave -> Dur -> Music Pitch`
  Functions for creating single notes of type Music Pitch.
  For example: `c 4 qn` ➔ `Prim (Note (1 % 4) (C,4))`
- `note :: Dur -> a :: Music a`
  Creates a Music value that is a single note.
  For example: `note qn (C,4)` ➔ `Prim (Note (1 % 4) (C,4))`
- `rest :: Dur -> Music a`
  Creates a Music value that is a single rest.
  For example: `rest qn` ➔ `Prim (Rest (1 % 4))`
- `line, chord :: [Music a] -> Music a`
  The line function combines everything in the list in sequence, while chord combines them in parallel.
  `line [x,y,z]` ➔ `x :+: y :+: z`
  `chord [x,y,z]` ➔ `x :=: y :=: z`

Common Problem: Case Matters!

- `Note` and `note` are ***not*** the same thing!!!
- `Note` and `Rest` are constructors for `Primitive`.
- `note` and `rest` are functions for creating `Music a` values.
- `C`, `Cs`, and `D` are constructors for `PitchClass`.
- `c`, `cs`, and `d` are functions for creating `Music Pitch` values.

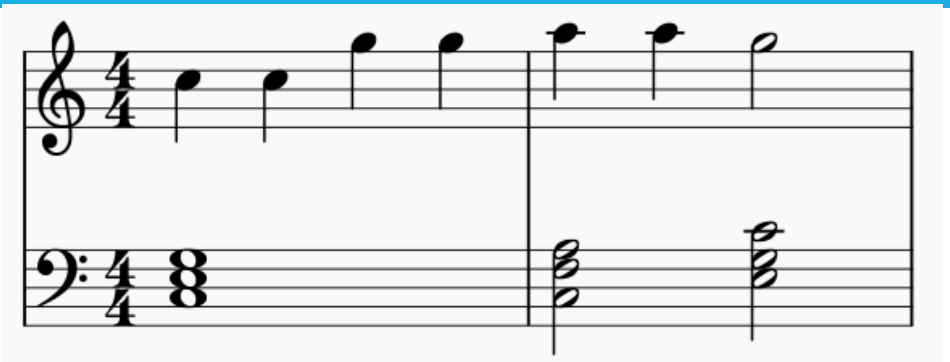
Making Some Music: Music Pitch

```
import Euterpea

melody :: Music Pitch
melody = line [c 5 qn, c 5 qn, g 5 qn,
    g 5 qn, a 5 qn, a 5 qn, g 5 hn]

chords :: Music Pitch
chords =
    chord [c 3 wn, e 3 wn, g 3 wn] :+:
    chord [c 3 hn, f 3 hn, a 3 hn] :+:
    chord [e 3 hn, g 3 hn, c 4 hn]

twinkle :: Music Pitch
twinkle = melody :=: chords
```



Try typing this into a file called Twinkle.hs and loading it in GHCi.

Play the final value:

```
play twinkle
```

Write a MIDI file:

```
writeMidi "twinkle.mid" twinkle
```

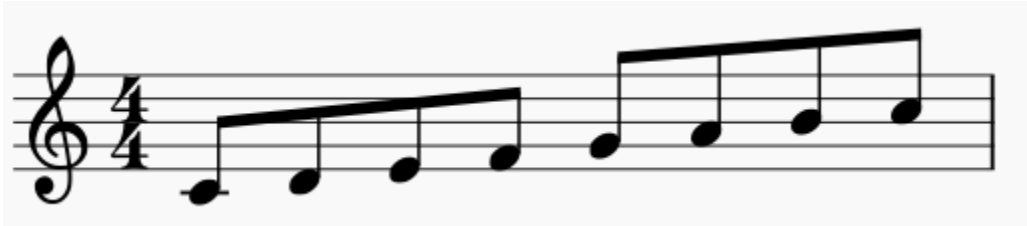
Making Some Music: `Music AbsPitch`

```
ps :: [AbsPitch]
ps = [60,62,64,65,67,69,71,72]

majScale :: Music AbsPitch
majScale = line (map (note en) ps)

vols :: [Volume]
vols = [40,50,60,70,80,90,100,110]

majScale2 :: Music (AbsPitch, Volume)
majScale2 = line (map (note en) (zip ps vols))
```



Some Playable Instances for `Music a`

- `Music Pitch`
  Ex: `c 4 qn` **or** `note qn (C,4 :: Octave)`

- `Music (Pitch, Volume)`
  Ex: `note qn ((C,4 :: Octave), 100 :: Volume)`

- `Music AbsPitch` introduced in Euterpea 2.0.0
  Ex: `note qn (60 :: AbsPitch)`

- `Music(AbsPitch, Volume)` introduced in Euterpea 2.0.5
  Ex: `note qn (60 :: AbsPitch, 100 :: Volume)`

Common Problem: Type Inference Failure

Beware of using integers without specifying their types!

Haskell has many number types. The compiler won't know that a number like 60 is an `Int` unless it can infer it from context. If there is no such context, this can happen in GHCi:

```
Prelude Euterpea> x = note qn 60
Prelude Euterpea> play x
<interactive>:4:1: error:
    * Ambiguous type variable `a0' arising from a use of `play' …
```

If you're not sure what type a value is, you can check in GHCi.

Use `:t` (or `:i`) on the value in GHCi to see its type.

```
Prelude Euterpea> :t x
x :: Num a => Music a
Prelude Euterpea> y = note qn (60 :: AbsPitch)
Prelude Euterpea> :t y
y :: Music AbsPitch
```

Common Problem: Mixing Types

- A particular `Music a` tree ***must*** have the same kind of data in all the leaf nodes. Every `Note` must have the same `a`.
- This will ***NOT*** work: `c 4 qn :+: note qn 60` ✗
- If you need to convert between Music types, you can use `mMap` to operate on the `a` part of the notes to change its type. For example:

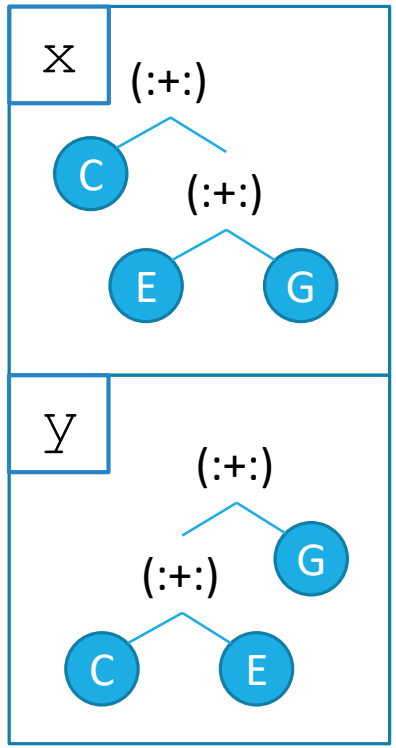
```
x :: Music Pitch
y :: Music AbsPitch
…
ok1 :: Music Pitch
ok1 = x :+: mMap pitch y   ✓

ok2 :: Music AbsPitch
ok2 = mMap absPitch x :+: y   ✓
```

Common Problem: Musical Equality

- Structurally different Music values can sound exactly the same.
- For example, in GHCi:

```
Prelude Euterpea> x = c 4 qn :+: (e 4 qn :+: g 4 qn)
Prelude Euterpea> y = (c 4 qn :+: e 4 qn) :+: g 4 qn
Prelude Euterpea> x == y
False
```



- If you want to compare performance equality, or how it will sound, then use the `perform` function on each value first.

```
Prelude Euterpea> perform x == perform y
True
```

The `perform` function is used as part of the conversion to MIDI messages and produces an event-style representation rather than a tree.

More Examples and Information

- More examples:
  euterpea.com/examples/

- Euterpea API and quick references:
  euterpea.com/api/

- Other Tutorials
  euterpea.com/tutorials