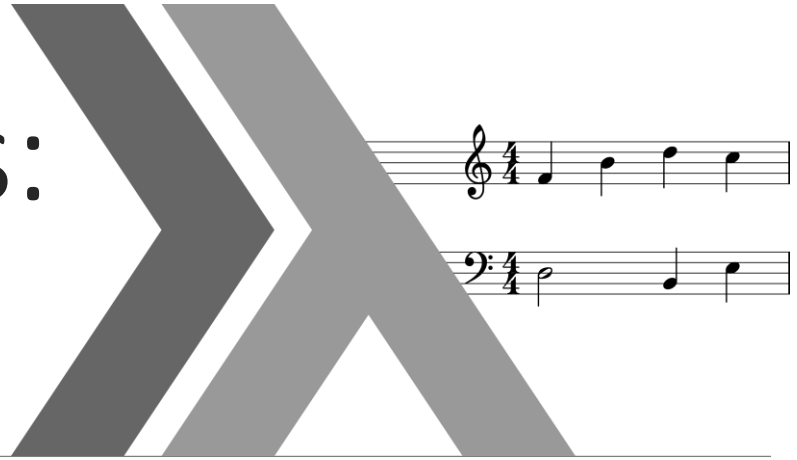
Euterpea's Music Types: Phrase Modifiers



DONYA QUICK

http://www.euterpea.com

Prerequisites

- Have Haskell and Euterpea installed.
- Be familiar with Euterpea's `Music` types.
  - Pitches, notes, rests, sequential and parallel composition, etc.
- Be familiar wth Euterpea's `Modify` constructor and the `Control` data type.
  - This constructor and datatype are for affecting performance without altering the underlying musical tree structure.
- Tutorials on these prerequisites are available here: http://www.euterpea.com/tutorials

In This Tutorial

- The `Phrase` constructor for the Control data type.
- Using phrase modifiers to affect performance of the music by adding dynamics and tempo changes.

The `Control` Data Type

```
data Control =
     Tempo       Rational
   | Transpose   AbsPitch
   | Instrument  InstrumentName
   | Phrase      [PhraseAttribute]
   | KeySig      PitchClass Mode
   | Custom      String
  deriving (Show, Eq, Ord)


phrase pas m =
    Modify (Phrase pas) m
```

We will focus on the **Phrase constructor** in this tutorial. `Phrase` is one of the constructors from the `Control` data type.

Within Euterpea, `Phrase` is primarily used to define gradual tempo and volume changes. Many of the other options that exist do not affect the playback within Euterpea, and are intended for use in interfacing with other systems (like score rendering software).

The `phrase` function (lowercase “p”) is a shorthand for using the `Phrase` constructor.

The `PhraseAttribute` Data Type

```
data PhraseAttribute =
     Dyn Dynamic |
     Tmp Tempo |
     Art Articulation |
     Orn Ornament
```

A `Phrase` takes a list of `PhraseAttribute` values. There are four categories: `Dyn` for dynamics, `Tmp` for tempo changes, `Art` for articulations (like stacatto), and `Orn` for ornaments (like trills).

Ornaments are not implemented in Euterpea's default playback algorithm. The same is true of many of the articulation options.

The `Dynamic` Data Type

```
data Dynamic
  = Accent Rational
  | Crescendo Rational
  | Diminuendo Rational
  | StdLoudness StdLoudness
  | Loudness Rational

data StdLoudness =
    PPP | PP | P  | MP |
    SF  | MF | NF | FF |
    FFF
```

Only the first three constructors of the `Dynamic` data type (in bold) cause audible changes with Euterpea's default MIDI playback implementation. Arguments to all three should be ≥0.

An **accent** indicates a volume increase. A value of `Accent 1.5` means to be 50% louder.

Crescendo means to increase the volume over time. A value of `Crescendo 1.0` means to linearly increase the volume by 100% over time. The first note's volume will be unchanged, the next will be louder, etc.

Diminuendo (or decrescendo) means to reduce the volume over time. `Diminuendo 100` means to linearly decrease the volume by 100% over time (trending towards zero by the end).

The `Tempo` Data Type

```
data Tempo =
    Ritardando Rational
  | Accelerando Rational
```

Be careful with types! The Tempo data type is **not** the same as the Tempo constructor of Control.

```
data Control = Tempo Rational | ...
```

The `Tempo` data type is to define increases and decreases in tempo over time. These impact the onset and length of note events in the intermediate `MEvent` representation that Music values pass through on the way to becoming MIDI events. **These constructors do not insert tempo change MIDI events and instead alter the onsets of the MIDI events.**

Arguments to both constructors should be ≥0.0.
A value of `Ritardando 0.5` will slow the music more than a value of `Ritardando 0.2`. Similarly, a value of `Accelerando 0.2` will speed up the music more than a value of `Accelerando 0.1`. Use values >1.0 with caution on values that contain more than just a few notes

The `Articulation` Data Type

```
data Articulation
   = Staccato Rational
   | Legato Rational
   | Slurred Rational
   | Tenuto
   | Marcato
   | Pedal
   | Fermata
   | ...
```

Only the first three `Articulation` constructors affect Euterpea's default MIDI playback algorithm. **These constructors do not insert tempo change MIDI events and instead alter the onsets of the MIDI events.**

`Staccato` and `Legato` actually do the same thing, and the two constructors only exist for semantic clarity. Arguments <1.0 will shorten the length of the notes and >1.0 will lengthen the notes. For example, a value of `Staccato 0.5` will make all of the notes 50% as long as their original durations. `Slurred` is like `Legato`, but does not operate on the last note(s).

Note onset is preserved in all cases, and arguments should always be >0.

The `Ornament` Data Type

```
data Ornament =
    Trill
  | Mordent
  | InvMordent
  | DoubleMordent
  | Turn
  | TrilledTurn
  | ShortTrill
  | Arpeggio
  | ...
```

The `Ornament` type exists purely to permit more advanced, user-defined playback algorithms and exporters to alternative file formats.

None of the constructors for this data type will affect Euterpea's default playback algorithm. To make use of these, you would need to make a custom playback algorithm. You can do this via the playC function and PlayParams datatype, but those features are beyond the scope of this tutorial.

Some examples

Try the following examples to observe the effects of the phrase modifiers.

```
m0, m1, m2 :: Music (AbsPitch, Volume)
m0 = times 10 (note qn (60, 50))
m1 = phrase [Art $ Staccato 0.2] m0
m2 = phrase [Tmp $ Accelerando 0.5, Dyn $ Crescendo 1.5] m0

m3, m4 :: Music Pitch
m3 = instrument Flute $
     times 5 (c 4 en :+: rest en :+: g 4 qn)
m4 = phrase [Art $ Legato 2.0] m3
```

Haskell syntax: `f $ g x` is equivalent to `f (g x)`.

Common Problem: Not Setting Volumes

- The default volume when using the `Music Pitch` and `Music AbsPitch` types is 127 – there is no room to get louder!
- Trying to crescendo or accent these values will have **NO EFFECT**.
- Before you use volume-related phrase modifiers, set a volume that is somewhere in the middle of the 0-127 range.
  - Use the `addVolume` function for `Music Pitch` values to make them into `Music (Pitch,Volume)`
  - Do an `mMap` on `Music AbsPitch` to make them into `Music (AbsPitch, Volume)`. For example:

```
mMap (\p -> (p,50::Volume)) someMusic
```

Common Problem: Bad Volume Values

- Currently, Euterpea does not check for volumes outside the 0-127 range. Use volume-related phrase modifiers cautiously!
- Bad volume values can either manifest as silence or end up overflowing elsewhere into the acceptable range.
- If you hear weird stuff, call `perform` on your music value and examine the volumes of the `MEvents`.
- There are two main ways around this:
  - Write a wrapper for the default perform function algorithm that caps the volumes and then use the `playC` function.
  - Use the performance framework from the HSoM library (see the Haskell School of Music textbook).

Common Problem: Bad Volume Values

Here's a way to implement the first solution: wrapping the default performance algorithm.

```
perform' :: (ToMusic1 a) => Music a ->
Performance
perform' = map volFun . perform where
  volFun mev =
    let v = eVol mev
        v' = min 127 (max 0 v)
    in  mev{eVol = v'}

play' :: (ToMusic1 a, NFData a) =>
    Music a -> IO ()
play' = playC defParams{perfAlg = perform'}
```

Example of value with bad volumes:

```
badVols = phrase [Dyn $ Crescendo 3.0] m0
```

You can see that the volumes in `badVols` above are above 127 by examining the value produced by the following:

```
perform badVols
```

and you can verify that they are fixed with perform' by examining this value:

```
perform' badVols
```

Compare the results of playback in GHCi this way from GHCi:

```
play badVols
play' badVols
```

More Examples and Information

- More examples:
  euterpea.com/examples/

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

- Other Tutorials
  euterpea.com/tutorials