Contents

The Music Suite

The Music Suite is a system for creating, processing or analyzing music. It is based on the Haskell language. It is designed with three goals in mind:

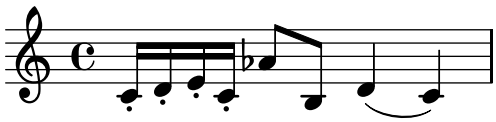
- Describe what the music *is*, rather than how it is to be performed.
- Avoid imposing stylistic or theoretical assumptions on the music.
- Include common notation and theory as a *special case*.

The Music Suite is both a language in its own right and a Haskell library. Being embedded in Haskell has several advantages, it allow the developers to focus on the contents and the users to make use of any feature in the Haskell language.

The Music Suite uses several advanced language constructs internally and requires a relatively new Haskell compiler (see Installing the Suite).

An example

To generate music we write an *expressions* such as this one:

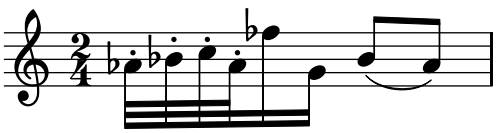


```
let
    m = staccato (scat [c,d,e,c]^/2) |> ab |> b_ |> legato (d |> c)^*2
in stretch (1/8) m
```

To transform music, we write a *function*. For example the following function halves all durations and transposes all pitches up a minor sixth:

```
up (minor sixth) . compress 2
```

Applied to the above music we get:



Input and output

The Music Suite works well with the following input and output formats.

- MusicXML
- Lilypond
- ABC notation
- MIDI

Other formats are being added in the near future, see Import and export for a more detailed overview.

A note on the versioning

The Music Suite consists of a group of packages released concurrently under a common optimistic version number. The library was deliberately released *prematurely* in order to encourage its developers to work on it more.

More information

For a complete reference, see the reference documentation.

Getting Started

Installing the Suite

The Music Suite depends on the Haskell platform.

While not strictly required, Lilypond is highly recommended as it allow you to preview musical scores. See Import and Export for other formats.

To install the suite, simply install the Haskell platform, and then run:

```
cabal install music-preludes
```

Writing music

This chapter will cover how to use the Music Suite to generate music. Later on we will cover how to *import* and *transform* music.

One of the main points of the Music Suite is to avoid committing to a *single*, closed music representation. Instead it provides a set of types and type constructors that can be used to construct an arbitrary representation of music.

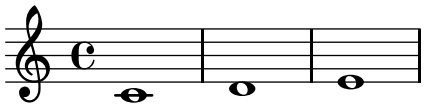
Usually you will not want to invent a new representation from scratch, but rather start with a standard representation and customize it when needed. The default representation is defined in the `Music.Prelude.Basic` module, which is implicitly imported in all the examples below. See Customizing the Music Representation for other examples.

With music files

A piece of music is described by a *expressions* such as this one:

```
c |> d |> e
```

The simplest way to render this expression is to save it in a file named `foo.music` (or similar) and convert it using `music2pdf foo.music`. This should render a file called `foo.pdf` containing the following:



There are several programs for converting music expressions:

- `music2mid`
- `music2musicxml`
- `music2ly`
- `music2wav`
- `music2pdf`

With Haskell files

Alternatively, you can create a file called `test.hs` (or similar) with the following structure:

```
import Music.Prelude.Basic

main = defaultMain music
music = c |> d |> e
```

Then either execute it using:

```
$ runhaskell test.hs
```

or compile and run it with

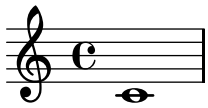
```
$ ghc --make test
$ ./test
```

In this case the resulting program will generate and open a file called `test.pdf` containing the output seen above.

Music files and Haskell files using `defaultMain` are equivalent in every aspect. In fact, the `music2...` programs are simple utilities that substitutes a single expression into a Haskell module such as the one above and executes the resulting main function.

Time and duration

A single note can be entered by its name. This will render a note in the middle octave with a duration of one. Note that note values and durations correspond exactly, a duration of `1` is a whole note, a duration of `1/2` is a half note, and so on.

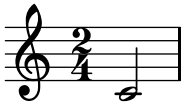


```
c
```

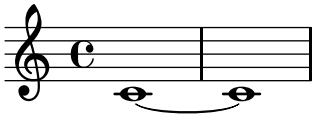
To change the duration of a note, use `stretch` or `compress`. Note that:

```
compress x = stretch (1/x)
```

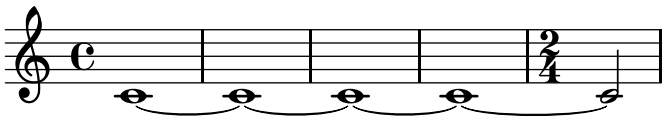
for all values of x.



```
stretch (1/2) c
```



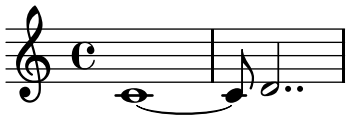
```
stretch 2 c
```



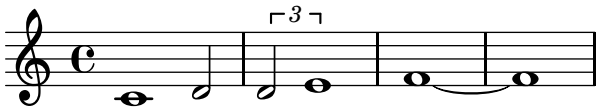
```
stretch (4+1/2) c
```

TODO delay

Offset and duration is not limited to simple numbers. Here are some more complex examples:



```
c^*(9/8) |> d^*(7/8)
```




```
stretch (2/3) (scat [c,d,e]) |> f^*2
```

As you can see, note values, tuplets and ties are added automatically

The `^*` and `^/` operators can be used as shorthands for `delay` and `compress`.

```
(c |> d |> e |> c |> d^*2 |> d^*2)^/16
```

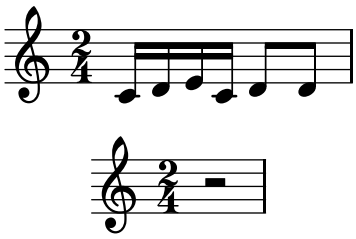
Allthough the actual types are more general, you can think of `c` as an expression of type `Score Note`, and the transformations as functions `Score Note -> Score Note`.

```
up (perfect octave) . compress 2 . delay 3 $ c
```

Composition

Music expressions can be composed `<>`:

```
c <> e <> g
```



TODO fundamentally, `<>` is the only way to compose music...

Or in sequence using `|>`:

```
c |> d |> e
```

Or partwise using `</>`:

```
c </> e </> g
```

Here is a more complex example:

```
let
    scale = scat [c,d,e,f,g,a,g,f]^/8
    triad a = a <> up _M3 a <> up _P5 a
in up _P8 scale </> (triad c)^/2 |> (triad g_)^/2
```

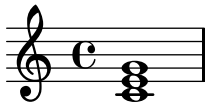
As a shorthand for `x |> y |> z ..`, we can write `scat [x, y, z]` (short for *sequential concatenation*).

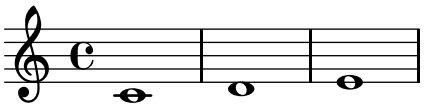
```
scat [c,e..g]^/4
```

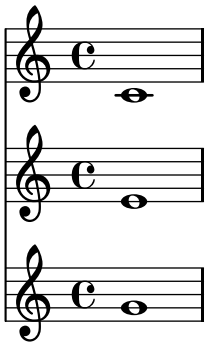
For `x <> y <> z ..`, we can write `pcat [x, y, z]` (short for *parallel concatenation*).

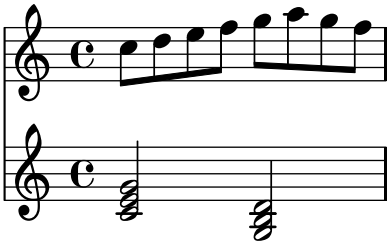
```
pcat [c,e..g]^/2
```

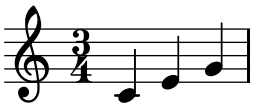
Actually, `scat` and `pcat` used to be called `melody` and `chord` back in the days, but I figured out that these are names that you actually want to use in your own code.

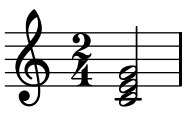












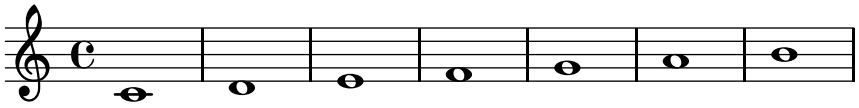
Pitch

Pitch names

To facilitate the use of non-standard pitch, the standard pitch names are provided as overloaded values, referred to as *pitch literals*.

To understand how this works, think about the type of numeric literal. The values $0, 1, 2$ etc. have type `Num a => a`, similarly, the pitch literals $c, d, e, f...$ have type `IsPitch a => a`.

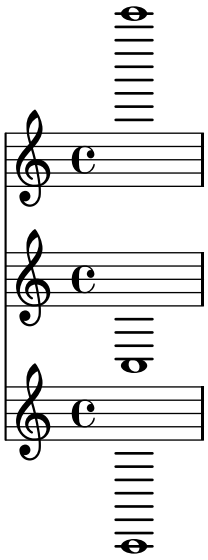
For Western-style pitch types, the standard pitch names can be used:



```
scat [c, d, e, f, g, a, b]
```

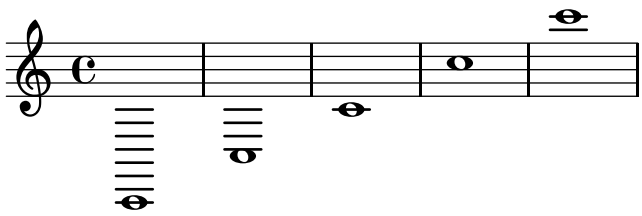
Pitch names in other languages work as well, for example `ut, do, re, mi, fa, so, la, ti, si`.

You can change octave using `octavesUp` and `octavesDown`:



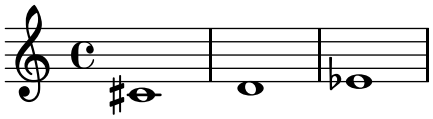
```
octavesUp 4 c
    </>
octavesUp (-1) c
    </>
octavesDown 2 c
```

There is also a shorthand for other octaves:



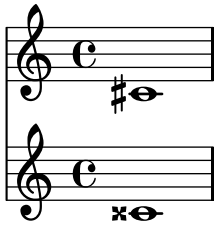
```
c__ |> c_ |> c |> c' |> c''
```

Sharps and flats can be added by the functions `sharp` and `flat`, which are written *postfix* thanks to some overloading magic.



```
c sharp |> d |> e flat
```

You can also use the ordinary (prefix) versions `sharpen` and `flatten`.

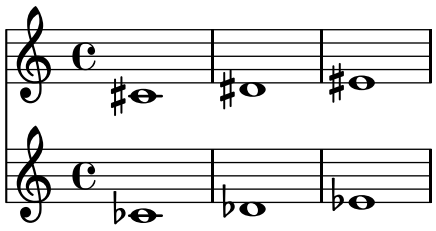


```
sharpen c
    </>
(sharpen . sharpen) c
```

As you might expect, there is also a shorthand for sharp and flat notes:

```
(cs |> ds |> es)     -- sharp
    </>
(cb |> db |> eb)     -- flat
```

Here is an overview of all pitch notations:



```
sharpen c             == c sharp       == cs
flatten d             == d flat        == ds
(sharpen . sharpen) c == c doubleSharp == css
(flatten . flatten) d == d doubleFlat  == dss
```

Note that `cs == db` may or may not hold depending on which pitch representation you use.

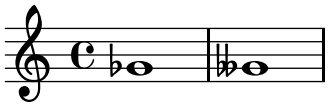
Interval names

Interval names are overloaded in a manner similar to pitches, and are consequently referred to as *interval literals*. The corresponding class is called `IsInterval`.

Here and elsewhere in the Music Suite, the convention is to follow standard theoretical notation, so *minor* and *diminished* intervals are written in lower-case, while *major* and *perfect* intervals are written in upper-case. Unfortunately, Haskell does not support overloaded upper-case values, so we have to adopt an underscore prefix:

```
minor third      == m3
major third      == _M3
perfect fifth    == _P5
diminished fifth == d5
minor ninth      == m9
```

Similar to `sharpen` and `flatten`, the `augment` and `diminish` functions can be used to alter the size of an interval. For example:



```
let
    intervals = [diminish (perfect fifth), (diminish . diminish) (perfect fifth)]
in scat $ fmap (`up` c) intervals
```

You can add pitches and intervals using the `.-.` and `.+^` operators. To memorize these operators, think of pitches and points `.` and intervals as vectors `^`.

```
music+haskellx setPitch (c .+^ m3) $ return c_
```

Qualified pitch and interval names

There is nothing special about the pitch and interval literals, they are simply values exported by the `Music.Pitch.Literal` module. While this module is reexported by the standard music preludes, you can also import it qualified if you want to avoid bringing the single-letter pitch names into scope.

```
Pitch.c |> Pitch.d .+^ Interval.m3
```

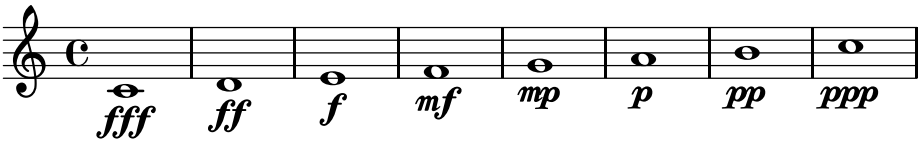
TODO overloading, explain why the following works:

```
return (c::Note) == (c::Score Note)
```

Dynamics

Dynamic values are overloaded in the same way as pitches. The dynamic literals are defined in `Music.Dynamics.Literal` and have type `IsDynamics a => a`.

An overview of the dynamic values:



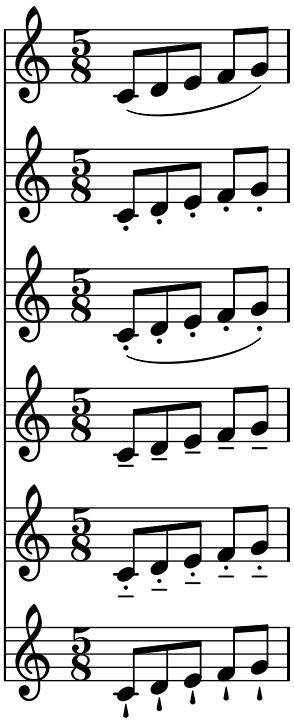

```
scat $ zipWith dynamics [fff,ff,_f,mf,mp,_p,pp,ppp] [c..]
```

TODO other ways of applying dynamics

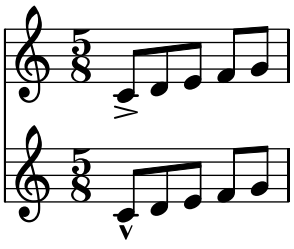
Articulation

Some basic articulation functions are `legato`, `staccato`, `portato`, `tenuto`, `separated`, `spiccato`:

```
legato (scat [c..g]^/8)
    </>
staccato (scat [c..g]^/8)
```



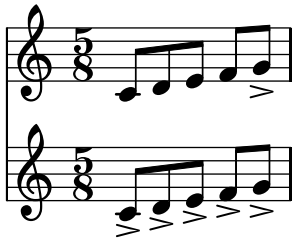
```
    </>
portato (scat [c..g]^/8)
    </>
tenuto (scat [c..g]^/8)
    </>
separated (scat [c..g]^/8)
    </>
spiccato (scat [c..g]^/8)

accent marcato
```



```
accent (scat [c..g]^/8)
    </>
```

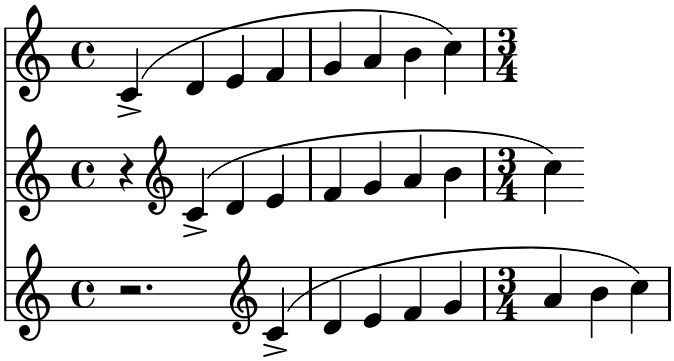
```
marcato (scat [c..g]^/8)
```

```
accentLast accentAll
```



```
accentLast (scat [c..g]^/8)
    </>
accentAll (scat [c..g]^/8)
```

Applying articulations over multiple parts:



```
let
    p1 = scat [c..c']^/4
    p2 = delay (1/4) $ scat [c..c']^/4
    p3 = delay (3/4) $ scat [c..c']^/4
in (accent . legato) (p1 </> p2 </> p3)
```

Parts

```
Division
```

```
Subpart
```

```
Part
```

```
Instrument
```

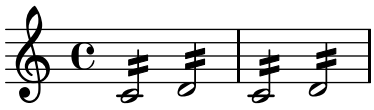
```
Solo
```

Space

TODO

Tremolo

`tremolo`

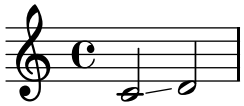


```
tremolo 2 $ times 2 $ (c |> d)^/2
```

TODO chord tremolo

Slides and glissando

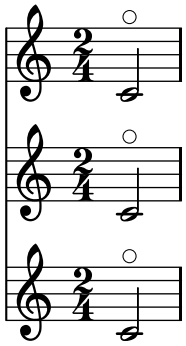
`slide` `glissando`



```
glissando $ scat [c,d]^/2
```

Harmonics

Use the `harmonic` function:



```
(harmonic 1 $ c^/2)
    </>
(harmonic 2 $ c^/2)
    </>
(harmonic 3 $ c^/2)
```

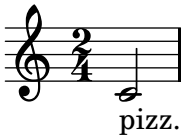
TODO artificial harmonics

```
artificial
```

Text

TODO

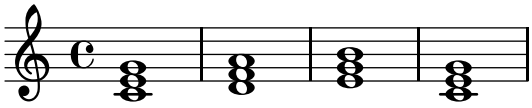
```
text
```




```
text "pizz." $ c^/2
```

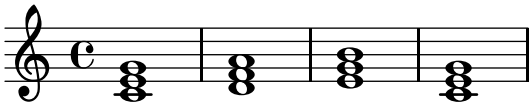
Chords

Note with the same onset and offset are rendered as chords by default. If you want to prevent this you must put them in separate parts.



```
scat [c,d,e,c] <> scat [e,f,g,e] <> scat [g,a,b,g]
```

Or, equivalently:



```
pcat [c,e,g] |> pcat [d,f,a] |> pcat [e,g,b] |> pcat [c,e,g]
```

TODO how part separation works w.r.t. division etc

`simultaneous`

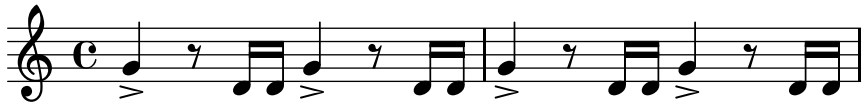
`mapSimultaneous`

Rests

Similar to chords, there is usually no need to handle rests explicitly.

TODO add explicit rests etc

`removeRests`

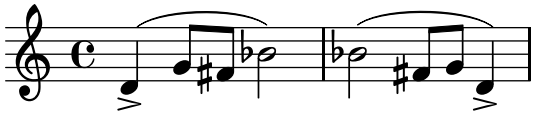


```
removeRests $ times 4 (accent g^*2 |> rest |> scat [d,d]^/2)^/8
```

Transformations

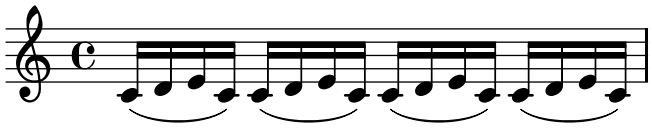
Time

`rev`

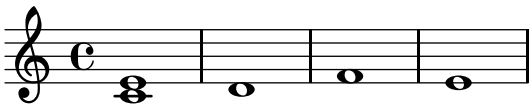


```
let
    melody = accent $ legato $ scat [d, scat [g,fs]^/2,bb^*2]^/4
in melody |> rev melody
```

`times`



```
let
    melody = legato $ scat [c,d,e,c]^/16
in times 4 $ melody
```

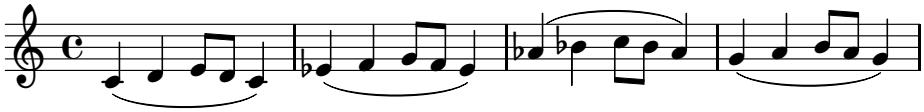


```
sustain
```

```
scat [e,d,f,e] <> c
```

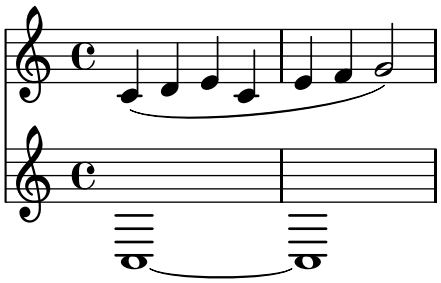
```
anticipate
```

```
repeated
```



```
let
    m = legato $ scat [c,d,scat [e,d]^/2, c]^/4
in [c,eb,ab,g] `repeated` (\p -> up (asPitch p .-. c) m)
```

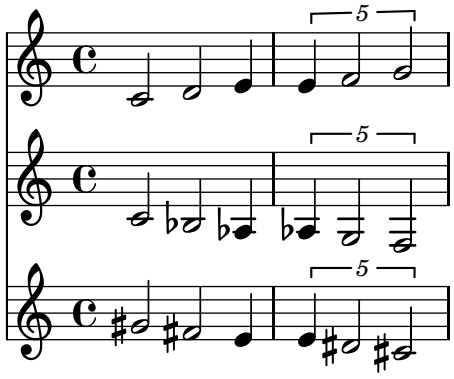
Onset and duration



```
let
    melody = asScore $ legato $ scat [scat [c,d,e,c], scat [e,f], g^*2]
    pedal  = asScore $ delayTime (onset melody) $ stretch (duration melody) $ c_
in compress 4 $ melody </> pedal
```

Pitch

```
inv
```




```
(scat [c..g]^*(2/5))
    </>
(inv c $ scat [c..g]^*(2/5))
    </>
(inv e $ scat [c..g]^*(2/5))
```

Pitches and intervals

TODO

Name and accidental

TODO

Spelling

TODO

Quality and number

TODO

Intonation

TODO

Inspecting dissonant intervals

TODO

Semitones and enharmonic equivalence

TODO

Spelling

TODO

Scales

TODO

Chords

TODO

Parts

Instrument, part and sub-part

Extracting and modifying parts

Part composition

Time-based structures

`Delayable`

`Stretchable`

`HasOnset`

`HasOffset`

`HasDuration`

Time and duration

`Time`

`Duration`

Spans

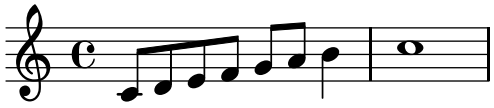
`Span`

Notes

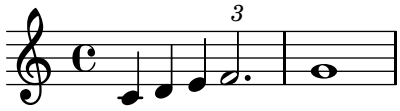
`Note`

Voice

A `Voice` represents a single voice of music. It consists of a sequence of values with duration, but no time.

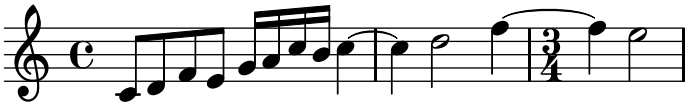


```
stretch (1/4) $ scat [c..a]^/2 |> b |> c'^*4
```




```
stretch (1/2) $ scat [c..e]^/3 |> f |> g^*2
```

It can be converted into a score by stretching each element and composing in sequence.

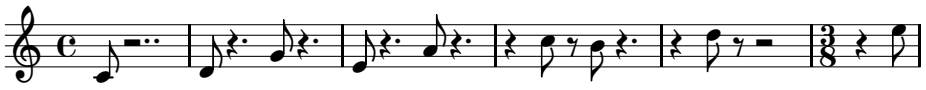


```
let
    x, y :: Voice Note

    x = voice [ (1, c),
                (1, d),
                (1, f),
                (1, e) ]

    y = join $ voice [ (1, x),
                       (0.5, up _P5 x),
                       (4, up _P8 x) ]

in stretch (1/8) $ voiceToScore $ y
```

Tracks

A `Track` is similar to a score, except that it events have no offset or duration. It is useful for representing point-wise occurrences such as samples, cues or percussion notes.

It can be converted into a score by delaying each element and composing in parallel. An explicit duration has to be provided.



```
let
    x, y :: Track Note
    x = track [ (0, c), (1, d), (2, e) ]
    y = join $ track [ (0, x), (1.5,  up _P5 x), (3.25, up _P8 x) ]
in trackToScore (1/8) y
```

Scores

`Score`

Meta-information

It is often desirable to annotate music with extraneous information, such as title, creator or time signature. Also, it is often useful to mark scores with structural information such as movement numbers, rehearsal marks or general

annotations. In the Music Suite these are grouped together under the common label *meta-information*.

Each type of meta-information is stored separately and can be extracted and transformed depending on its type. Each type of meta-information has a default value which is implicitly chosen if no meta-information of the given type has been entered (for example the default title is empty, the default key signature is C major and so on).

The distinction between ordinary musical data and meta-data is not always clear cut. As a rule of thumb, meta-events are any kind of event that does not directly affect how the represented music sounds when performed. However they might affect the appearance of the musical notation. For example, a *clef* is meta-information, while a *slur* is not. A notable exception to this rule is meta-events affecting tempo such as metronome marks and fermatas, which usually *do* affect the performance of the music.

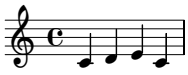
Title

```
title
```

```
subtitle
```

```
subsubtitle
```

Frere Jaques



```
title "Frere Jaques" $ scat [c,d,e,c]^/4
```

Attribution

```
composer
```

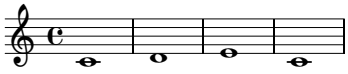
```
lyricist
```

```
arranger
```

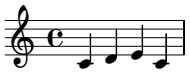
```
attribution
```

```
attributions
```

Anonymous



Anonymous



```
composer "Anonymous" $ scat [c,d,e,c]
```

```
composer "Anonymous" $ lyricist "Anonymous" $ arranger "Hans" $ scat [c,d,e,c]^/4
```

Key signatures

key

keySignature

keySignatureDuring

withKeySignature

Time signatures

time

compoundTime

timeSignature

timeSignatureDuring

withTimeSignature

Tempo

metronome

tempo

tempoDuring

withTempo

[renderTempo][renderTempo]

Fermatas, caesuras and breathing marks

TODO

Ritardando and accellerando

TODO

Rehearsal marks

TODO

`rehearsalMark`

`rehearsalMarkDuring`

`withRehearsalMark`

Barlines and repeats

There is generally no need to enter bars explicitly, as this information can be inferred from other meta-information. Generally, the following meta-events (in any part), will force a change of bar:

- Key signature changes
- Time signature changes
- Tempo changes
- Rehearsal marks

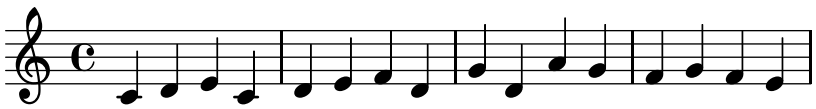
However, the user may also enter explicit bar lines using the following functions:

`barline`

`doubleBarline`

`finalBarline`

Whenever a bar line is created as a result of a meta-event, an shorted time signature may need to be inserted as in:



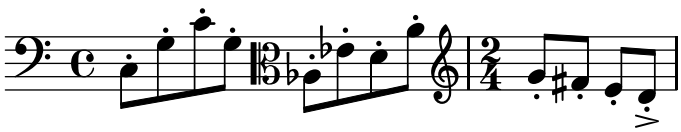
```
compress 4 $ timeSignature (4/4) (scat [c,d,e,c,d,e,f,d,g,d]) |> timeSignature (3/4) (scat
```

TODO adapt getBarDurations and getBarTimeSignatures to actually do this

TODO repeats

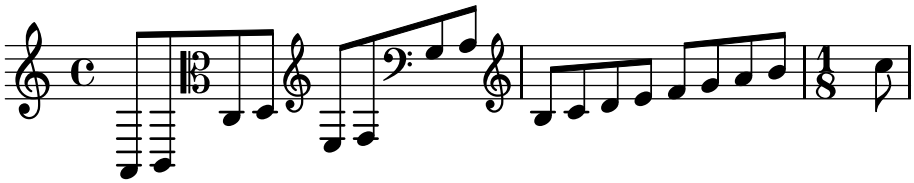
Clefs

To set the clef for a whole passage, use `clef`. The clef is used by most notation backends and ignored by audio backends.



```
let
    part1 = clef FClef $ staccato $ scat [c_,g_,c,g_]
    part2 = clef CClef $ staccato $ scat [ab_,eb,d,a]
    part3 = clef GClef $ staccato $ accentLast $ scat [g,fs,e,d]
in compress 8 $ part1 |> part2 |> part3
```

To set the clef for a preexisting passage in an existing score, use `clefDuring`.

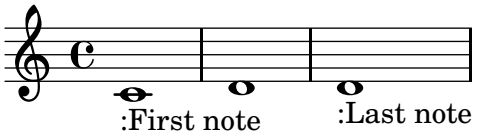


```
clefDuring (0.25 <-> 0.5) CClef $ clefDuring (0.75 <-> 1) FClef $ compress 8 $ scat [c_..c']
```

Annotations

Annotations are simply textual values attached to a specific section of the score. In contrast to other types of meta-information annotations always apply to the whole score, not to a single part. To annotate a score use `annotate`, to annotate a specific span, use `annotateSpan`.

Annotations are invisible by default. To show annotations in the generated output, use `showAnnotations`.




```
showAnnotations $ annotate "First note" c |> d |> annotate "Last note" d
```

Custom meta-information

Meta-information is not restricted to the types described above. In fact, the user can add meta-information of any type that satisfies the `IsAttribute` constraint, including user-defined types. Each type of meta-information is stored separately from other types, and is invisible to the user by default. You might think of each score as having one an infinite set of associated meta-scores, each containing both part-specific and global meta information.

Meta-information is required to implement `Monoid`. The `mempty` value is used as a default value for the type, while the `mappend` function is used to combine the default value and all values added by the user.

`addMetaNote`

`addGlobalMetaNote`

`withMeta`

`withGlobalMeta`

`withMetaAtStart`

`withGlobalMetaAtStart`

Typically, you want to use a monoid similar to `Maybe`, `First` or `Last`, but not one derived from the list type. The reason for this is that meta-scores compose, so that `getMeta (x <> y) = getMeta x <> getMeta y`.

TODO unexpected results with filter and recompose, solve by using a good Monoid Acceptable Monoids are Maybe and Set/Map, but not lists (ordered sets/unique lists OK) See issue 103

Import and export

The standard distribution (installed as part of `music-preludes`) of the Music Suite includes a variety of input and output formats. There are also some experimental formats, which are distributed in separate packages, these are marked as experimental below.

The conventions for input or output formats is similar to the convention for properties (TODO ref above): for any type `a` and format `T a`, input formats are defined by an *is* constraint, and output format by a *has* constraint. For example, types that can be exported to Lilypond are defined by the constraint `HasLilypond a`, while types that can be imported from MIDI are defined by the constraint `IsMidi a`.

MIDI

All standard representations support MIDI input and output. The MIDI representation uses HCodecs and the real-time support uses hamid.

Beware that MIDI input may contain time and pitch values that yield a non-readable notation, you need an sophisticated piece of analysis software to convert raw MIDI input to quantized input.

Lilypond

All standard representations support Lilypond output. The lilypond package is used for parsing and pretty printing of Lilypond syntax. Lilypond is the recommended way of rendering music.

Lilypond input is not available yet but will hopefully be added soon.

An example:

```
toLyString $ asScore $ scat [c,d,e]
```

```
<<
    \new Staff { <c'>1 <d'>1 <e'>1 }
>>
```

MusicXML

All standard representations support MusicXML output. The musicxml2 package is used for parsing and pretty printing.

The output is fairly complete, with some limitations (reports welcome). There are no plans to support input in the near future.

Beware of the extreme verboseness of XML, for example:

```
toXmlString $ asScore $ scat [c,d,e]
```

```
<?xml version='1.0' ?>
<score-partwise>
  <movement-title>Title</movement-title>
  <identification>
    <creator type="composer">Composer</creator>
  </identification>
  <part-list>
    <score-part id="P1">
```

```
    <part-name></part-name>
  </score-part>
</part-list>
<part id="P1">
  <measure number="1">
    <attributes>
      <key>
        <fifths>0</fifths>
        <mode>major</mode>
      </key>
    </attributes>
    <attributes>
      <divisions>768</divisions>
    </attributes>
    <direction>
      <direction-type>
        <metronome>
          <beat-unit>quarter</beat-unit>
          <per-minute>60</per-minute>
        </metronome>
      </direction-type>
    </direction>
    <attributes>
      <time symbol="common">
        <beats>4</beats>
        <beat-type>4</beat-type>
      </time>
    </attributes>
    <note>
      <pitch>
        <step>C</step>
        <alter>0.0</alter>
        <octave>4</octave>
      </pitch>
      <duration>3072</duration>
      <voice>1</voice>
      <type>whole</type>
    </note>
  </measure>
  <measure number="2">
    <note>
      <pitch>
        <step>D</step>
        <alter>0.0</alter>
        <octave>4</octave>
      </pitch>
```

```
        <duration>3072</duration>
        <voice>1</voice>
        <type>whole</type>
      </note>
    </measure>
    <measure number="3">
      <note>
        <pitch>
          <step>E</step>
          <alter>0.0</alter>
          <octave>4</octave>
        </pitch>
        <duration>3072</duration>
        <voice>1</voice>
        <type>whole</type>
      </note>
    </measure>
  </part>
</score-partwise>
```

ABC Notation

ABC notation (for use with abcjs or similar engines) is still experimental.

Guido

Guido output (for use with the GUIDO engine) is not supported yet. This would be useful, as it allow real-time rendering of scores.

Vextab

Vextab output (for use with Vexflow) is not supported yet.

Sibelius

The music-sibelius package provides experimental import of Sibelius scores (as MusicXML import is not supported).

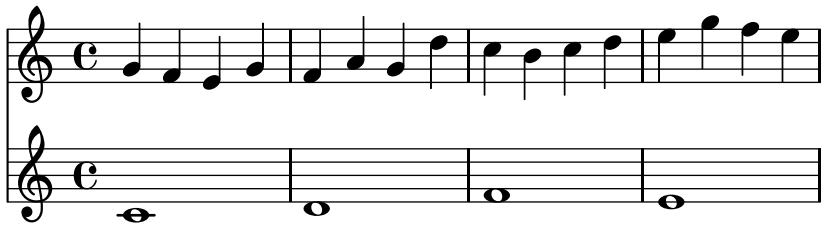
Customizing music representation

TODO

Examples

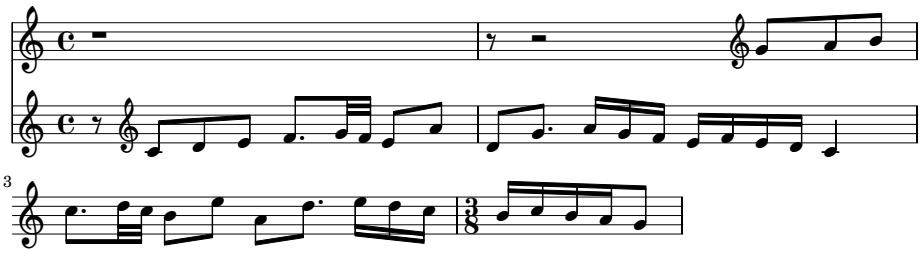
Some more involved examples:

Counterpoint



```
let
    subj = asScore $ scat [ c,       d,       f,          e          ]
    cs1  = asScore $ scat [ g,f,e,g, f,a,g,d', c',b,c',d', e',g',f',e' ]
in compress 4 cs1 </> subj
```

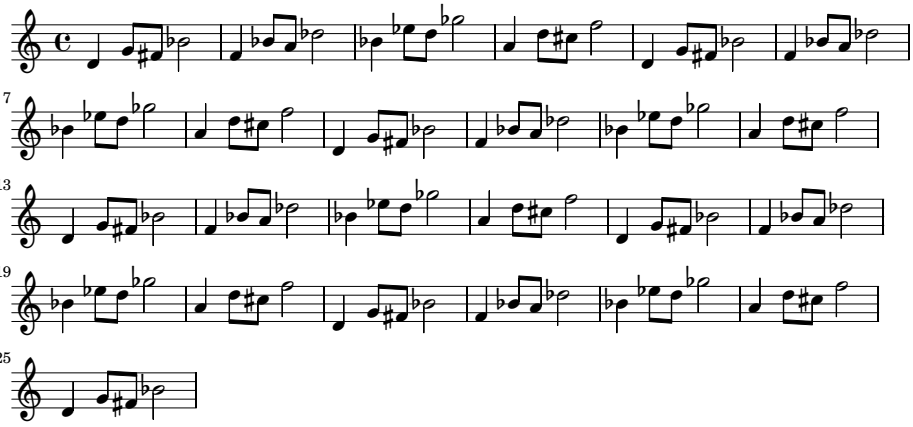
TODO about



```
let subj = removeRests $ scat [
            scat [rest,c,d,e],
            f^*1.5, scat[g,f]^/4, scat [e,a,d], g^*1.5,
            scat [a,g,f,e,f,e,d]^/2, c^*2
        ]^/8

in (delay 1.5 . up _P5) subj </> subj
```

Generative music

```
let
    row = cycle [c,eb,ab,asPitch g]
    mel = asScore $ scat [d, scat [g,fs]^/2,bb^*2]^/4
in (take 25 $ row) `repeated` (\p -> up (asPitch p .-. c) mel)
```



Viola duo

Design overview

TODO

The Music Suite consists of a number of packages. These can be divided into two categories:

- *Music* packages that provide classes, types and functions related to a particular aspect of musical representation such as time, pitch, dynamics and so on. The name of these packages always begin with `music`.
- *Supporting* packages that implement a musical representation (`musicxml2`, `lilypond`, `abcnotation`), or miscellaneous functionality such as cross-platform MIDI support (`hamid`). These packages can be used as stand-alone packages but are included in the Suite for completeness.

There is no central package, instead the aim has been to separate the various issues that arise in music representations as clearly as possible. In particular, the `music-score` package, which provide scores and other temporal containers, does *not* depend on packages that provide models of musical aspects such as `music-pitch`, neither do these libraries depend on `music-score`.

The reason for this is that we want to keep musical structure and content separate. This is a form of the expression problem: if one depended on the other we would either always force the user into a particular form of musical structure, or a particular form of musical material.

However, some packages have special roles:

- The `music-pitch-literal` and `music-dynamics-literal` are minimal packages that provide musical *literals*, i.e. common vocabulary overloaded on result type. This means other packages can import and provide instances for the literals without having to depend on a specific representation.
- The `music-preludes` provides modules that import modules from both `music-score` and `music-pitch` and its sister packages.

Compability with other libraries

The Music Suite libraries does not profess to be compatible with any other music *representation* library[1], and deliberately claims the whole `Music` top-level package. The aim is that functionality from these packages should eventually be included into the Music Suite packages. However it can be used with packages that implement audio processing, synthesis, interaction with musical instruments, FRP libraries and so on.

The concepts and abstractions used in the suite overlap with some fundamental concepts form FRP, but the focus is fundamentally different. While FRP libraries focus on reacting to the external world, the Music Suite focus on modeling temporal values and musical concepts.

Acknowledgements

The Music Suite is indebted to many other previous libraries and computer music environments, particularly Common Music, Max/MSP, SuperCollider, music21, Guido, Lilypond and Abjad. Some of the ideas for the quantization algorithms came from Fomus.

It obviously ows a lot to the Haskell libraries that it follows including Haskore, Euterpea and temporal-media. The idea of defining a custom internal representation, but relying on standardized formats for input and output comes from Pandoc. The idea of splitting the library into a set of packages (and the name) comes from the Haskell Suite. The temporal structures, their instances and the concept of denotational design comes from Reactive (and its predecessors). Diagrams provided the daring example and some general influences on the design.

[1] Including Haskore, Euterpea, hts and temporal-media