EuterpeaScala

EuterpeaScala is a concise music notation for fast and easy transcription and construction of music, with a very uniform syntax, while allowing a rich set of musical capability and performance, and the capability to construct all aspects of the music programmatically.

Contents

[Introduction and Scope 2](#_Toc479090965)

[Notation Overview and Simple Examples 4](#_Toc479090966)

[Constructing Music from the Simplest Components 4](#_Toc479090967)

[Example 1 4](#_Toc479090968)

[Example 2 5](#_Toc479090969)

[Modifiers and Music 5](#_Toc479090970)

[Modifiers 5](#_Toc479090971)

[Repetition 5](#_Toc479090972)

[Melody, Notes, Rests and Drums 6](#_Toc479090973)

[Note Pitch 6](#_Toc479090974)

[Octave 6](#_Toc479090975)

[Note Range Limitation 6](#_Toc479090976)

[Rests and Drums 6](#_Toc479090977)

[Lyrics 7](#_Toc479090978)

[Beats, Tempo and Time Signatures 7](#_Toc479090979)

[Beats and Durations 7](#_Toc479090980)

[Beat Combinations and Scaling 7](#_Toc479090981)

[Time Signatures 8](#_Toc479090982)

[Tempo 8](#_Toc479090983)

[Bars and Structure 8](#_Toc479090984)

[Tracks, Channels, Instruments 9](#_Toc479090985)

[Keys, Harmony, Chords and Transposition 10](#_Toc479090986)

[Keys 10](#_Toc479090987)

[Chords 10](#_Toc479090988)

[Harmony Transformations 11](#_Toc479090989)

[Chords on the Key Tonality 11](#_Toc479090990)

[Broken Chords and Arpeggios 12](#_Toc479090991)

[Thansposition 12](#_Toc479090992)

[Note Volume, Width and Dynamics 13](#_Toc479090993)

[Volume 13](#_Toc479090994)

[Note Width 14](#_Toc479090995)

[Dynamics Patterns 14](#_Toc479090996)

[Rhythm Patterns 15](#_Toc479090997)

[Ornaments 16](#_Toc479090998)

[Continuous Controls 17](#_Toc479090999)

[Controls 17](#_Toc479091000)

[Pitch Bend 17](#_Toc479091001)

[Checking, Saving and Playing Music 18](#_Toc479091002)

[Reference 19](#_Toc479091003)

[Music 19](#_Toc479091004)

[Operators 19](#_Toc479091005)

[Modifiers 19](#_Toc479091006)

# 

# Introduction and Scope

EuterpeaScala is a “Domain-Specific Language” (DSL) for music built on top of the Scala programming language. It is a project very loosely modelled on the Yale Haskell Euterpea[[1]](#footnote-1) project (Haskell School of Music - HSoM), from where I have stolen the name. While it is certainly not a port of the Yale Euterpea from the Haskell programming language to Scala, it aims to achieve similar capability (and perhaps more) for Midi music only.

Unlike the HSoM project, which is aimed at performance of notated or generated music, the emphasis in EuterpeaScala is on composition and structure. So, notions of bars and phrases are retained throughout, including across tempo and time signature changes. Similarly, the structures of melody, rhythm, harmony and dynamics can all be separately described and combined into a piece of music.

It supports:

* multiple tracks, multiple instruments, multiple channels
* both melodic and drum instruments, with a full range of note pitches and beat lengths
* lyrics
* arbitrary changes of tempo, time signature, key signature and modulation of tonality
* clear identification of bar boundaries which aids validation of correct transcription
* construction of chords and harmonies, including modes of the current tonality, and including broken chords and arpeggios
* beat scaling for complex rhythms and cross-rhythms
* dynamics on volume and note width
* optionally independent specification of rhythm and note sequence, and re-use of either
* transposition - both chromatic and diatonic
* patterns for note dynamics (volume, width and timing) for phrase-based control of stress, swing, humanisation, etc
* note ornaments (turns, mordants, trills etc) within the current tonality
* control envelopes for continuous Midi controls and pitch bending
* range limitation of all notes with octave shifts to keep notes within an instrument's range

One important capability that it shares with the HSoM Euterpea is that the music is written as a ***program*** in a host programming language (in this case Scala), and running that program generates the resultant Midi messages to be played or saved as a Midi file. This has two significant implications:

* The music notation remains a well-formed Scala program. Scala is a good language for hosting embedded Domain-Specific Language, and the DSL need not look like any normal Scala program (if there is such a thing). But global constraints of the programming language (e.g. on layout, punctuation, comments, syntax, format of tokens, etc) cannot be avoided and mal-formed music notation will result in program text that will not compile as Scala and so will not work. The music may need to be “debugged” just like any program,
* The musical structures can be constructed by the full power of a Scala program. This construction includes:
  + easy re-use of musical components,
  + parameterizable components that can be used differently in different contexts,
  + programmatic generation of the music from any sort of input, data or randomly, etc.

# Notation Overview and Simple Examples

At its simplest, the music consists of constructions of two fundamental classes, arranged by the notation into an “Abstract Syntax Tree” (AST). It is this AST that will be processed either to save the music in Midi File Format, or to play the music directly.

## Constructing Music from the Simplest Components

The two fundamental classes are **Music** and **Modifier**. And there are several different sub-classes each of Music and Modifier, which can be combined with a limited number of simple operators to make a single Music object that represents the entire piece.

To give a flavour of how music is constructed, we will construct a couple of small examples and we will limit ourselves to using just five elements of the notation

* Individual notes of music
* The “-“ operator to construct music in sequence
* The “Beat” modifier to control timing
* Repetition
* The “&“ operator to construct music to play at the same time

Later, we will expand on the capabilities of each of these, together with other sub-classes of Music and Modifier and other operators to combine them

## Example 1

val tune1 = (E - D - C/2) \* 2 - (G - (F - F)/8 - E/2) \* 2

This results in the music:



I.e. two lines of “Three blind mice” in the key of C

Points to note are:

* This is a normal Scala “val” (value) definition of a value named “tune1”. It is “proper” Scala, but the objects and operators from which it is constructed are part of the EuterpeaScala “language”, that this document will describe and explain. The value tune1 has type Music.
* Individual Notes are a sub-class of Music and are indicated by capital letters from A to G
* The sequence comes from the “-“ operator between pieces of music, such as notes or other grouping. Music before “-“ plays first, followed by the Music after.
* There are a couple of common Modifiers used here, introduced with the “/” operator. These are “Beat” modifiers: “/2” specifying that the note plays as a Half-note beat and “/8” specifying that the note plays as an Eighth-note beat – the default beat is a Quarter note
* Grouped pieces of music can be surrounded with parentheses – this is normal Scala syntax for expressions. In this case, the pair of F notes both get the Eighth beat Modifier.
* Pieces of music can be repeated a number of times, indicated by the “\* 2”
* The spacing used in this example is not strictly necessary, but has been found to aid readability and avoid confusion when there are several modifiers or operators

## Example 2

val tune2 = {

val topLine = (G \* 12 - B \* 6)/8 - +C/2/Dot

val bottomLine = (F \* 6 - E \* 6 - D \* 6)/8 - C/2/Dot

TimeSig(6,8) /: (topLine & bottomLine)

}

This results in the music:



I.e. “Chopsticks”

Further points to note here are:

* We are using the Scala programming language to define the “top” line and “bottom” line separately. The curly braces enclose a block defining the multi-line structure of the tune, and the last line of the block is the resultant value of tune2.
* The “&” operator combines the two lines of music, just like “-“, but they play at the same time, rather than sequentially
* The final C on the top line is an octave higher, indicated by the leading “+”
* The “Dot” modifier dots the beat, extending its duration by 50%
* The time signature of Chopsticks is not the default 4/4. The use of the TimeSig Modifier will be described later, but its meaning should be clear

# Modifiers and Music

## Modifiers

In EuterpeaScala music, there will be many cases where a piece of Music is altered by a Modifier. Universally any Modifier can be applied to any Music in one of two ways, using one of two operators.

* It can be represented as Music / Modifier. This is how the Beat and the Dot Modifiers are used in the examples above.
* It can be represented as Modifier /: Music. This is how the TimeSig Modifier is used in the example above.

Choice of which operator to use is down to personal preference, though any one Music object can only use a single operator for all its modifiers – they can only be mixed using parentheses. In general it feels more natural to use the “/” operator (and with no spaces) for small local Modifiers, and the “/:” operator (with spaces around it) for Modifiers with wider scope.

In addition, a value defined as multiple Modifiers (joined by the “/” or “/:” operators) can be used as a Modifier. This allows a group of related Modifiers to be defined as a value and used multiple times within a piece of Music.

## Repetition

The “\*” operator repeats Music a number of times, as in the earlier example with “\* 2” to repeat a phrase twice.

# Melody, Notes, Rests and Drums

## Note Pitch

In music, a melody line is a sequence of notes, with fixed or varying beat and which may be interspersed with rests. As illustrated by the examples above, the “-“ operator is the fundamental connector for two pieces of music that constructs them in a sequence. Either or both of the two pieces may themselves be sequences. But at the lowest level there will be notes and rests.

Notes have a pitch represented by the letters A to G. On their own, the letters correspond to the seven white notes starting at Middle C (also known as C4). But there are many ways in which the pitch can be altered.

* A Note can be sharpened or flattened, denoted by one or two letters “s” or “f” – e.g. Fs (F-sharp) or Bf (B-flat). Where appropriate, “ss” and “ff” are double sharp and double flat.
* When a key signature is applied as a Modifier to music, Notes which are sharpened or flattened in that key signature will be automatically sharpened or flattened
* To avoid a Note being sharpened or flattened by the key signature, it can be specified as “natural” denoted by the letter “n” – e.g. Fn (F-natural)

So, the Notes available in the middle octave are then:

* Cf, C, Cn, Cs, Css, Dff, Df, D, Dn, Ds, Dss, Eff, Ef, E, En, Es, Ff, F, Fn, Fs, Fss, Gff, Gf, G, Gn, Gs, Gss, Aff, Af, A, An, As, Ass, Bff, Bf, B, Bn, Bs

The duration of a Note is given by the current Beat. Beat Modifiers are described in the next section, and the previous examples show simple uses.

## Octave

The pitch of each Note defaults to the current octave, but there are three ways in which the octave of Notes can be changed:

* Any Note can be specified as being in a specific fixed octave by following the note letter by the octave number in parentheses – e.g. C(3) is the C below Middle C
* A Note can be specified as being one octave higher or lower by preceding it with the operator “+” or “-“ (note that this is a different use of “-“ from joining music sequentially)
* Any piece of music can be specified as being transposed one or more octaves higher or lower by the Modifier “Octave(n)”, where “n” is the (positive or negative) number of octaves.

## Note Range Limitation

As Music is modified (e.g. by harmonies as described later), the pitches of a Note may become outside the suitable range for a voice or instrument. The Range Modifier constrains Music to be within the range specified by specified low and high notes – E.g.

Range(A(3),D(5))

Notes will be transposed up and down the minimum number of octaves to bring the pitch within range.

## Rests and Drums

As well as Notes as the lowest class of Music, there are also:

* Rests – represented by the single word “Rest”. These also have duration given by the current Beat and can have their own Beat Modifier – e.g. Rest/2 – a Half-note rest
* Drum notes, which have no pitch and sound on a dedicated Drums channel (described later). An example of a single Drum note is Drum(Drum.Bass\_Drum\_1). This is a single hit of the bass drum at the current beat. For conciseness and readability, it is often useful to declare Drum notes as local values:

val bd = Drum(Drum.Bass\_Drum\_1)

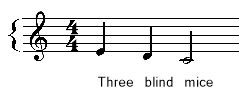
val sn = Drum(Drum.Snare\_Drum\_1)

val drumPart = (bd – Rest – sn – Rest) \* 4

## Lyrics

Music (and particularly Notes) can have associated lyrics. The Lyric modifier is simply a string – e.g.:

val threeBlindMice = E/"Three" - D/"blind" - C/2/"mice"



Alternatively, the Lyrics Modifier takes a string of multiple lyric words and syllables and applies them sequentially to the notes of the modified Music. So an alternative way of generating the same music with lyrics would be:

val threeBlindMice = (E – D – C/2)/Lyrics("Three blind mice")

Within the Lyrics string:

* A dash “-“ can be used within a word to split it into multiple syllables, each with its own Note
* Dashes can be added at the end of a word or syllable to allow it to be sung acrosss multiple Notes
* An underscore “\_” within a word will be treated as a space ***within a word*** when multiple unstressed words can be sung as a single Note

It will be detected and reported as an error if the number of Notes modified does not match that needed for the Lyrics string.

For example, the Georgean song “Shen Khar Venaki” has long slow flowing vowels:

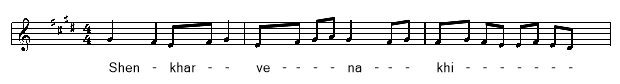
val melody1 = G/4 - F/4 - E - F - G/4 |   
E - F - G - A - G/4 - F - G |   
F - G - F - E - E - F - E - D

val lyrics1 = Lyrics("Shen- khar-- ve--- na-- khi-------")

val line1 = melody1/lyrics1

EMaj /: Eighth /: Tempo(60) /: tune

is:



# Beats, Tempo and Time Signatures

## Beats and Durations

Music comprises sequences of notes of different durations – different “beats”. In EuterpeaScala music, these different beats are handled by the “Beat” sub-class of Modifier.

In practice, the Beat class is never used directly. Instead there are several defined Beat objects, which correspond to normal notated music:

* Whole
* Half
* Quarter
* Eighth
* Sixteenth
* ThirtySecond

So, music can be written as:

val threeBlindMice = E - D - C/Half

val seeHowTheyRun = G - (F - F)/Eighth - E/Half

As a convenient notation, an integer value can be used as a modifier and if it is a power of two (1, 2, 4, 8, 16, 32) it will be the corresponding Beat Modifier. So, the following is just the same:

val threeBlindMice = E - D - C/ 2

val seeHowTheyRun = G - (F - F)/8 - E/2

## Beat Combinations and Scaling

Not all note durations are precisely these. Beats can be added together, though to avoid confusion the integer notation cannot be used. But (Half + Eighth) is a valid Beat value.

Beats can be scaled by the BeatScale Modifier sub-class. Four BeatScale Modifier objects are defined to cover common cases:

* Dot
* DotDot
* 3
* 5

The Dot Modifier extends the current Beat duration by half as much again, making a dotted note. The DotDot Modifier extends the current Beat by half as much plus quarter as much again, making it I¾ length.

The “/3” and “/5” Modifiers scale the beat by 2/3 and 4/5 respectively – forming triplets and quintuplets.

And for complex cross-rhythms, any beat scaling can be defined as BeatScale(numberOfNotes,numberOfBeats). So “/Beatscale(7,5)” could allow a cross-rhythm of 5 against 7, which would be easier to write than to play!

## Time Signatures

Music has a time signature, which imposes a “bar structure” onto the sequence of notes. The use of bars is described later, but the time signature of the music (or any part of the music) can be specified by the TimeSig Modifier. This takes the form “TimeSig(numberOfBeats,beat)” The Beat is an object of class Beat (i.e. Quarter, Eighth, etc.) but the integer abbreviations can (and often will) be used.

So, TimeSig(4,4) is the default 4/4 time, TimeSig(6,8) is 6/8 (jig) time TimeSig(3,2) is slow waltz 3/2, etc.

## Tempo

Music has a Tempo, which gives each beat a defined amount of time to play. The default tempo is 120 quarter-note beats per minute (BPM). The Tempo Modifier allows a different number of beats per minute to be specified for the Music it modifies. This Modifier takes one of two forms:

* Tempo(nnn)  
  this means that the Music will be at the constant “nnn” BPM, where the beat is that of the time signature (Quarter, Eighth, etc)
* Tempo(startNnn,endNnn)  
  this means that the tempo of the Music will gradually change over its duration from startNnn BPM to endNnn BPM, causing an acceleration (*accelerando*) or slow down (*ritardando*).

Note that these tempo changes affect the performance of the music, but do not in any way change its structure – there are still the same number of beats in each bar.

# Bars and Structure

## Bars

Given a known time signature, it is unambiguous in the music where the bars are. And understanding the bar structure is of great assistance in understanding the music.

To help with this, EuterpeaScala has the operator “|” which acts identically to “-“ ***except that*** the program reports an error if the position of the operator is not precisely on a bar boundary of the current time signature. Its sole point is to aid readability of music to be able to understand a sequence of notes as a sequence of bars.

So, using the two definitions from earlier, Three Blind Mice can be written:

threeBlindMice | threeBlindMice | seeHowTheyRun | seeHowTheyRun

And it would be an error if the duration of either threeBlindMice or seeHowTheyRun was not an exact number of bars.

To handle the case where a tied note straddles a bar boundary, there is a variant of “|” which is “+|+beat” where the beat is a tied addition at the start of the new bar to extend the duration of the last Note of the previous bar. For example:

C - D - E - G +|+4 - E - D - C +|+1

results in:



The checks that bar operators appear precisely on a bar boundary also apply to time signature and key signature changes. It is an error for a time signature or key signature to occur other than at the start of a bar. And the music modified by a TimeSig or KeySig Modifier must also be an exact number of bars (in that time signature).

## Lines and pickups

Many pieces of music can be formed of lines which do not precisely align with bars. Often there can be one or two unstressed notes in a line which precede the first bar as a “pickup” or “anacrusis”.

In order that the structure of the lines can be preserved, while also maintaining the structure of bars, EuterpeaScala has the Pickup Modifier. The Music modified by Pickup appears ***immediately before*** the current position in the constructed music, and does not follow and extend the music duration.

# Tracks, Channels, Instruments

The General Midi standard defines 128 distinct instrument names that a Midi device can play. A piece of music can comprise up to 16 channels, any or all of which can play simultaneously – on the same or (usually) different instruments. In addition, a Midi file can organize the music into a number of tracks that can be muted independently, and usually edited independently by sequencer software.

EuterpeaScala has Modifiers to control each of these three concepts.

* Instrument(name) modifies the Music to play on the named instrument. The names are the names of instruments as defined in the General Midi standard.
* Instrument(id) modifies the Music to play on the identified instrument, where the id is a value of the form “Instruments.Acoustic\_Grand\_Piano” or “Instruments.Xylophone”. The identifiers are derived from the names in General Midi standard, but with spaces and other punctuation replaced with underscores to make them into valid Scala identifiers.
* Channel(name) modifies the Music to play on the named channel. All Music with the Channel Modifier for the same channel name will play on the same Midi channel. So, there can be a maximum of 16 distinct channel names. The names themselves have no meaning outside EuterpeaScala. The channel name “Drums” is predefined as Midi channel 10, which the General Midi specification reserves for drums and untuned percussion.
* Track(name) places the music on the named track. All Music with the Track Modifier for the same track name will be added to a saved Midi file on the same track, which will have that name. There are no limits to the number of distinct tracks in a Midi file.

Tracks will usually all be on the same Channel, though there is no requirement that this is the case. However, a channel may have its notes spread across multiple tracks – e.g. for separate piano left and right hand tracks, or for different Drum sounds on separate tracks.

# Keys, Harmony, Chords and Transposition

## Keys

Music will generally be written in a specific key, which defines the “tonality” and in particular the “root” (or “home” or “tonic”) note.

EuterpeaScala has the KeySig Modifier to specify that all or part of a piece of music plays in a particular key. In practice, the KeySig class is never used directly. Instead there are several defined KeySig objects, which correspond to the 24 available key signatures (with a few enharmonics):

* CMaj, GMaj, DMaj, AMaj, EMaj, BMaj, FsMaj, CsMaj, FMaj, BfMaj, EfMaj, AfMaj, DfMaj, GfMaj, CfMaj, FfMaj
* AMin, EMin, BMin, FsMin, CsMin, DMin, GMin, CMin, FMin, BfMin, EfMin, AfMin, DfMin

The default key if none is specified is CMaj, which has no sharps or flats.

The effect of a key signature is that:

* In a Midi file, the key signature will normally affect how the music is printed by sequencing software.
* The Notes C, D, E, F, G, A, B will correspond to the sharpened or flattened Note that the key signature implies. So for example, in GMaj, “F” is the same as “Fs” (F-sharp). To get an F-natural when the key is GMaj, it is necessary to write “Fn”
* The current key can provide the intervals for some chords, as described later

## Chords

EuterpeaScala has explicit support for Chords to allow the harmonic structure of music to be described separately from melody and rhythm as required. While it is certainly possible to construct a chord using the underlying notes explicitly, this is not generally the best way. So it is not recommended to write a chord like:

val dodgyCMin = C & Ef & G

Chords are described in EuterpeaScala using a root Note and a Harmony modifier. And the root note can be explicit or can be based on the current key or tonality.

A Harmony Modifier object is a set of intervals from the root note, counting in semitones. A rich set of common harmonies are provided, and they in turn can be further transformed to add and remove intervals. So it is rare to explicitly use the Harmony Modifier class directly. While it would also be possible to write:

val dodgyCMin2 = C/Harmony(0,3,7)

in practice, this would be used as C/Min.

The common Harmony objects that are defined to apply to a Note are:

* Maj – a major triad
* Min – a minor triad
* Dom7 – the dominant 7th, with a major third and minor 7th
* Maj7 – the major 7th, with a major third and major 7th
* Min7 – the minor 7th, with a minor third and minor 7th
* Triad – a triad of a third and a fifth ***in the current tonic scale***

## Harmony Transformations

From these starting harmonies, further transformations to the Harmony can be added using the notation “.xxx”, where “.xxx” is one of:

* .inv1 – the first inversion with the third as the lowest note
* .inv2 – the second inversion with the fifth as the lowest note
* .inv3 – the third inversion with the seventh as the lowest note
* .b – the first inversion with the third as the lowest note
* .c – the second inversion with the fifth as the lowest note
* .d – the third inversion with the seventh as the lowest note
* .aug – replacing the fifth with an augmented fifth
* .dim – replacing the fifth with an diminished fifth
* .sus – replacing the third with a (suspended) fourth
* .sus2 – replacing the third with a (suspended) second
* .add6 – adding a major sixth
* .min7– adding a minor seventh
* .maj7 – adding a major seventh
* .add8 – adding an octave
* .add9 – adding a ninth
* .add11– adding an eleventh
* .add13 – adding a thirteenth
* .flat – transpose all notes down a semitone
* .sharp – transpose all notes up a semitone

A simple example is:

A/Min - G/Maj - F/Maj - E/Maj



## Chords on the Key Tonality

Another way in which a Chord can be formed is based on a harmonic mode of the current key tonic note. These Chord positions are conventionally written using Roman numerals and in EuterpeaScala, these Roman numerals are always upper case letters.

There are fourteen tonic-relative Chord position objects:

* I, II, III, IV, V, VI, VII
* I7, II7, III7, IV7, V7, VI7, VII7

The root of these triads is the one of the seven chord positions ***in the current key or tonic***. The optional “7” for each chord adds the seventh – again in the current scale. And these Chord objects can also have the same transforms listed above for a Harmony.

A simple example is:

VI7 - II7.b - V7 – I



For any chord value “ch”, the value of “c.hroot” is the note at its root. This is particularly useful when used with the tonic-relative Chord position objects. It can be used for example to extract a bass line from a chord sequence.

The Root Modifier extracts the root notes from ***all*** Chords in the modified Music, so could for example extract the base line from a Chord sequence.

## Broken Chords and Arpeggios

Chords need not sound their notes totally simultaneously. They can be Broken or Arpeggios.

A Broken chord sounds its notes simultaneously, but with different start times so that the attack of each Note in the Chord is heard independently. The higher notes are slightly delayed.

The Broken Modifier can be applied to any Music and affects the playing of all Chord within that Music. For each Chord it specifies the proportion of the current beat that each higher note is delayed by. So, to separate the notes of the first three Chords by a 20th of the beat duration:

(VI7 - II7.b - V7)/Broken(0.05) - I

An Arpeggio plays the Notes of the Chord one or more at a time at a fast beat in a pattern within the duration of the Chord. The last arpeggiated note of the pattern is held for any remaining part of the total Chord duration. So, in the example:

val pattern = Arpeggio(Thirtysecond, 1,2,3,4,3,1)

(VI7 - II7.b - V7)/pattern - I

Each note in the chord plays for a Thirtysecond (i.e. eight notes within each quarter beat) and with the specified pattern, resulting in:



The values in the pattern can be single integers (as above) or a tuple of 2, 3 or 4 integers (in parentheses, separated by commas – standard Scala syntax). These tupled notes sound together:

val pattern = Arpeggio(Sixteenth,1,(2,3,4),(1,3),(2,3,4))

((VI7 - II7.b - V7 - I)/pattern)



## Transposition

Another aspect of harmonization is transposition. Music can be transposed up or down either:

* Chromatically by a fixed number of semitones, or
* Diatonically, by a number of scale steps in the current tonality or key

Both use the Transpose Modifier

* **Transpose(nn)**, where “nn” is a positive or negative integer, transposes the music up or down by that number of semitones
* **Transpose(pos1 -> pos2)** where pos1 and pos2 are Roman-numbered Chord positions, transpose the music diatonically within the tonality, keeping the notes within the scale.

So, for example:

Transpose(I -> II) /: (C - D - E - F - G)

transposes up a scale step in the current key of CMaj and results in



Finally, the tonality (and so the tonic or root note) can be changed for part of the music ***without*** changing the key signature.

The Modulate Modifier takes a key signature parameter and “modulates” the music into that key for the purposes of harmonization only. Individual notes will continue being interpreted within the current key signature, which is unchanged. But harmonies and transpositions using Roman numeral Chord positions will be relative to the specified Modulate key.

# Note Volume, Width and Dynamics

## Volume

Notes in a piece of music do not all play at the same volume; varying the volume adds expression (or “dynamics”) to a performance. This variation can be:

* across a sequence of Notes or Chords to play them louder or software, or
* repetitive (such as stressing the start of each bar), or
* varying over longer phrases (as *crescendo* and *diminuendo*), or
* can affect individual notes (such as a *sfotzando*)

Other dynamics that can affect the expression of the music are the note width (such as staccato or legato) and subtle changes in timing of notes. All of these can be controlled by EuterpeaScala.

EuterpeaScala has the Volume Modifier to specify that all or part of a piece of music plays at a particular volume. In practice, the Volume class is not used directly. Instead there are several defined Volume objects, which correspond to the common loudness notations in printed music:

* Vmf – Mezzo-forte
* Vmp – Mezzo-piano
* Vf – Forte
* Vff – Fortissimo
* Vfff – Multo fortissimo
* Vp– Piano
* Vpp– Pianissimo
* Vppp – Multo pianissimo

These are the Scala objects for the notations mf, mp, f, ff, fff, p, pp, ppp. The leading “V” is unfortunate, but necessary for the Scala system to distinguish “f” from the note “F”.

Each of these corresponds to a use of the Volume(nnn), where nnn is the Midi note volume (or velocity), in the range 0 to 127. The objects are ordered as

* Vppp < Vpp < Vp < Vmp < Vmf < Vf < Vff < Vfff

So, using one of these Volume objects as a Modifier causes the modified Music to play at that volume.

There are also a set of ***relative*** Volume Modifiers, which specify a music volume relative to that currently used. These are:

* Vcurrent
* Vlouder
* Vlouder2
* Vlouder3
* Vquieter
* Vquieter2
* Vquieter3

The objects are ordered as

* Vquieter3 < Vquieter2 < Vquieter < Vcurrent < Vlouder < Vlouder2 < Vlouder3

There is also a VolumeChange Modifier, which is also never used directly. Instead, a Modifier of the form “(V1 -> V2)”, where V1 and V2 are one of the above defined Volume objects, causes the volume to change smoothly from V1 to V2 across the duration of the music. For example:

(Vmp -> Vff) /: (C - D - E - G - E - D) –

(Vff -> Vmf) /: (C(5) - B - A - F - A - B – C)

is a smooth crescendo from mp to ff in the middle and then a diminuendo back down to mf.

## Note Width

The Note width is the proportion of a Note’s duration that it actually sounds. This is controlled by the Width Modifier, which takes a single parameter in the range 0.0 to 1.0.

There are four pre-defied objects of the Width Modifier class:

* DefaultWidth – which is Width(0.9), with a slight gap between notes
* Staccato – which is Width(0.3), so that notes are short and sharp
* Marcato – which is Width(0.7), with notes clearly separated
* Legato – which is Width(1.0), with notes flowing together with no gaps

## Dynamics Patterns

For more complex control of expression, dynamics and phrasing, the Dynamics Modifier is very powerful and allows rich variation of volume, note width and note timing in a repeated pattern.

Before considering the complete flexibility of the Dynamics Modifier, there are three defined standard Dynamics objects[[2]](#footnote-2) which can be used as Modifiers:

* Dynamics.pulse(beat,strength) increases the note volume by the value “strength” for every note which is precisely on a multiple of the beat (e.g. at the start of each bar)
* Dynamics.swing(beat,delayFactor), which delays the timing of notes so that every other beat specified is delayed by the specified factor of a Quarter beat duration, bending the tempo for performance, but not changing it in the structure of the music. The delay is smoothly applied between beats to add a swing to the music.
* Dynamics.delay(delayFactor), which delays the timing of the start of all notes in the music so that every note is delayed by the specified factor of a Quarter beat duration. A negative factor causes the note to play earlier. This can allow some instruments or tracks to play slightly behind or ahead of the current beat. This is also used internally in the implementation of broken Chords.

The Dynamics Modifier takes an arbitrary number of parameters, each of which is created by the object “X” and which define points in a repeated “Dynamics envelope”. Each point takes the form: X(beat, …) where the … are zero or more optional named parameters. The beat is the duration it takes for the dynamics values to change from the previous values to the ones specified by the “X”. The sum of all these beat values is the duration of repetition of the Dynamics Modifier as a whole.

The optional named parameters for each X, specify changed values at that point in the envelope and are:

* volumeInc – an integer giving the volume increment
* noteWidthInc – a number in the range 0.0 to 1.0 giving the increment in note width
* timingInc – a number in the range 0.0 to 1.0 giving the increment in delay of the note start, as a multiple of a Quarter beat duration
* timingJitter – a number in the range 0.0 to 1.0 giving the limits to random variation in delay of the note start, as a multiple of a Quarter beat duration. This can make music sound more human and less metronomic

The pattern in the Dynamics Modifier is applied repeatedly across the duration of the modified Music.

To aid understanding of the complexity of the Dynamics Modifier, the implementation of the built-in pre-defined Dynamics objects are:

def pulse(beat: Beat, strength: Int) =

Dynamics(X(NoDuration,volumeInc=strength), X(NoDuration), X(beat))

def swing(beat: Beat, delayFactor: Double) =

Dynamics(X(beat,timingInc=delayFactor), X(beat))

def delay(delayFactor: Double) =

Dynamics(X(NoDuration,timingInc=delayFactor))

A final notation to control the dynamics of music is the “slur”. Two notes separated by the “-- “ (double dash) operator instead of the usual single dash are slurred together. The first note’s width is extended to 1.0, avoiding any gap between the notes. The width of the second note is reduced by 30% of its current width and volume of the second note reduced by one incremental step.

# Rhythm Patterns

In some pieces of music, a rhythm pattern may appear many times.

* There may be multiple harmonized parts all playing different pitches or instruments, but at the same rhythm.
* A rhythmically complex section of music may be repeated with a differently pitched melody.
* Or a pattern may be used many times as a “motif”.

In these and other cases, the rhythm forms part of the structure of the music that EuterpeaScala can handle explicitly.

## Adding Rhythm to a Sequence of Notes

The Rhythm Modifier takes as a parameter ***another* p**iece of Music (the “rhythm Music”). The effect of the modification is that the notes of the modified music play, ignoring the beats in that music, and with the beats taken instead from the rhythm Music. The rhythm Music is used repeatedly as long as there are Notes in the modified Music requiring the rhythmic beat.

The rhythmMusic may be another part of the same musical piece. Or it may be provided simply to express the rhythm. In the latter case, the specific notes used are not relevant and the rhythm Music can be written using the pseudo-Note letter “N”, which would not sound. Rests in the rhythm Music are added to the resultant music every time they repeat – they do not correspond to any Note pitches in the modified music. Rests in the modified Music will be ignored, as they are defined solely by the rhythm Music.

For example:

val pattern = (Rest - N\*3)/8 - N/2

CMin /: Rhythm(pattern) /: (G\*3 - E - F\*3 - D)

is the intro to Beethoven’s Fifth Symphony:



## Repeating Individual Notes in a Rhythm

The NoteRhythm Modifier applies to each individual Note, Drum or Chord in the modified music and causes that Note, Drum or Chord to be repeatedly played according to the rhythm pattern specified in its parameter. The duration of each Note, Drum or Chord will be the length of the rhythm Music, and ant Beats in the modified music will be ignored.

# Ornaments

In many styles of music, notes can have “ornaments”, which are patterns of nearby notes added as the start of note is played. In EuterpeaScala music, these different ornaments are handled by the “Ornament” sub-class of Modifier.

In practice, the Ornament class is never used directly. Instead there are several defined Ornament objects. These use the Note to which the Ornament is applied (the “Home” note), plus the note a scale step above (“Up”) and the note a scale step (or optionally a semitone) below (“Down”). Each of these objects takes a Beat parameter, to specify the beat at which the ornament is played within the current beat duration of the Note as a whole:

* TurnUpDown – Home, Up, Home, Down, Home
* TurnDownUp – Home, Down, Home, Up, Home
* TurnDown – Up, Home, Down, Home
* TurnUp –Down, Home, Up, Home
* MordentUp – Home, Up, Home
* MordentDown – Home, Down, Home
* AcciaccaturaUp – Up, Home
* AcciaccaturaDown – Down, Home
* Trill – Home, Up (repeated for the entire note duration)

For example:

C - D - E - F - G/TurnDownUp(Thirtysecond)

Plays as:



# Continuous Controls

## Controls

Beyond the capability of usual musical notation, Midi music can specify “continuous controllers” to modify aspects of musical performance while notes are sounding.

EuterpeaScala can specify “control envelopes” that allow these continuous controllers to have their values updated as music plays. Within any piece of music can be “control points”. A control point is a combination of a “Controller” and a value in the range 0 to 127. At the time of those control points, the Midi value of the control is that specified. But ***between*** the control points, the Midi value of the control varies continuously between the two end-point values.

So in the example:

val modulation = Controller(Controller.Modulation\_Wheel)

C - D - E - G - modulation(0) –

E - D - C(5) - B - modulation(100) –

A - F - A - B - C - modulation(127)

the changes to the “Modulation\_Wheel” Controller for the current Midi channel start after the G, increasing smoothly in value from 0 to 100 up to the B, and then more slowly (though also smoothly) to 127 by the end.

As an alternative to the absolute integer value for a control, the parameter can be a floating-point number in the range 0.0 to 1.0 which spans the entire range of the control values.

As well as the Controller.Modulation\_Wheel, all the General Midi controllers can be used in this manner.

## Pitch Bend

There is also a pseudo-Controller called Controller.Pitch\_Bend\_Pseudo\_Control and that can be used to change the pitch of notes playing on the current Midi channel. In the case of that pseudo-Controller, the range of values is -8191 to +8191 or -1.0 to +1.0.

For pitch-bend (only), there is another notation available. A pitch-bend envelope can be created and named. It can then be applied to play simultaneously with music. In this way, the same envelope can easily be used to bend many different notes. In the example below, the PitchBend envelope starts after a Quarter beat, rising to maximum value (8191) after a further Quarter, then descending back to zero and holding at zero – the entire pattern lasting a Whole beat. This then plays simultaneously with the G

val pitchBend = PitchBend(0, P(Quarter,0),   
P(Quarter,1.0),   
P(Quarter,0),  
P(Quarter,0))

C - D - E - F - (G/1 & pitchBend)

Additional parameters to a Control or P pitch-bend envelope points can be used to smooth out the envelope curves.

# Checking, Saving and Playing Music

The EuterpeaScala capabilities described above all allow the construction of a Music object, which is an Abstract Syntax Tree (AST) describing the structure and content of the music.

There are three methods that can be applied to the constructed Music object.

* Music.check(), which does nothing, except check the music for “correctness” – in particular in its use of bars.
* Music.play(), which plays the constructed Midi in an internal simple Midi sequencer.
* Music.save(filename), which saves the music as a file in Midi File Format

Music.save() takes an optional second Boolean parameter, which if provided and has value “true”, will cause the Midi file to use “strict” timing. All widths will be 1.0 (Legato) and all Dynamics Modifiers will be ignored. This then makes the file more suitable for printing musical notation in external sequencer software as has been used to provide the illustrations in this documentation.

All three of these methods returns a List of errors found in the Music. Each error comprises a pair of String values:

1. the position of the error, formatted as Bar:Beat:Ticks
2. the message describing the error

Music will only play or save if the list of errors returned is empty.

# An Extended Example

Finally, we conclude with an extended example, which uses many of these capabilities to combine transcription and composition to produce an arrangement of “Caledonia” by Dougie MacLean.

Examining the code, and working backwards from the end, this uses normal Scala definitions to build the tune from verse and chorus (with a tempo and key), and each of the verse and chorus from its individual lines. The chorus finishes with a single chord line which is to be played differently.

But each line is constructed by a call to a locally defined function “line”, which combines that line’s melody notes, lyrics and chords into an arrangement.

The interesting part is in the “line” function:

* The “asMelody” combination of Modifiers puts its melody Music on a Midi track and channel called “melody” which plays on a Cello
* The “asHarmony” combination of Modifiers puts its harmony chords Music on a Midi track and channel called “harmony” which plays on an acoustic guitar. But the chords are automatically arpeggiated, except on the last chord which plays as broken
* The “asBass” combination of Modifiers puts its melody Music on a Midi track and channel called “bass” which plays on an acoustic bass
* The “melody” is formed from the notes (defaulting to Eighth note beat) with the lyrics.
* The “harmony” is formed of the chords, each playing for a dotted Half note (three Quarter notes)
* The “bass” line is not provided. Instead it is derived from the harmony, by extracting the Root note of each Chord. But then that root note is played to a “bassRhythm” with a long bass note and a short up-beat at the end of each bar
* Finally, each line is constructed as the melody, harmony and bass playing together

val caledonia =   
{

def line(notes: Music, lyricText: String, chords: Music, lastChord: Boolean=false): Music =

{

val asMelody = Track("melody") / Channel("melody") /

Instrument(Instruments.Cello)

val asHarmony = Track("harmony") / Channel("harmony") /

Instrument(Instruments.Acoustic\_Guitar\_Nylon) /

(if (lastChord) Broken(0.05) else Arpeggio(8,1,2,(3,4),1,(3,4),1))

val asBass = Track("bass") / Channel("bass") /

Instrument(Instruments.Acoustic\_Bass)

val melody = notes/8/Lyrics(lyricText)

val harmony = chords/2/Dot

val bassRhythm = if (lastChord) NoModifier else NoteRhythm(N/(Half+Eighth) - N/Eighth)

val bass = Octave(-1) /: bassRhythm /: Root /: harmony

melody/asMelody & harmony/asHarmony & bass/asBass

}

val verse1Line1 =

line(C - F - F/4/Dot - C | C - G - G/4/Dot - C | A - B - +C/4 - A - A | B - A - G -- F/4,

"I don't know if you can see the chan-ges that have come ov-er me.-",

F/Maj | C/Maj | D/Min | -Bf/Maj)

val verse1Line2 =

line(C/Pickup | C - F - F/4/Dot - C | C - G - G/4/Dot - C |   
 A/4 - B/4 - +C/4 | B/2/Dot | Rest/2,

"These last few days I've been a-fraid that\_I might drift a-way.",

F/Maj | C/Maj | D/Min | -Bf/Maj | -Bf/Maj7)

val verse1Line3 =

line((C - C)/Pickup | C - F - F/4 - D - C | C - G - G/4/Dot - C |   
 A - B - +C/4 - A - A | B - A - G - F/4/Dot,

"So\_I've been tell-ing old sto-ries, sing-ing songs, “ +   
 “that make me think a-bout where I come from",

F/Maj | C/Maj | D/Min | -Bf/Maj)

val verse1Line4 =

line(C - F - F/4/Dot - C | C - G - G/4/Dot - C |   
 A - B - +C/4/Dot - A | B/2/Dot | Rest/2/Dot,

"That's the reas-on why I seem so far a-way to-day",

F/Maj | C/Maj | D/Min | -Bf/Maj | -Bf/Maj7)

val verse1 = verse1Line1 | verse1Line2 | verse1Line3 | verse1Line4

val chorusLine1 =

line((A - B)/Pickup | +C - +C - +C - B - B - A | A - G/4/Dot -F - G |   
 A/Dot - A/16 - A - +C - (A -- G)/16 - F | D/2/Dot,

"Ah but Let me tell you that I love you and I think a-bout you all- the time",

F/Maj | C/Maj | D/Min | -Bf/Maj)

val chorusLine2 =

line(D/Dot - F/16 - F/4 - F - D | D - C - C/4 - B - A | B - A - G/4/Dot - F | F/2,

"Cal-e-don-ia you're call-ing me and- now I'm go-ing home",

Bf/Maj | F/Maj | C/Maj | F/Maj)

val chorusLine3 =

line((C)/Pickup | +C - +C - +C - B - B - A | A - G/4/Dot -F - G |   
 A/Dot - A/16 - A - +C - (A -- G)/16 - F | D/2/Dot,

"But if I should become a stran-ger you know that it would make me more- than sad",

F/Maj | C/Maj | D/Min | -Bf/Maj)

val chorusLine4 =

line(D/Dot - F/16 - F/4 - F - A | A - G - G - F - (A--G)/16 - F | F/2,

"Cal-e-don-ia been ev-ery-thing I've ev--er had",

Bf/Maj | C/Maj | F/Maj)

val chorus = chorusLine1 | chorusLine2 | chorusLine3 | chorusLine4 |   
 line(Rest/2/Dot, "", F/Maj, true)

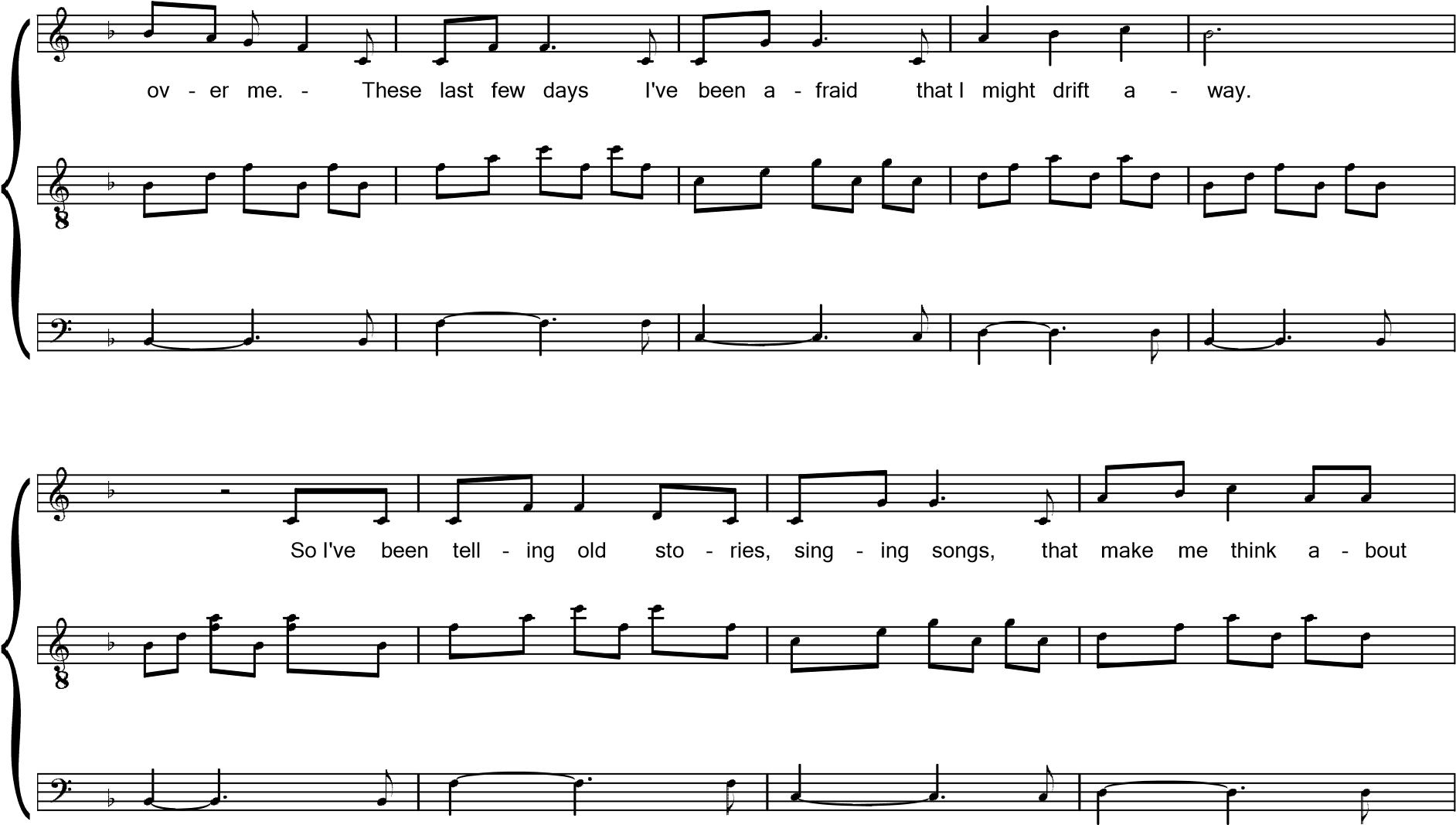
TimeSig(3,4) /: FMaj /: Tempo(100) /: (verse1 | chorus)

}

This results in the Midi file which prints as:

Caledonia

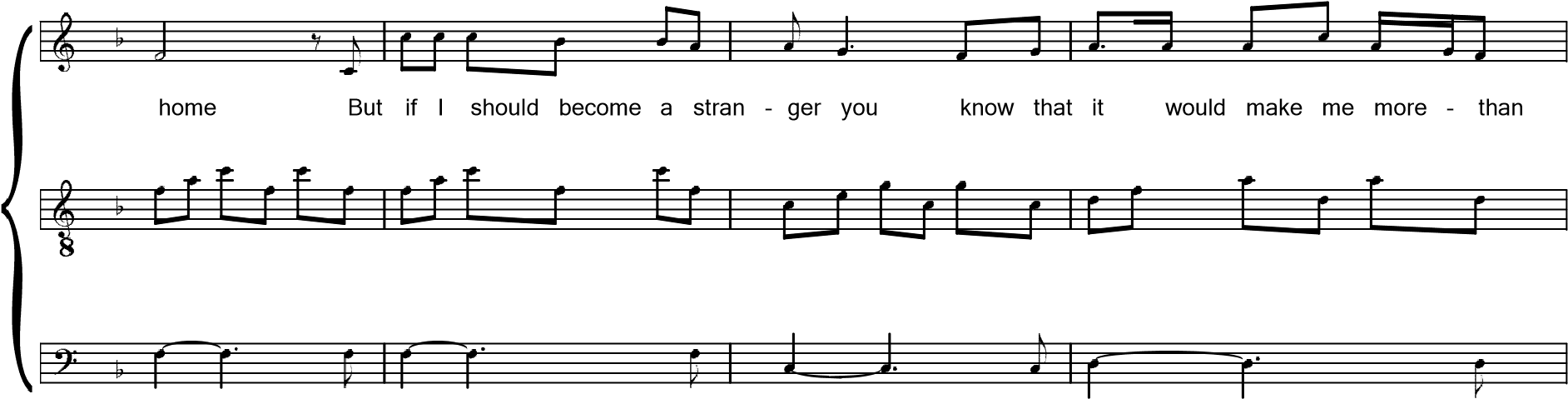


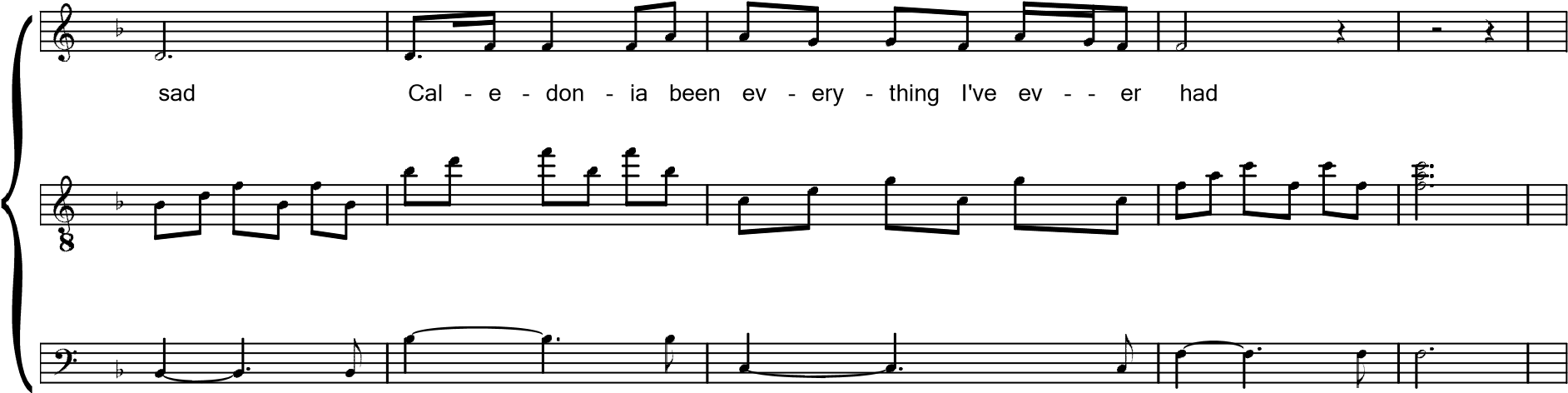




Caledonia







# Reference

## Music

## Operators

## Modifiers

1. Euterpe was the ancient Greek muse of music - the less well known sister of Terpsichore, muse of dance. [↑](#footnote-ref-1)
2. And a further one used behind the scenes to implement the VolumeChange Modifier [↑](#footnote-ref-2)