Haskore Music Tutorial

Paul Hudak
Yale University
Department of Computer Science
New Haven, CT 06520
paul.hudak@yale.edu

February 14, 1997
(Revised November 1998)
(Revised February 2000)
(Constantly mixed up in 2004 - 2007 by Henning Thielemann :-)

Contents

1 Introduction

Haskore is a collection of Haskell modules designed for expressing musical structures in the high-level, declarative style of *functional programming*. In Haskore, musical objects consist of primitive notions such as notes and rests, operations to transform musical objects such as transpose and tempo-scaling, and operations to combine musical objects to form more complex ones, such as concurrent and sequential composition. From these simple roots, much richer musical ideas can easily be developed.

Haskore is a means for describing *music*—in particular Western Music—rather than *sound*. It is not a vehicle for synthesizing sound produced by musical instruments, for example, although it does capture the way certain (real or imagined) instruments permit control of dynamics and articulation.

Haskore also defines a notion of *literal performance* through which *observationally equivalent* musical objects can be determined. From this basis many useful properties can be proved, such as commutative, associative, and distributive properties of various operators. An *algebra of music* thus surfaces.

In fact a key aspect of Haskore is that objects represent both *abstract musical ideas* and their *concrete implementations*. This means that when we prove some property about an object, that property is true about the music in the abstract *and* about its implementation. Similarly, transformations that preserve musical meaning also preserve the behavior of their implementations. For this reason Haskell is often called an *executable specification language*; i.e. programs serve the role of mathematical specifications that are directly executable.

Building on the results of the functional programming community's Haskell effort has several important advantages: First, and most obvious, we can avoid the difficulties involved in new programming language design, and at the same time take advantage of the many years of effort that went into the design of Haskell. Second, the resulting system is both *extensible* (the user is free to add new features in substantive, creative ways) and *modifiable* (if the user doesn't like our approach to a particular musical idea, she is free to change it).

In the remainder of this paper I assume that the reader is familar with the basics of functional programming and Haskell in particular. If not, I encourage reading at least *A Gentle Introduction to Haskell* [HF92] before proceeding. I also assume some familiarity with *equational reasoning*; an excellent introductory text on this is [BW88].

1.1 Acknowledgements

Many students have contributed to Haskore over the years, doing for credit what I didn't have the spare time to do! I am indebted to them all: Amar Chaudhary, Syam Gadde, Bo Whong, and John Garvin, in particular. Thanks also to Alastair Reid for implementing the first Midi-file writer, to Stefan Ratschan for porting Haskore to GHC, and to Matt Zamec for help with the Csound compatibility module. I would also like to express sincere thanks to my friend and talented New Haven composer, Tom Makucevich, for being Haskore's most faithful user.

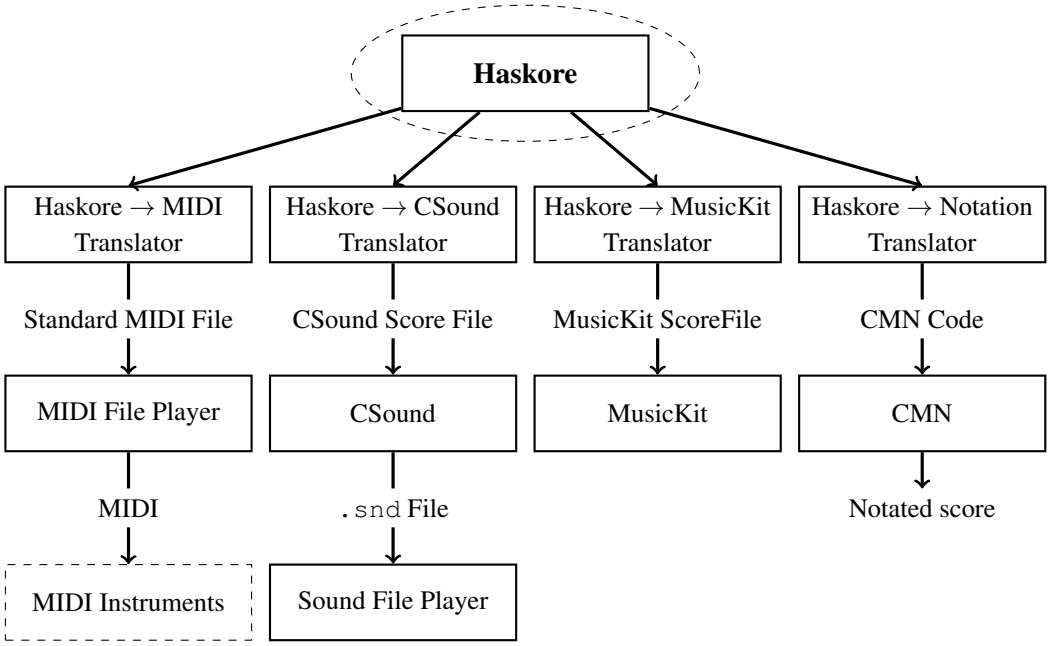


Figure 1: Overall System Diagram

2 The Architecture of Haskore

Figure 1 shows the overall structure of Haskore. Note the independence of high level structures from the "music platform" on which Haskore runs. Originally, the goal was for Haskore compositions to run equally well as conventional midi-files [IMA90], NeXT MusicKit score files [JB91] [1], and CSound score files [Ver86] [2], and for Haskore compositions to be displayed and printed in traditional notation using the CMN (Common Music Notation) subsystem. [3] In reality, three platforms are currently supported: MIDI, CSound, and some signal processing routines written in Haskell. For musical notation an interface to Lilypond is currently in progress.

In any case, the independence of abstract musical ideas from the concrete rendering platform is accomplished by having abstract notions of *musical object*, *player*, *instrument*, and *performance*. All of this resides in the box labeled "Haskore" in the diagram above.

At the module level, Haskore is organized as follows:

```
module Haskore (
   module Haskore.Music,
   module Haskore.Performance,
   module Haskore.Performance.Player,
   module Haskore.Interface.MIDI.Write,
   module Haskore.Interface.MIDI.Read,
   module Haskore.Interface.MIDI.Render,
```

[1] The NeXT music platform is obsolete.

[2] There also exists a translation to CSound for an earlier version of Haskore.

[3] We have abandoned CMN entirely, as there are now better candidates for notation packages into which Haskore could be mapped.

```
   module Sound.MIDI.File.Save,
   module Sound.MIDI.File.Load,
   ) where

import qualified Haskore.Music
import qualified Haskore.Performance
import qualified Haskore.Performance.Player
import qualified Haskore.Interface.MIDI.Write
import qualified Haskore.Interface.MIDI.Read
import qualified Haskore.Interface.MIDI.Render
import qualified Sound.MIDI.File.Save
import qualified Sound.MIDI.File.Load
```

This document was written in the *literate programming style*, and thus the LaTeX manuscript file from which it was generated is an *executable Haskell program*. It can be compiled under LaTeX in two ways: a basic mode provides all of the functionality that most users will need, and an extended mode in which various pieces of lower-level code are provided and documented as well (see file header for details). This version was compiled in extended mode. The document can be retrieved via WWW from: http://haskell.org/haskore/ (consult the README file for details). It is also delivered with the standard joint Nottingham/Yale Hugs release.

The Haskore code conforms to Haskell 1.4, and has been tested under the June, 1998 release of Hugs 1.4. Unfortunately Hugs does not yet support mutually recursive modules, so all references to the module `Player` in this document are commented out, which in effect makes it part of module `Performance` (with which it is mutually recursive).

A final word before beginning: As various musical ideas are presented in this Haskore tutorial, I urge the reader to question the design decisions that are made. There is no supreme theory of music that dictates my decisions, and what I present is actually one of several versions that I have developed over the years (this version is much richer than the one described in [HMGW96]; it is the "Haskore in practice" version alluded to in Section 4.2 of that paper). I believe that this version is suitable for many practical purposes, but the reader may wish to modify it to better satisfy her intuitions and/or application.

3 Creation of Music

3.1 Composing Music

3.1.1 Pitch

Perhaps the most basic musical idea is that of a *pitch*, which consists of an *octave* and a *pitch class* (i.e. one of 12 semi-tones, cf. Section C):

```
module Haskore.Basic.Pitch where

import Data.Ix(Ix)

type T      = (Octave, Class)
data Class  = Cf | C | Cs | Df | D | Ds | Ef | E | Es | Ff | F | Fs
            | Gf | G | Gs | Af | A | As | Bf | B | Bs
```

A_2	`(-3,A)`	27.5 Hz
A_1	`(-2,A)`	55.0 Hz
A	`(-1,A)`	110.0 Hz
a	`( 0,A)`	220.0 Hz
a^1	`( 1,A)`	440.0 Hz
a^2	`( 2,A)`	880.0 Hz

Figure 2: Note names, Haskore representations and frequencies.

```
    deriving (Eq,Ord,Ix,Enum,Show,Read)
type Octave = Int
```

So a `Pitch.T` is a pair consisting of a pitch class and an octave. Octaves are just integers, but we define a datatype for pitch classes, since distinguishing enharmonics (such as $G^{\#}$ and A^{b}) may be important (especially for notation). Figure 2 shows the meaning of the some `Pitch.T` values.

Treating pitches simply as integers is useful in many settings, so let's also define some functions for converting between `Pitch.T` values and `Pitch.Absolute` values (integers):

```
type Absolute = Int
type Relative = Int

toInt :: T -> Absolute
toInt (oct,pc) = 12*oct + classToInt pc

fromInt :: Absolute -> T
fromInt ap =
   let (oct, n) = divMod ap 12
   in  (oct, [C,Cs,D,Ds,E,F,Fs,G,Gs,A,As,B] !! n)

classToInt :: Class -> Relative
classToInt pc = case pc of
     Cf -> -1;  C ->  0; Cs ->  1   -- or should Cf be 11?
     Df ->  1;  D ->  2; Ds ->  3
     Ef ->  3;  E ->  4; Es ->  5
     Ff ->  4;  F ->  5; Fs ->  6
     Gf ->  6;  G ->  7; Gs ->  8
     Af ->  8;  A ->  9; As -> 10
     Bf -> 10;  B -> 11; Bs -> 12   -- or should Bs be 0?
```

Now two functions for parsing and formatting pitch classes in a more human way, that is using '#' and 'b' suffixes instead of 's' and 'f'. We do not simply use

```
classParse :: ReadS Class
classParse (p:'#':r) = reads (p:'s':r)
classParse (p:'b':r) = reads (p:'f':r)
classParse r = reads r

classFormat :: Class -> ShowS
classFormat pc =
   let (p:r) = show pc
```

```
   in  (p:) .
       case r of
          [] -> id
          's':[] -> ('#':)
          'f':[] -> ('b':)
          _ -> error ("classFormat: Pitch.Class.show must not return suffixes" ++
                      " other than 's' and 'f'")
```

Using `Pitch.Absolute` we can compute the frequency associated with a pitch:

```
intToFreq :: Floating a => Absolute -> a
intToFreq ap = 440 * 2 ** (fromIntegral (ap - toInt (1,A)) / 12)
```

We can also define a function `Pitch.transpose`, which transposes pitches (analogous to `Music.transpose`, which transposes values of type `Music.T`):

```
transpose :: Relative -> T -> T
transpose i p = fromInt (toInt p + i)
```

1 Exercise. *Show that* `toInt . fromInt = id`*, and, up to enharmonic equivalences,* `fromInt . toInt = id`.

2 Exercise. *Show that* `transpose i (transpose j p) = transpose (i+j) p`.

3.1.2 Music

```
module Haskore.Music where

import qualified Haskore.Basic.Pitch    as Pitch
import qualified Haskore.Basic.Duration as Duration

import qualified Medium.Temporal as Temporal
import qualified Medium.Controlled as CtrlMedium
import qualified Medium.Controlled.List as CtrlMediumList
import qualified Medium
import Medium (prim, serial, parallel)

import Haskore.General.Utility (maximum0, )
import Data.Tuple.HT (mapPair, mapSnd, )
import Data.Maybe.HT (toMaybe, )
import Data.Maybe (isJust, )
import qualified Data.List as List
```

Melodies consist essentially of the musical atoms notes and rests.

```
type Dur = Duration.T

type Atom note = Maybe note
```

If the atom is `Nothing` then it means a rest, if it is `Just` it contains a note. A note is described by its pitch and a list of `NoteAttributes` (defined later). Both notes and rests have a duration of type `Dur`, which is a rational Section C. The duration is measured in ratios of whole notes.

Notes and rests along with the duration are put into the `Primitive` type.

```
data Primitive note =
        Atom Dur (Atom note) -- a note or a rest
    deriving (Show, Eq, Ord)
```

A primitive can not only be an atom but also a controller as defined below. We had to make controllers alternatives of `Atom`s because the `Medium` type doesn't support them and it would damage the beauty of `Medium` if we add it at the same level as parallel and serial compositions.

```
data Control =
        Tempo      DurRatio        -- scale the tempo
      | Transpose  Pitch.Relative  -- transposition
      | Player     PlayerName      -- player label
      | Phrase     PhraseAttribute -- phrase attribute
    deriving (Show, Eq, Ord)

type DurRatio   = Dur
type PlayerName = String

atom :: Dur -> Atom note -> T note
atom d' = prim . Atom d'
control :: Control -> T note -> T note
control ctrl = CtrlMedium.control ctrl

mkControl :: (a -> Control) -> (a -> T note -> T note)
mkControl ctrl = control . ctrl
changeTempo :: DurRatio -> T note -> T note
changeTempo = mkControl Tempo
transpose :: Pitch.Relative -> T note -> T note
transpose = mkControl Transpose
setPlayer :: PlayerName -> T note -> T note
setPlayer = mkControl Player
phrase :: PhraseAttribute -> T note -> T note
phrase = mkControl Phrase
```

- `changeTempo a m` scales the rate at which `m` is played (i.e. its tempo) by a factor of `a`.
- `transpose i m` transposes `m` by interval `i` (in semitones).
- `setPlayer pname m` declares that `m` is to be performed by player `pname`.
- `phrase pa m` declares that `m` is to be played using the phrase attribute (described later) `pa`. (cf. Section C)

From these primitives we can build more complex musical objects. They are captured by the `Music.T` datatype: [4]

```
type T note = CtrlMediumList.T Control (Primitive note)

infixr 7 +:+  {- like multiplication -}
infixr 6 =:=  {- like addition -}
```

[4] I prefer to call these "musical objects" rather than "musical values" because the latter may be confused with musical aesthetics.

```
-- make them visible for importers of Music
(+:+), (=:=) :: T note -> T note -> T note
(+:+) = (Medium.+:+)
(=:=) = (Medium.=:=)
```

- Musical objects can be composed sequentially by `Medium.serial` or by `(+:+)`. That is both `serial [m0, m1]` and `m0 +:+ m1` denote that `m0` and `m1` are played in sequence. (cf. Section C)
- Similarly `Medium.parallel` and `(=:=)` compose parallely. E.g. both `parallel [m0, m1]` and `m0 =:= m1` mean that `m0` and `m1` are played simultaneously.

It is convenient to represent these ideas in Haskell as a recursive datatype rather then simple function calls because we wish to not only construct musical objects, but also take them apart, analyze their structure, print them in a structure-preserving way, interpret them for performance purposes, etc. Nonetheless using functions that are mapped to constructors has the advantage that song descriptions can stay independent from a particular music data structure.

3.1.3 Duration

```
module Haskore.Basic.Duration where

import qualified Medium.Temporal as TemporalMedium
import Data.Ratio((%))

import qualified Haskore.General.Utility as Utility
import Haskore.General.Map (Map)
import qualified Haskore.General.Map as Map

import qualified Numeric.NonNegative.Wrapper as NonNeg
```

```
type T = TemporalMedium.Dur
type Ratio  = T
type Offset = Rational

infixl 7 %+
(%+) :: Integer -> Integer -> T
(%+) x y = fromRatio (x%y)

fromRatio :: Rational -> T
fromRatio = NonNeg.fromNumberMsg "Duration.fromRatio"

toRatio :: T -> Rational
toRatio = NonNeg.toNumber

toNumber :: Fractional a => T -> a
toNumber = fromRational . NonNeg.toNumber

scale :: Ratio -> T -> T
```

```
scale = (*)

add :: Offset -> T -> T
add d = NonNeg.fromNumberMsg "Duration.add" . (d+) . toRatio
```

`add` may have undefined result.

```
divide :: T -> T -> Integer
divide r1 r2 = Utility.divide (toRatio r1) (toRatio r2)

divisible :: T -> T -> Bool
divisible r1 r2 = Utility.divisible (toRatio r1) (toRatio r2)

gcd :: T -> T -> T
gcd r1 r2 = fromRatio (Utility.gcdDur (toRatio r1) (toRatio r2))
```

```
dotted, doubleDotted :: T -> T
dotted       = ((3%+2) *)
doubleDotted = ((7%+4) *)

bn, wn, hn, qn, en, sn, tn, sfn    :: T
dwn, dhn, dqn, den, dsn, dtn       :: T
ddhn, ddqn, dden                   :: T

bn   = 2       -- brevis
wn   = 1       -- whole note
hn   = 1%+ 2    -- half note
qn   = 1%+ 4    -- quarter note
en   = 1%+ 8    -- eight note
sn   = 1%+16    -- sixteenth note
tn   = 1%+32    -- thirty-second note
sfn  = 1%+64    -- sixty-fourth note

dwn  = dotted wn    -- dotted whole note
dhn  = dotted hn    -- dotted half note
dqn  = dotted qn    -- dotted quarter note
den  = dotted en    -- dotted eighth note
dsn  = dotted sn    -- dotted sixteenth note
dtn  = dotted tn    -- dotted thirty-second note

ddhn = doubleDotted hn  -- double-dotted half note
ddqn = doubleDotted qn  -- double-dotted quarter note
dden = doubleDotted en  -- double-dotted eighth note
```

```
nameDictionary :: Map T String
nameDictionary =
   let names  = "b" : "w" :  "h" :  "q" :  "e" :  "s" :  "t" :  "sf" : []
       durs   = zip (iterate (/2) 2) names
       ddurs  = map (\(d,s) -> (dotted       d, "d" ++s)) durs
       dddurs = map (\(d,s) -> (doubleDotted d, "dd"++s)) durs
   in  Map.fromList $
          durs ++
          take 6 (drop 1 ddurs) ++
          take 3 (drop 2 dddurs)
```

```
{- |
Converts @1%4@ to @\"qn\"@ and so on.
-}
toString :: T -> String
toString dur =
   maybe
      ("(" ++ show dur ++ ")")
      (++"n")
      (Map.lookup nameDictionary dur)
```

Check proper formatting.

```
propToString :: Bool
propToString =
   all (\(dur,name) -> toString dur == name) $
     (bn, "bn") : (wn, "wn") : (hn, "hn") : (qn, "qn") :
     (en, "en") : (sn, "sn") : (tn, "tn") : (sfn, "sfn") :
     (dwn, "dwn") : (dhn, "dhn") : (dqn, "dqn") :
     (den, "den") : (dsn, "dsn") : (dtn, "dtn") :
     (ddhn, "ddhn") : (ddqn, "ddqn") : (dden, "dden") : []
```

3.1.4 Rests

3.1.5 Some Simple Examples

With this modest beginning, we can already express quite a few musical relationships simply and effectively.

Lines and Chords. Two common ideas in music are the construction of notes in a horizontal fashion (a *line* or *melody*), and in a vertical fashion (a *chord*):

```
line, chord :: [T note] -> T note
line  = serial
chord = parallel
```

Delay and Repeat. Suppose now that we wish to describe a melody `m` accompanied by an identical voice a perfect 5th higher. In Haskore we simply write "`m =:= transpose 7 m`". Similarly, a canon-like structure involving `m` can be expressed as "`m =:= delay d m`", where:

```
delay :: Dur -> T note -> T note
delay d' m = if d' == 0 then m else rest d' +:+ m
```

Of course, Haskell's non-strict semantics also allows us to define infinite musical objects. For example, a musical object may be repeated *ad nauseum* using this simple function:

```
repeat :: T note -> T note
repeat m = line (List.repeat m)
```

```
rest :: Dur -> T note
rest d' = prim (Atom d' Nothing)

bnr, wnr, hnr, qnr, enr, snr, tnr, sfnr :: T note
dwnr, dhnr, dqnr, denr, dsnr, dtnr      :: T note
ddhnr, ddqnr, ddenr                     :: T note

bnr   = rest Duration.bn     -- brevis rest
wnr   = rest Duration.wn     -- whole note rest
hnr   = rest Duration.hn     -- half note rest
qnr   = rest Duration.qn     -- quarter note rest
enr   = rest Duration.en     -- eight note rest
snr   = rest Duration.sn     -- sixteenth note rest
tnr   = rest Duration.tn     -- thirty-second note rest
sfnr  = rest Duration.sfn    -- sixty-fourth note rest

dwnr  = rest Duration.dwn    -- dotted whole note rest
dhnr  = rest Duration.dhn    -- dotted half note rest
dqnr  = rest Duration.dqn    -- dotted quarter note rest
denr  = rest Duration.den    -- dotted eighth note rest
dsnr  = rest Duration.dsn    -- dotted sixteenth note rest
dtnr  = rest Duration.dtn    -- dotted thirty-second note rest

ddhnr = rest Duration.ddhn  -- double-dotted half note rest
ddqnr = rest Duration.ddqn  -- double-dotted quarter note rest
ddenr = rest Duration.dden  -- double-dotted eighth note rest
```

Figure 3: Convenient rest definitions.

Thus an infinite ostinato can be expressed in this way, and then used in different contexts that extract only the portion that's actually needed.

A limitted loop can be defined the same way.

```
replicate :: Int -> T note -> T note
replicate n m = line (List.replicate n m)
```

Determining Duration It is sometimes desirable to compute the duration in beats of a musical object; we can do so as follows:

```
dur :: T note -> Dur
dur = Temporal.dur

instance Temporal.C (Primitive note) where
   dur (Atom d' _) = d'
   none d' = Atom d' Nothing

instance Temporal.Control Control where
   controlDur (Tempo t) d' = d' / t
   controlDur  _        d' = d'
   anticontrolDur (Tempo t) d' = d' * t
   anticontrolDur  _        d' = d'
```

However, this measurement ignores the temporal effects of phrases like ritardando.

Super-retrograde. Using `dur` we can define a function `reverse` that reverses any `Music.T` value (and is thus considerably more useful than `retro` defined earlier). Note the tricky treatment of parallel compositions. Also note that this version wastes time. It computes the duration of smaller structures in the case of parallel compositions. When it descends into a structure of which it has computed the duration it computes the duration of its sub-structures again. This can lead to a quadratic time consumption.

```
reverse :: T note -> T note
reverse = mapList
   (,)
   (flip const)
   List.reverse
   (\ms -> let durs = map dur ms
               dmax = maximum0 durs
           in  zipWith (delay . (dmax -)) durs ms)
```

Truncating Parallel Composition Note that the duration of `m0 =:= m1` is the maximum of the durations of
codem0 and `m1` (and thus if one is infinite, so is the result). Sometimes we would rather have the result be of duration equal to the shorter of the two. This is not as easy as it sounds, since it may require interrupting the longer one in the middle of a note (or notes).

We will define a "truncating parallel composition" operator `(/=:)`, but first we will define an auxiliary function `Music.take` such that `Music.take d m` is the musical object `m` "cut short" to have at most duration `d`. The name matches the one of the module `List` because the function is quite similar.

```
take :: Dur -> T note -> T note
take newDur m =
   if newDur < 0
   then error ("Music.take: newDur " ++ show newDur ++ " must be non-negative")
   else snd (take' newDur m)

takeLine :: Dur -> [T note] -> [T note]
takeLine newDur = snd . takeLine' newDur

take' :: Dur -> T note -> (Dur, T note)
take' 0 = const (0, rest 0)
take' newDur =
   switchList
      (\oldDur at -> let takenDur = min oldDur newDur
                     in (takenDur, atom takenDur at))
      (\ctrl -> case ctrl of
          Tempo t -> mapPair ((/t), changeTempo t) .
                               take' (newDur * t)
          _       -> mapSnd  (control ctrl) .
                               take' newDur)
      (mapSnd line . takeLine' newDur)
      (mapPair (maximum0,chord) . unzip . map (take' newDur))

takeLine' :: Dur -> [T note] -> (Dur, [T note])
takeLine' 0 _  = (0, [])
takeLine' _ [] = (0, [])
takeLine' newDur (m:ms) =
   let m'  = take' newDur m
       ms' = takeLine' (newDur - fst m') ms
   in  (fst m' + fst ms', snd m' : snd ms')
```

Note that `Music.take` is ready to handle a `Music.T` object of infinite length. The implementation of `takeLine'` and `take'` would be simpler if one does not compute the duration of the taken part of the music in `take'`. Instead one could compute the duration of the taken part where it is needed, i.e. after `takeLine'` calls `Music.take'`. The drawback of this simplification would be analogously to `Music.reverse`: The duration of sub-structures must be computed again and again, which results in quadratic runtime in the worst-case.

With `Music.take`, the definition of `(/=:)` is now straightforward:

```
(/=:) :: T note -> T note -> T note
m0 /=: m1 = Haskore.Music.take (min (dur m0) (dur m1)) (m0 =:= m1)
```

Unfortunately, whereas `Music.take` can handle infinite-duration music values, `(/=:)` cannot.

3 Exercise. *Define a version of* `(/=:)` *that shortens correctly when either or both of its arguments are infinite in duration.*

For completeness we want to define a function somehow dual to `Music.take`. The `Music.drop` removes a prefix of the given duration from the music. Notes that begin in the removed part are lost. This is especially important for notes which start in the removed part and end in the remainder. They are replaced by rests.

We would like to design `drop'` such that it returns the duration of the remaining music. This design fails for infinite music. Thus we return the duration of the part that was dropped. When going through a serial composition, if we could drop less from a music item than we wanted then the music item must have been gone completely and must drop subsequent items. If we dropped as much as we wanted we are ready. If we dropped more than we wanted this indicates an error. Remaining rests of zero duration, empty compositions and so on may be removed by subsequent optimizations.

```
drop :: Dur -> T note -> T note
drop remDur =
   if remDur < 0
   then error ("Music.drop: remDur " ++ show remDur ++ " must be non-negative")
   else snd . drop' remDur

dropLine :: Dur -> [T note] -> [T note]
dropLine remDur = snd . dropLine' remDur

drop' :: Dur -> T note -> (Dur, T note)
drop' 0 = (,) 0
drop' remDur =
   switchList
      (\oldDur _ -> let newDur = min oldDur remDur
                    in (newDur, rest (oldDur-newDur)))
      (\ctrl -> case ctrl of
          Tempo t -> mapPair ((/t), changeTempo t) .
                              drop' (remDur * t)
          _       -> mapSnd (control ctrl) .
                              drop' remDur)
      (mapSnd line . dropLine' remDur)
      (mapPair (maximum0,chord) . unzip . map (drop' remDur))

dropLine' :: Dur -> [T note] -> (Dur, [T note])
dropLine' 0 m  = (0, m)
dropLine' _ [] = (0, [])
dropLine' remDur (m:ms) =
   let (dropped, m') = drop' remDur m
   in  case compare dropped remDur of
         LT -> mapPair ((dropped+), id) (dropLine' (remDur - dropped) ms)
         EQ -> (dropped, m' : ms)
         GT -> error "dropLine': program error: dropped more than we wanted"
```

Note that `mapPair` is prepared for infinite lists.

We will now define functions for filtering out notes. This way you can e.g. extract all notes for a particular instrument. Non-matching notes are replaced by rests. You may want to merge them using `Optimization.rest`.

```
filter :: (note -> Bool) -> T note -> T note
filter p =
   fmap (\(Atom d' mn) -> Atom d' (mn >>= \n -> toMaybe (p n) n))
--   fmap (\(Atom d' mn) -> Atom d' (listToMaybe $ filter p $ maybeToList mn))

partition :: (note -> Bool) -> T note -> (T note, T note)
partition p =
   foldList
```

```
      (\ d' mn ->
          mapPair
             (atom d', atom d')
             (if maybe False p mn
                then (mn, Nothing)
                else (Nothing, mn)))
      (\k -> mapPair (control k, control k))
      (mapPair (line,  line)  . unzip)
      (mapPair (chord, chord) . unzip)

partitionMaybe :: (noteA -> Maybe noteB) -> T noteA -> (T noteB, T noteA)
partitionMaybe f =
   foldList
      (\ d' mn ->
          mapPair
             (atom d', atom d')
             (let m = mn >>= f
              in  if isJust m
                    then (m, Nothing)
                    else (Nothing, mn)))
      (\k -> mapPair (control k, control k))
      (mapPair (line,  line)  . unzip)
      (mapPair (chord, chord) . unzip)
```

Inspecting a `Music.T` Here are some routines which specialize functions from module `Medium` to module `Music`.

```
applyPrimitive ::
   (Dur -> Atom note -> b) ->
   Primitive note -> b
applyPrimitive fa (Atom d' at) = fa d' at

switchBinary ::
   (Dur -> Atom note -> b) ->
   (Control -> T note -> b) ->
   (T note -> T note -> b) ->
   (T note -> T note -> b) ->
   b -> T note -> b
switchBinary fa fc fser fpar =
   CtrlMedium.switchBinary (applyPrimitive fa) fser fpar fc

switchList ::
   (Dur -> Atom note -> b) ->
   (Control -> T note -> b) ->
   ([T note] -> b) ->
   ([T note] -> b) ->
   T note -> b
switchList fa fc fser fpar =
   CtrlMedium.switchList (applyPrimitive fa) fser fpar fc

foldBin ::
   (Dur -> Atom note -> b) ->
   (Control -> b -> b) ->
   (b -> b -> b) ->
```

```
   (b -> b -> b) ->
   b -> T note -> b
foldBin fa fc fser fpar none' =
   CtrlMedium.foldBin (applyPrimitive fa) fser fpar fc none'

foldList ::
   (Dur -> Atom note -> b) ->
   (Control -> b -> b) ->
   ([b] -> b) ->
   ([b] -> b) ->
   T note -> b
foldList fa fc fser fpar =
   CtrlMedium.foldList (applyPrimitive fa) fser fpar fc

mapListFlat ::
   (Dur -> Atom noteA -> (Dur, Atom noteB)) ->
   (Control -> T noteA -> T noteB) ->
   ([T noteA] -> [T noteB]) ->
   ([T noteA] -> [T noteB]) ->
   T noteA -> T noteB

mapListFlat fa fc fser fpar =
   CtrlMediumList.mapListFlat (uncurry Atom . applyPrimitive fa) fser fpar fc

mapList ::
   (Dur -> Atom noteA -> (Dur, Atom noteB)) ->
   (Control -> T noteB -> T noteB) ->
   ([T noteB] -> [T noteB]) ->
   ([T noteB] -> [T noteB]) ->
   T noteA -> T noteB

mapList fa fc fser fpar =
   CtrlMediumList.mapList (uncurry Atom . applyPrimitive fa) fser fpar fc

-- Could be an instance of fmap if Music.T would be an algebraic type.
mapNote :: (noteA -> noteB) -> T noteA -> T noteB
mapNote f' = fmap (\(Atom d' at) -> Atom d' (fmap f' at))

{-
This is useful for duration dependend attributes,
and duration dependend instrument sounds.
However it seems to be more appropriate to pass the duration in seconds
to the sound generators rather than the relative duration.
-}
mapDurNote :: (Dur -> noteA -> noteB) -> T noteA -> T noteB
mapDurNote f' = fmap (\(Atom d' at) -> Atom d' (fmap (f' d') at))
```

3.1.6 Trills

```
module Haskore.Composition.Trill where

import qualified Haskore.Music as Music
```

A *trill* is an ornament that alternates rapidly between two (usually adjacent) pitches. Let's implement a trill as a function that take a note as an argument and returns a series of notes whose durations add up to the same duration as as the given note.

A trill alternates between the given note and another note, usually the note above it in the scale. Therefore, it must know what other note to use. So that the structure of `trill` remains parallel across different keys, we'll implement the other note in terms of its interval from the given note in half steps. Usually, the note is either a half-step above (interval = 1) or a whole-step above (interval = 2). Using negative numbers, a trill that goes to lower notes can even be implemented.

Also, the trill needs to know how fast to alternate between the two notes. One way is simply to specify the type of smaller note to use. (Another implementation will be discussed later.) So, our `trill` has the following type:

```
trill :: Int -> Music.Dur -> Music.T note -> Music.T note
```

Its implementation:

```
trill i d m =
   let atom = Music.take d m
   in  Music.line (Music.takeLine (Music.dur m)
          (cycle [atom, Music.transpose i atom]))
```

Since the function uses `Music.tranpose` one can even trill more complex objects like chords.

The next version of `trill` starts on the second note, rather than the given note. It is simple to define a function that starts on the other note:

```
trill' :: Int -> Music.Dur -> Music.T note -> Music.T note
trill' i sDur m =
      trill (negate i) sDur (Music.transpose i m)
```

Another way to define a trill is in terms of the number of subdivided notes to be included in the trill.

```
trillN :: Int -> Integer -> Music.T note -> Music.T note
trillN i nTimes m =
      trill i (Music.dur m / fromIntegral nTimes) m
```

This, too, can be made to start on the other note.

```
trillN' :: Int -> Integer -> Music.T note -> Music.T note
trillN' i nTimes m =
      trillN (negate i) nTimes (Music.transpose i m)
```

Finally, a `roll` can be implemented as a trill whose interval is zero. This feature is particularly useful for percussion.

```
roll  :: Music.Dur -> Music.T note -> Music.T note
rollN :: Integer   -> Music.T note -> Music.T note

roll  d      = trill  0 d
rollN nTimes = trillN 0 nTimes
```

3.1.7 Percussion

Percussion is a difficult notion to represent in the abstract, since in a way, a percussion instrument is just another instrument, so why should it be treated differently? On the other hand, even common practice notation treats it specially, even though it has much in common with non-percussive notation. The midi standard is equally ambiguous about the treatment of percussion: on one hand, percussion sounds are chosen by specifying an octave and pitch, just like any other instrument, on the other hand these notes have no tonal meaning whatsoever: they are just a convenient way to select from a large number of percussion sounds. Indeed, part of the General Midi Standard is a set of names for commonly used percussion sounds.

```
module Haskore.Composition.Drum
   (T, GM.Drum(..), Element(..), na,
    toMusic, toMusicDefaultAttr,
    lineToMusic, elementToMusic, funkGroove) where

import Haskore.Composition.Trill
import qualified Haskore.Basic.Duration as Duration
import Haskore.Basic.Duration (qn, en, )
import Haskore.Music (qnr, enr, (=:=), changeTempo, rest, )
import Haskore.Melody.Standard (NoteAttributes, na, )

import qualified Haskore.Music             as Music
import qualified Haskore.Music.GeneralMIDI as MidiMusic
import qualified Haskore.Music.Rhythmic    as RhyMusic
import qualified Sound.MIDI.General as GM

type T = GM.Drum
```

Since Midi is such a popular platform, we can at least define some handy functions for using the General Midi Standard. We start by defining the datatype shown in Figure **??**, which borrows its constructor names from the General Midi standard. The comments reflecting the "Midi Key" numbers will be explained later, but basically a Midi Key is the equivalent of an absolute pitch in Haskore terminology. We will not adapt the MIDI treatment of drums in Haskore since it makes no sense, e.g. to transpose drums by increasing the key number. Thus we defined a special constructor for drums in `RhyMusic.T`. We will now give a function which constructs a `RhyMusic.T` for a given value specifying a drum:

```
toMusic :: drum -> Duration.T -> NoteAttributes -> RhyMusic.T drum instr
toMusic drm dr nas =
   Music.atom dr (Just (RhyMusic.noteFromAttrs nas (RhyMusic.Drum drm)))

toMusicDefaultAttr ::
   drum -> Duration.T -> RhyMusic.T drum instr
toMusicDefaultAttr drm dr = toMusic drm dr na
```

For example, here are eight bars of a simple rock or "funk groove" that uses `Drum.toMusic` and `Drum.roll`:

```
funkGroove :: MidiMusic.T
funkGroove =
    let p1 = toMusic GM.LowTom        qn na
        p2 = toMusic GM.AcousticSnare en na
    in changeTempo 3 (Music.take 8 (Music.repeat
```

```
         ( (Music.line [p1, qnr, p2, qnr, p2,
                        p1, p1, qnr, p2, enr])
           =:= roll en (toMusic GM.ClosedHiHat 2 na) )
       ))
```

We can go one step further by defining our own little "percussion datatype":

```
data Element =
     R               Duration.T                -- rest
   | N               Duration.T NoteAttributes -- note
   | Roll  Duration.T Duration.T NoteAttributes -- roll w/duration
   | Rolln Integer    Duration.T NoteAttributes -- roll w/number of strokes

lineToMusic :: T -> [Element] -> MidiMusic.T
lineToMusic dsnd =
   Music.line . map (elementToMusic dsnd)

elementToMusic :: T -> Element -> MidiMusic.T
elementToMusic dsnd el =
   let drum = toMusic dsnd
   in  case el of
         R            dur     -> rest dur
         N            dur nas -> drum dur nas
         Roll  sDur   dur nas -> roll sDur (drum dur nas)
         Rolln nTimes dur nas -> rollN nTimes (drum dur nas)
```

3.1.8 Phrasing and Articulation

The `Phrase` constructor permits one to annotate an entire musical object with a `PhraseAttribute`. This attribute datatype covers a wide range of attributions found in common practice notation, and is shown in Figure 4. Beware that use of them requires the use of a player that knows how to interpret them! Players will be described in more detail in Section 3.3.

Again, to stay independent from the underlying data structure we define some functions that simplify the application of several phrases.

```
dynamic :: Dynamic -> T note -> T note
dynamic = phrase . Dyn

tempo :: Tempo -> T note -> T note
tempo = phrase . Tmp

articulation :: Articulation -> T note -> T note
articulation = phrase . Art

ornament :: Ornament -> T note -> T note
ornament = phrase . Orn


accent, crescendo, diminuendo, loudness1,
   ritardando, accelerando ::
      Rational -> T note -> T note
```

```
data PhraseAttribute = Dyn Dynamic
                     | Tmp Tempo
                     | Art Articulation
                     | Orn Ornament
     deriving (Eq, Ord, Show)

data Dynamic = Loudness Rational | Accent Rational
             | Crescendo Rational | Diminuendo Rational
     deriving (Eq, Ord, Show)

data Tempo = Ritardando Rational | Accelerando Rational
     deriving (Eq, Ord, Show)

data Articulation = Staccato Dur | Legato Dur | Slurred Dur
                  | Tenuto | Marcato | Pedal | Fermata | FermataDown | Breath
                  | DownBow | UpBow | Harmonic | Pizzicato | LeftPizz
                  | BartokPizz | Swell | Wedge | Thumb | Stopped
     deriving (Eq, Ord, Show)

data Ornament = Trill | Mordent | InvMordent | DoubleMordent
              | Turn | TrilledTurn | ShortTrill
              | Arpeggio | ArpeggioUp | ArpeggioDown
              | Instruction String | Head NoteHead
     deriving (Eq, Ord, Show)

-- this is more a note attribute than a phrase attribute
data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
              | TremoloHead | SlashHead | ArtHarmonic | NoHead
     deriving (Eq, Ord, Show)
```

Figure 4: Note and Phrase Attributes.

```
accent     = dynamic . Accent
crescendo  = dynamic . Crescendo
diminuendo = dynamic . Diminuendo
loudness1  = dynamic . Loudness

ritardando  = tempo . Ritardando
accelerando = tempo . Accelerando

staccato, legato :: Dur -> T note -> T note

staccato = articulation . Staccato
legato   = articulation . Legato
```

Note that some of the attributes are parameterized with a numeric value. This is used by a player to control the degree to which an articulation is to be applied. For example the articulations `Staccato`, `Legato`, `Slurred` describe the overlapping between notes. We would expect `Legato 1.2` to create more of a legato feel than `Legato 1.1`, and `Staccato 2` to be stronger than `Staccato 1`.

The following constants represent default values for some of the parameterized attributes:

```
defltLegato, defltStaccato,
  defltAccent, bigAccent :: T note -> T note

defltLegato    = legato   Duration.sn
defltStaccato  = staccato Duration.sn
defltAccent    = accent 1.2
bigAccent      = accent 1.5
```

To understand exactly how a player interprets an attribute requires knowing how players are defined. Haskore defines only a few simple players, so in fact many of the attributes in Figure 4 are to allow the user to give appropriate interpretations of them by her particular player. But before looking at the structure of players we will need to look at the notion of a *performance* (these two ideas are tightly linked, which is why the `Player` and `Performance` modules are mutually recursive).

4 Exercise. *Find a simple piece of music written by your favorite composer, and transcribe it into Haskore. In doing so, look for repeating patterns, transposed phrases, etc. and reflect this in your code, thus revealing deeper structural aspects of the music than that found in common practice notation.*

Section B.2 shows the first 28 bars of Chick Corea's "Children's Song No. 6" encoded in Haskore.

3.1.9 Intervals

In music theory, an interval is the difference (a ratio or logarithmic measure) in pitch between two notes and often refers to those two notes themselves (otherwise known as a dyad).

Here we list some common names for some possible intervals.

```
module Haskore.Basic.Interval where

unison, minorSecond, majorSecond, minorThird, majorThird,
 fourth, fifth, minorSixth, majorSixth, minorSeventh, majorSeventh,
```

```
 octave, octaveMinorSecond, octaveMajorSecond, octaveMinorThird,
 octaveMajorThird, octaveFourth, octaveFifth, octaveMinorSixth,
 octaveMajorSixth, octaveMinorSeventh, octaveMajorSeventh :: Integral a => a
unison       =  0
minorSecond  =  1
majorSecond  =  2
minorThird   =  3
majorThird   =  4
fourth       =  5
fifth        =  7
minorSixth   =  8
majorSixth   =  9
minorSeventh = 10
majorSeventh = 11
octave       = 12
octaveMinorSecond  = octave + minorSecond
octaveMajorSecond  = octave + majorSecond
octaveMinorThird   = octave + minorThird
octaveMajorThird   = octave + majorThird
octaveFourth       = octave + fourth
octaveFifth        = octave + fifth
octaveMinorSixth   = octave + minorSixth
octaveMajorSixth   = octave + majorSixth
octaveMinorSeventh = octave + minorSeventh
octaveMajorSeventh = octave + majorSeventh
```

3.1.10 Chords

Earlier I described how to represent chords as values of type `Music.T`. However, sometimes it is convenient to treat chords more abstractly. Rather than think of a chord in terms of its actual notes, it is useful to think of it in terms of its chord "quality", coupled with the key it is played in and the particular voicing used. For example, we can describe a chord as being a "major triad in root position, with root middle C". Several approaches have been put forth for representing this information, and we cannot cover all of them here. Rather, I will describe two basic representations, leaving other alternatives to the skill and imagination of the reader.[5]

First, one could use a *pitch* representation, where each note is represented as its distance from some fixed pitch. `0` is the obvious fixed pitch to use, and thus, for example, `[0,4,7]` represents a major triad in root position. The first zero is in some sense redundant, of course, but it serves to remind us that the chord is in "normal form". For example, when forming and transforming chords, we may end up with a representation such as `[2,6,9]`, which is not normalized; its normal form is in fact `[0,4,7]`. Thus we define:

> A chord is in *pitch normal form* if the first pitch is zero, and the subsequent pitches are monotonically increasing.

One could also represent a chord *intervalically*; i.e. as a sequence of intervals. A major triad in root position, for example, would be represented as `[4,3,-7]`, where the last interval "returns" us to the

[5] For example, Forte prescribes normal forms for chords in an atonal setting [For73].

"origin". Like the `0` in the pitch representation, the last interval is redundant, but allows us to define another sense of normal form:

> A chord is in *interval normal form* if the intervals are all greater than zero, except for the last which must be equal to the negation of the sum of the others.

In either case, we can define a chord type as:

```
module Haskore.Composition.Chord where

import qualified Haskore.Music  as Music
import qualified Haskore.Melody as Melody
import qualified Haskore.Basic.Pitch    as Pitch
import qualified Haskore.Basic.Interval as I
import           Haskore.General.Utility (foldrf, )
import           Data.Ord.HT (comparing, )
import           Data.List.HT (viewR, )
import           Data.List (genericLength, minimumBy, )

type T = [Pitch.Relative]
```

We might ask whether there is some advantage, computationally, of using one of these representations over the other. However, there is an invertible linear transformation between them, as defined by the following functions, and thus there is in fact little advantage of one over the other:

```
pitchToInterval :: T -> T
pitchToInterval [] = error "pitchToInterval: Chord must be non-empty."
pitchToInterval ch@(p:ps) =
   zipWith (-) (ps++[p]) ch

intervalToPitch :: T -> T
intervalToPitch [] = error "intervalToPitch: Chord must be non-empty."
intervalToPitch ch =
   let Just (chInit, chLast) = viewR (scanl (+) 0 ch)
   in  if chLast==0
         then chInit
         else error "intervalToPitch: intervals do not sum-up to zero."
```

5 Exercise. *Show that* `pitchToInterval` *and* `intervalToPitch` *are* inverses *in the following sense: for any chord* `ch1` *in pitch normal form, and* `ch2` *in interval normal form, each of length at least two:*

```
intervalToPitch (pitchToInterval ch1) = ch1
pitchToInterval (intervalToPitch ch2) = ch2
```

Another operation we may wish to perform is a test for *equality* on chords, which can be done at many levels: based only on chord quality, taking inversion into account, absolute equality, etc. Since the above normal forms guarantee a unique representation, equality of chords with respect to chord quality and inversion is simple: it is just the standard (overloaded) equality operator on lists. On the other hand, to measure equality based on chord quality alone, we need to account for the notion of an *inversion*.

Using the pitch representation, the inversion of a chord can be defined as follows:

```
pitchInvert, intervalInvert :: T -> T
pitchInvert (0:p2:ps) = 0 : map (subtract p2) ps ++ [12-p2]
pitchInvert _ =
   error "pitchInvert: Pitch chord representation must start with a zero."
```

Although we could also directly define a function to invert a chord given in interval representation, we will simply define it in terms of functions already defined:

```
intervalInvert = pitchToInterval . pitchInvert . intervalToPitch
```

We can now determine whether a chord in normal form has the same quality (but possibly different inversion) as another chord in normal form, as follows: simply test whether one chord is equal either to the other chord or to one of its inversions. Since there is only a finite number of inversions, this is well defined. In Haskell:

```
samePitch, sameInterval :: T -> T -> Bool
samePitch ch1 ch2 =
 let invs = take (length ch1) (iterate pitchInvert ch1)
 in  ch2 `elem` invs

sameInterval ch1 ch2 =
 let invs = take (length ch1) (iterate intervalInvert ch1)
 in  ch2 `elem` invs
```

For example, `samePitch [0,4,7] [0,5,9]` returns `True` (since `[0,5,9]` is the pitch normal form for the second inversion of `[0,4,7]`).

Here we provide a list of some common types of chords.

```
majorInt, minorInt, majorSeventhInt, minorSeventhInt,
   dominantSeventhInt, minorMajorSeventhInt,
   sustainedFourthInt :: [Pitch.Relative]

majorInt = [I.unison, I.majorThird, I.fifth]
minorInt = [I.unison, I.minorThird, I.fifth]

majorSeventhInt      = [I.unison, I.majorThird, I.fifth, I.majorSeventh]
minorSeventhInt      = [I.unison, I.minorThird, I.fifth, I.minorSeventh]
dominantSeventhInt   = [I.unison, I.majorThird, I.fifth, I.minorSeventh]
minorMajorSeventhInt = [I.unison, I.minorThird, I.fifth, I.majorSeventh]

sustainedFourthInt = [I.unison, I.fourth, I.fifth]

type Inversion = Int

fromIntervals ::
   [Pitch.Relative] -> Inversion -> Music.T note -> [Music.T note]
fromIntervals int inv m =
   let err = error ("Chord.fromInterval: inversion number "
                    ++ show inv ++ " too large")
   in  map (flip Music.transpose m) (zipWith const
             (drop inv (init (int ++ map (12+) int) ++ repeat err)) int)
```

```
major, minor, majorSeventh, minorSeventh, dominantSeventh,
   minorMajorSeventh, sustainedFourth ::
      Inversion -> Music.T note -> [Music.T note]

major = fromIntervals majorInt
minor = fromIntervals minorInt

majorSeventh      = fromIntervals majorSeventhInt
minorSeventh      = fromIntervals minorSeventhInt
dominantSeventh   = fromIntervals dominantSeventhInt
minorMajorSeventh = fromIntervals minorMajorSeventhInt

sustainedFourth = fromIntervals sustainedFourthInt
```

We want to offer a special service: The computer shall find out inversions for chords in a sequence such that the overall pitch does not vary so much.

A very simple approach is to compute the "center" of a chord, that is the average of all pitches. We do now try to keep the center as close as possible to an overall trend. This is especially easy because for a chord of n notes the change to the next inversion moves the center of the chord by $\frac{12}{n}$ tones.

The function gets the inversion of the first and the last chord and the list of chords represented by the base note and the intervals of all notes of the chord.

```
data Generic attr = Generic {
   genericPitchClass :: Pitch.Class,
   genericIntervals  :: T,
   genericDur        :: Music.Dur,
   genericAttr       :: attr}

type Boundary = (Pitch.T, Pitch.T)

generic :: Pitch.Class -> T -> Music.Dur -> attr -> Generic attr
generic = Generic

leastVaryingInversions ::
   Boundary -> [Generic attr] -> [[Melody.T attr]]
leastVaryingInversions (begin,end) gs =
   let beginCenter = fromIntegral (Pitch.toInt begin)
       endCenter   = fromIntegral (Pitch.toInt end)
       steep = (endCenter - beginCenter) / (genericLength gs - 1)
       trend = map (\k -> beginCenter + steep * fromIntegral k)
                   [0 .. (length gs - 1)]
       invs = zipWith
                 (\g t -> round (matchingInversion g t))
                 gs trend
   in  zipWith genericToNotes invs gs

inversionIncrement :: T -> Double
inversionIncrement ps = 12 / genericLength ps

matchingInversion :: Generic attr -> Double -> Double
matchingInversion g dst =
  let c = chordCenter g
      inc = inversionIncrement (genericIntervals g)
```

```
  in  (dst-c)/inc

mean :: [Pitch.Relative] -> Double
mean ps = sum (map fromIntegral ps) / genericLength ps

chordCenter :: Generic attr -> Double
chordCenter (Generic pc ps _ _) =
   fromIntegral (Pitch.classToInt pc) + mean ps

boundaryCenter :: (Pitch.Octave,Inversion) -> Generic attr -> Double
boundaryCenter (oct,inv) g =
   12 * fromIntegral oct + chordCenter g +
      fromIntegral inv * inversionIncrement (genericIntervals g)

invert :: Inversion -> T -> T
invert inv ps =
   let (q,r) = divMod inv (length ps)
   in  zipWith (+) ps
          (replicate r (12*(q+1)) ++ repeat (12*q))

genericToNotes :: Inversion -> Generic attr -> [Melody.T attr]
genericToNotes inv (Generic pc ps dur attr) =
   map (\t -> Melody.note (Pitch.transpose t (0,pc)) dur attr)
       (invert inv ps)
```

A more complicated algorithm will also work for other definitions of variation. We compute the mean pitch for every chord and minimize the variation of the pitch. The variation is defined here as the sum of the squared differences of successive chords.

This leads to a shortest ways search in a graph where each inversion of a chord is a node and each possible neighbourhood of inversions is an edge. The nodes for the inversions of a chord and the nodes for the inversions of the succeeding chord build a complete bi-partite graph.

First we write a shortest ways search algorithm that is specialised to our problem. In each step we process one chord. We construct a list of inversions, where each inversion is associated with the optimal way from the beginning chord to this inversion and its variation. This list passed to the processing of the next chord. For reasons of simplicity we process the list backwards.

The inputs of the algorithm are a distance function and the list of concurrent inversions for each chord. The first element of the list contains all starting inversions, the last element contains all ending inversions. If you want a definitive start and end inversion, use one-element lists. The output is the list of the optimal inversion for each chord. More precisely it is a list of all optimal ways, where for each starting inversion there is one optimal way to the closest ending inversion.

```
shortestWays :: (Num b, Ord b) =>
   (a -> a -> b) -> [[a]] -> [(b,[a])]
shortestWays dist =
   foldrf (processZone dist) (map (\x->(0,[x])))

processZone :: (Num b, Ord b) =>
   (a -> a -> b) -> [a] -> [(b,[a])] -> [(b,[a])]
processZone dist srcs ways =
   let distToWay src (d,dst:_) = d + dist src dst
       distToWay _   (_,[])    =
```

```
         error "processZone: list is never empty if called from shortestWays"
   in  map (\src -> minimumBy (comparing fst)
             (map (\way -> (distToWay src way, src : snd way)) ways)) srcs

propShortestWays :: Int -> Int -> Bool
propShortestWays n k =
   let sws = shortestWays (\x y -> (x-y)^(2::Int))
                          (replicate n [0..(n*k)] ++ [[0]])
   in  head sws == (0, replicate (n+1) 0)  &&
       last sws == (n*k^(2::Int), reverse [0,k..n*k])
```

This routine could be made more efficient because the centers of the chords with different inversions are equidistant.

```
leastVaryingInversionsSW ::
   Boundary -> [Generic attr] -> [[Melody.T attr]]
leastVaryingInversionsSW bnd gs =
   let dist (_,c0) (_,c1) = (c0-c1)^(2::Int)
       [(_,invs)] =
          shortestWays dist
             (inversionCenters bnd gs)
   in  zipWith (\(inv,_) -> genericToNotes inv) invs gs

inversionCenters :: Boundary -> [Generic attr] -> [[(Inversion,Double)]]
inversionCenters (begin,end) gs =
   let margin = 7
       beginCenter = fromIntegral (Pitch.toInt begin)
       endCenter   = fromIntegral (Pitch.toInt end)
       lower = min beginCenter endCenter - margin
       upper = max beginCenter endCenter + margin
       inversions g =
          let c = chordCenter g
              inc = inversionIncrement (genericIntervals g)
              invs :: [Inversion]
              invs = [floor ((lower-c)/inc) ..
                      ceiling ((upper-c)/inc)]
          in  map (\inv -> (inv, c + inc * fromIntegral inv)) invs
       boundInv g center =
          (round (matchingInversion g center), center)
   in  [[boundInv (head gs) beginCenter]] ++
       map inversions (tail (init gs)) ++
       [[boundInv (last gs) endCenter]]
```

Now two helper functions for creating a harmonic and a melodic chord, that is chords of notes of the same length in sequentially or simultaneously.

```
melodicGen, harmonicGen :: attr -> Music.Dur ->
   [Music.Dur -> attr -> Melody.T attr] -> Melody.T attr
melodicGen  attr d = Music.line  . map (\n -> n d attr)
harmonicGen attr d = Music.chord . map (\n -> n d attr)
```

3.1.11 Scales

```
module Haskore.Basic.Scale
         (T, ionian, dorian, phrygian, lydian, mixolydian,
             aeolian, lokrian, altered, htwt, wtht,
             ionianRel, dorianRel, phrygianRel, lydianRel, mixolydianRel,
             aeolianRel, lokrianRel, alteredRel, htwtRel, wthtRel,

             fromOffsets, fromIntervals, continue) where

import qualified Haskore.Basic.Pitch as Pitch
import Control.Monad(liftM2)
```

Some of the following code is taken from the EasyScale implementation of Martin Schwenke.

```
type T = [Pitch.Absolute]
type Intervals = [Pitch.Relative]
```

Make a scale given a list of absolute pitches, usually starting at 0, and a `Pitch.Class` representing the root note of the scale.

```
fromOffsets :: [Pitch.Absolute] -> Pitch.Class -> T
fromOffsets ns pc
  = map (+ Pitch.classToInt pc) ns
```

Create a scale from a list of intervals between successive notes.

```
fromIntervals :: Intervals -> Pitch.Class -> T
fromIntervals = fromOffsets . scanl (+) 0
```

Continue a scale to all octaves.

```
continue :: T -> T
continue = liftM2 (+) (iterate (12+) 0)
```

Now some general useful scales from music theory.

```
ionianRel, dorianRel, phrygianRel, lydianRel, mixolydianRel,
  aeolianRel, lokrianRel, alteredRel, htwtRel,
  wthtRel :: Intervals

ionianRel     = [ 2, 2, 1, 2, 2, 2, 1 ]
dorianRel     = [ 2, 1, 2, 2, 2, 1, 2 ]
phrygianRel   = [ 1, 2, 2, 2, 1, 2, 2 ]
lydianRel     = [ 2, 2, 2, 1, 2, 2, 1 ]
mixolydianRel = [ 2, 2, 1, 2, 2, 1, 2 ]
aeolianRel    = [ 2, 1, 2, 2, 1, 2, 2 ]
lokrianRel    = [ 1, 2, 2, 1, 2, 2, 2 ]
alteredRel    = [ 1, 2, 1, 2, 2, 2, 2 ]
htwtRel       = [ 1, 2, 1, 2, 1, 2, 1, 2 ]
wthtRel       = [ 2, 1, 2, 1, 2, 1, 2, 1 ]

ionian, dorian, phrygian, lydian, mixolydian,
  aeolian, lokrian, altered, htwt,
  wtht :: Pitch.Class -> T
```

```
ionian     = fromIntervals ionianRel
dorian     = fromIntervals dorianRel
phrygian   = fromIntervals phrygianRel
lydian     = fromIntervals lydianRel
mixolydian = fromIntervals mixolydianRel
aeolian    = fromIntervals aeolianRel
lokrian    = fromIntervals lokrianRel
altered    = fromIntervals alteredRel
htwt       = fromIntervals htwtRel
wtht       = fromIntervals wthtRel
```

Example: Alternatively to applying `continue` to a scale you can create an infinitely increasing scale using the definition by intervals, e.g. `fromIntervals (cycle ionianRel) Pitch.C`.

3.1.12 Tempo

```
module Haskore.Basic.Tempo where

import qualified Haskore.Basic.Pitch as Pitch
import Haskore.Basic.Duration (qn, en, sn, (%+), )
import qualified Haskore.Music as Music
import Haskore.Music(changeTempo, line, (+:+), (=:=), )
import qualified Haskore.Melody as Melody

import qualified Haskore.Basic.Duration as Dur

import qualified Data.List as List
```

Set tempo. To make it easier to initialize the duration element of a `PerformanceContext.T` (see Section 3.2), we can define a "metronome" function that, given a standard metronome marking (in beats per minute) and the note type associated with one beat (quarter note, eighth note, etc.) generates the duration of one whole note:

```
metro :: Fractional a => a -> Music.Dur -> a
metro setting dur = 60 / (setting * Dur.toNumber dur)
```

Additionally we define some common tempos and some range of interpretation as in Figure 5. This means, the tempo Andante may vary between `fst andanteRange` and `snd andanteRange` beats per minute. For example, `metro andante qn` creates a tempo of 92 quarter notes per minute.

Polyrhythms. For some rhythmical ideas, consider first a simple *triplet* of eighth notes; it can be expressed as "`Tempo (3%2) m`", where `m` is a line of three eighth notes. In fact `Tempo` can be used to create quite complex rhythmical patterns. For example, consider the "nested polyrhythms" shown in Figure 6. They can be expressed quite naturally in Haskore as follows (note the use of the `where` clause in `pr2` to capture recurring phrases):

```
largoRange, larghettoRange, adagioRange, andanteRange,
  moderatoRange, allegroRange, prestoRange, prestissimoRange
    :: Fractional a => (a,a)

largoRange       = ( 40, 60) -- slowly and broadly
larghettoRange   = ( 60, 68) -- a little less slow than largo
adagioRange      = ( 66, 76) -- slowly
andanteRange     = ( 76,108) -- at a walking pace
moderatoRange    = (108,120) -- at a moderate tempo
allegroRange     = (120,168) -- quickly
prestoRange      = (168,200) -- fast
prestissimoRange = (200,208) -- very fast


largo, larghetto, adagio, andante, moderato, allegro,
  presto, prestissimo :: Fractional a => a

average :: Fractional a => a -> a -> a
average x y = (x+y)/2

largo       = uncurry average largoRange
larghetto   = uncurry average larghettoRange
adagio      = uncurry average adagioRange
andante     = uncurry average andanteRange
moderato    = uncurry average moderatoRange
allegro     = uncurry average allegroRange
presto      = uncurry average prestoRange
prestissimo = uncurry average prestissimoRange
```

Figure 5: Common names for tempo.

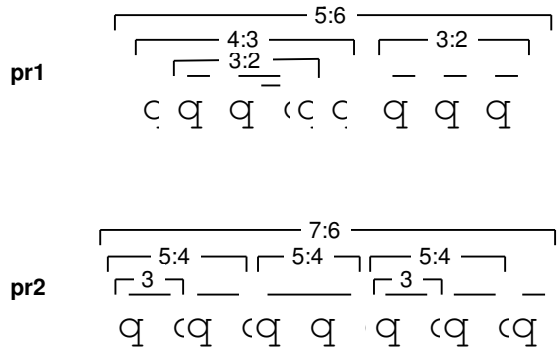


Figure 6: Nested Polyrhythms

```
pr1, pr2 :: Pitch.T -> Melody.T ()
pr1 p =
   changeTempo (5%+6)
      (changeTempo (4%+3)
         (line [mkLn 1 p qn,
                changeTempo (3%+2)
                   (line [mkLn 3 p en,
                          mkLn 2 p sn,
                          mkLn 1 p qn] ),
                 mkLn 1 p qn]) +:+
         changeTempo (3%+2) (mkLn 6 p en))

pr2 p =
   changeTempo (7%+6)
      (line [m1,
             changeTempo (5%+4) (mkLn 5 p en),
             m1,
             mkLn 2 p en])
   where m1 = changeTempo (5%+4) (changeTempo (3%+2) m2 +:+ m2)
         m2 = mkLn 3 p en

mkLn :: Int -> Pitch.T -> Music.Dur -> Melody.T ()
mkLn n p d = line (take n (List.repeat (Melody.note p d ())))
```

To play polyrhythms `pr1` and `pr2` in parallel using middle C and middle G, respectively, we would do the following (middle C is in the 5th octave):

```
pr12 :: Melody.T ()
pr12 = pr1 (5, Pitch.C) =:= pr2 (5, Pitch.G)
```

Symbolic Meter Changes We can implement a notion of "symbolic meter changes" of the form "oldnote = newnote" (quarter note = dotted eighth, for example) by defining a function:

```
(=/=) :: Music.Dur -> Music.Dur -> Music.T note -> Music.T note
old =/= new  =  changeTempo (new/old)
```

Of course, using the new function is not much longer than using `changeTempo` directly, but it may have nemonic value.

3.2 Interpretation and Performance

```
module Haskore.Performance where

import Haskore.Music(PlayerName, PhraseAttribute)

import qualified Haskore.Basic.Duration as Dur
import qualified Haskore.Basic.Pitch    as Pitch
import qualified Haskore.Music          as Music
import qualified Data.EventList.Relative.TimeBody    as TimeList
import qualified Data.EventList.Relative.TimeTime    as TimeListPad
```

```
import qualified Data.EventList.Relative.TimeMixed   as TimeListPad
import qualified Numeric.NonNegative.Class as NonNeg

import Haskore.General.Utility (maximum0, )
import Data.Tuple.HT (mapPair, )
import qualified Data.Record.HT as Record
import Data.Ord.HT (comparing, )
import Control.Monad.Trans.Reader (Reader, runReader, ask, asks, local, )
import Control.Applicative(WrappedMonad(WrapMonad), unwrapMonad, )
import Data.Traversable(sequenceA)
import Data.List (foldl')

import Prelude hiding (Monad)
```

Now that we have defined the structure of musical objects, let us turn to the issue of *performance*, which we define as a temporally ordered sequence of musical *events*:

```
type T      time dyn note = TimeList.T     time (Event time dyn note)
type Padded time dyn note = TimeListPad.T  time (Event time dyn note)
```

The `Padded` performance has a trailing time value. It can be considered as the duration after the last event after which the performance finishes. This need not to be the duration of the last event, as in the case, where the last note is a short one, that is played while an earlier long note remains playing. Another exception is a performance which ends with a rest.

```
data Event time dyn note =
       Event {eventDur       :: time,
              eventDynamics  :: dyn,
              eventTranspose :: Pitch.Relative,
              eventNote      :: note}
     deriving (Eq, Show)

-- this order is just for the old test cases which rely on it
instance (Ord time, Ord dyn, Ord note) =>
             Ord (Event time dyn note) where
   compare =
      Record.compare
         [comparing eventNote,
          comparing eventDynamics,
          comparing eventTranspose,
          comparing eventDur]
```

An event is the lowest of our music representations not yet committed to Midi, CSound, or the MusicKit. An event `Event {eventDur = d, eventNote = n}` captures the fact that the note `n` respecting all its attributes is played for a duration `d` (where now duration is measured in seconds, rather than beats).

We introduce the type variables `time` and `dyn` here which are used for time and dynamics quantities. For every-day use where only efficiency counts you will infer these type variables with `Float` or `Double`. For testing the validity of axioms (see Section 3.2.1) we need exact computation which can be achieved with `Rational`.

To generate a complete performance of, i.e. give an interpretation to, a musical object, we must know the time to begin the performance, and the proper volume, key and tempo. We must also know what *player*s to

use; that is, we need a mapping from the `PlayerNames` in an abstract musical object to the actual players to be used. (We don't yet need a mapping from abstract `Instrs` to instruments, since this is handled in the translation from a performance into, say, Midi, such as defined in Section 4.2.)

We can thus model a performer as a function `fromMusic` which maps all of this information and a musical object into a performance:

```
fromMusic ::
   (NonNeg.C time, RealFrac time, Ord dyn, Fractional dyn, Ord note) =>
   PlayerMap time dyn note -> Context time dyn note -> Music.T note -> T time dyn note

type PlayerMap time dyn note = PlayerName -> Player time dyn note
data Context time dyn note =
   Context {contextDur       :: time,
            contextPlayer    :: Player time dyn note,
            contextTranspose :: Pitch.Relative,
            contextDynamics  :: dyn}
      deriving Show

type UpdateContext time dyn note a =
   (a -> a) -> Context time dyn note -> Context time dyn note

updatePlayer    :: UpdateContext time dyn note (Player time dyn note)
updatePlayer    f c = c {contextPlayer    = f (contextPlayer c)}
updateDur       :: UpdateContext time dyn note time
updateDur       f c = c {contextDur       = f (contextDur c)}
updateTranspose :: UpdateContext time dyn note Pitch.Relative
updateTranspose f c = c {contextTranspose = f (contextTranspose c)}
updateDynamics  :: UpdateContext time dyn note dyn
updateDynamics  f c = c {contextDynamics  = f (contextDynamics c)}

fromMusic pmap c@Context {contextStart = t, contextPlayer = pl, contextDur = dt, contextTranspose = k
 case m of
    Note p d nas    -> playNote pl c p d nas
    Rest d          -> []
    m1 :+: m2       -> fromMusic pmap c m1 ++
                       fromMusic pmap (c {contextStart = t + dur m1 * dt}) m2
    m1 :=: m2       -> merge (fromMusic pmap c m1) (fromMusic pmap c m2)
    Tempo  a   m    -> fromMusic pmap (c {contextDur = dt / fromRational a}) m
    Transpose  p  m -> fromMusic pmap (c {contextTranspose = k + p}) m
    Instrument nm m -> fromMusic pmap (c {cInst = nm}) m
    Player nm  m    -> fromMusic pmap (c {contextPlayer = pmap nm}) m
    Phrase pas m    -> interpretPhrase pl pmap c pas m
```

Some things to note:

1. The function `monadFromMusic` does not simply convert a music object to a performance but it converts a music to an action (`Reader` monad). Given a context we can start the action by `runReader` and we get an event. The way `monadFromMusic` works is to build a big action from many small actions.

2. The `Context` is the running "state" of the performance, and gets updated in several different ways. For example, the interpretation of the `Tempo` constructor involves scaling the duration of a whole note appropriately and updating the `contextDur` field of the context.

```
fromMusic pmap c = fst . TimeListPad.viewTimeR . paddedFromMusic pmap c

paddedFromMusic ::
   (NonNeg.C time, RealFrac time, Ord dyn, Fractional dyn, Ord note) =>
   PlayerMap time dyn note -> Context time dyn note ->
      Music.T note -> Padded time dyn note
paddedFromMusic pmap c =
   TimeListPad.catMaybes . fst . flip runReader c . monadFromMusic pmap

type PaddedWithRests time dyn note =
        TimeListPad.T time (Maybe (Event time dyn note))

type Monad time dyn note =
   Reader
      (Context time dyn note)
      (PaddedWithRests time dyn note, time)

sequenceReader :: [Reader r a] -> Reader r [a]
sequenceReader = unwrapMonad . sequenceA . map WrapMonad

combine ::
   ([performance] -> performance, [time] -> time) ->
   [Reader r (performance, time)] ->
   Reader r (performance, time)
combine f =
   fmap (mapPair f . unzip) . sequenceReader

monadFromMusic ::
   (NonNeg.C time, RealFrac time, Ord dyn, Fractional dyn, Ord note) =>
   PlayerMap time dyn note -> Music.T note -> Monad time dyn note

monadFromMusic pmap =
   Music.foldList
      (\d at -> flip fmap ask $ \c ->
         let noteDur = Dur.toNumber d * contextDur c
             events =
                maybe
                   (TimeList.singleton 0 Nothing)
                   (TimeList.mapBody Just .
                    playNote (contextPlayer c) c d) at
         in  (TimeListPad.snocTime events noteDur, noteDur))
      (\ctrl ->
         case ctrl of
            Music.Tempo     a  -> local (updateDur (/ Dur.toNumber a))
            Music.Transpose p  -> local (updateTranspose (+ p))
            Music.Player    nm -> local (updatePlayer (const (pmap nm)))
            Music.Phrase    pa -> \m ->
               asks contextPlayer >>= \pl -> interpretPhrase pl pa m)
      (combine (TimeListPad.concat, sum))
      (combine (foldl' TimeListPad.merge (TimeListPad.pause 0), maximum0))

This implementation fails on
    mel = a 0 wn () +:+ b 0 wn ()  =:=  rest qn +:+ mel

{- this does only work if the performance in the Monad does not have a Maybe for each note

monadFromMusicOld :: (Ord time, Fractional time, Ord note) =>
   PlayerMap time dyn note -> Music.T note ->
   Reader (Context time dyn note) (Padded time dyn note, time)

monadFromMusicOld pmap =
   Music.foldList
      (\d at -> flip fmap ask $ \c ->
         let noteDur = fromRational d * contextDur c
```

It's better not to manipulate the members of `Context` directly, but to use the abstractions from `PerformanceContext`. This way we can stay independent of the concrete definition of `Context`. (I would like to define this data structure in `PerformanceContext` but the current Haskell compilers have a complicated handling of mutually dependent modules.)

3. Interpretation of notes and phrases is player dependent. Ultimately a single note is played by the `playNote` function, which takes the player as an argument. Similarly, phrase interpretation is also player dependent, reflected in the use of `interpretPhrase`. Precisely how these two functions work is described in Section 3.3.

4. The `Dur` component of the context is the duration, in seconds, of one whole note. See Section 3.1.12 for assisting functions.

5. In the treatment of `Serial`, note that the sub-sequences are appended together, with the start time of the second argument delayed by the duration of the first. The function `dur` (defined in Section 3.1.5) is used to compute this duration. Note that this results in a quadratic time complexity for `fromMusic`. A more efficient solution is to have `fromMusic` compute the duration directly, returning it as part of its result. This version of `fromMusic` is shown in Figure 7.

6. In contrast, the sub-sequences derived from the arguments to `Parallel` are merged into a time-ordered stream. This is done with `merge` from the module module `Data.EventList.Relative.TimeTime`.

3.2.1 Equivalence of Literal Performances

A *literal performance* is one in which no aesthetic interpretation is given to a musical object. The function `Pf.fromMusic` in fact yields a literal performance; aesthetic nuances must be expressed explicitly using note and phrase attributes.

There are many musical objects whose literal performances we expect to be *equivalent*. For example, the following two musical objects are certainly not equal as data structures, but we would expect their literal performances to be identical:

```
(m0 +:+ m1) +:+ (m2 +:+ m3)
  m0 +:+ m1 +:+ m2 +:+ m3
```

Thus we define a notion of equivalence:

6 Definition. Two musical objects `m0` and `m1` are *equivalent*, written `m0` $\equiv$ `m1`, if and only if:

```
(∀imap,c)  Pf.fromMusic imap c m0 = Pf.fromMusic imap c m1
```

where "=" is equality on values (which in Haskell is defined by the underlying equational logic).

One of the most useful things we can do with this notion of equivalence is establish the validity of certain *transformations* on musical objects. A transformation is *valid* if the result of the transformation is equivalent (in the sense defined above) to the original musical object; i.e. it is "meaning preserving". Some

of these connections are used in the module `Optimization` (Section 5.1) in order to simplify a musical data structure.

The most basic of these transformation we treat as *axioms* in an *algebra of music*. For example:

7 Axiom. *For any* `r0`, `r1`, *and* `m`*:*

```
changeTempo r0 (changeTempo r1 m) ≡ changeTempo (r0*r1) m
```

To prove this axiom, we use conventional equational reasoning (for clarity we omit `imap`, simplify the context to just `dt`, and omit `fromRational`):

Proof.

```
Pf.fromMusic dt (changeTempo r0 (changeTempo r1 m))
= Pf.fromMusic (dt / r0) (changeTempo r1 m)    -- unfolding Pf.fromMusic
= Pf.fromMusic ((dt / r0) / r1) m              -- unfolding Pf.fromMusic
= Pf.fromMusic (dt / (r0 * r1)) m              -- simple arithmetic
= Pf.fromMusic dt (changeTempo (r0*r1) m)      -- folding Pf.fromMusic
```

□

Here is another useful transformation and its validity proof (for clarity in the proof we omit `imap` and simplify the context to just `(t,dt)`):

8 Axiom. *For any* `r`, `m0`, *and* `m1`*:*

```
changeTempo r (m0 +:+ m1) ≡ changeTempo r m0 +:+ changeTempo r m1
```

In other words, *tempo scaling distributes over sequential composition*.

Proof.

```
Pf.fromMusic (t,dt) (changeTempo r (m0 +:+ m1))
= Pf.fromMusic (t,dt/r) (m0 +:+ m1)            -- unfolding Pf.fromMusic
= Pf.fromMusic (t,dt/r) m0 ++
     Pf.fromMusic (t',dt/r) m1                 -- unfolding Pf.fromMusic
= Pf.fromMusic (t,dt) (changeTempo r m0) ++
     Pf.fromMusic (t',dt) (changeTempo r m1)   -- folding Pf.fromMusic
               where t'  = t + dur m0 * dt/r
= Pf.fromMusic (t,dt) (changeTempo r m0) ++
     Pf.fromMusic (t'',dt) (changeTempo r m1) -- folding dur
               where t'' = t + dur (changeTempo r m0) * dt
= Pf.fromMusic (t,dt)
     (changeTempo r m0 +:+ changeTempo r m1)   -- folding Pf.fromMusic
```

□

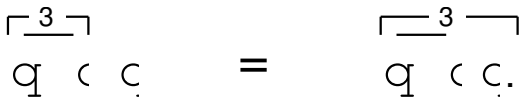


Figure 8: Equivalent Phrases

An even simpler axiom is given by:

9 Axiom. *For any* `m`*:*

```
changeTempo 1 m ≡ m
```

In other words, *unit tempo scaling is the identity*.

Proof.

```
Pf.fromMusic (t,dt) (changeTempo 1 m)
= Pf.fromMusic (t,dt/1) m                    -- unfolding Pf.fromMusic
= Pf.fromMusic (t,dt) m                      -- simple arithmetic
```

□

Note that the above proofs, being used to establish axioms, all involve the definition of `Pf.fromMusic`. In contrast, we can also establish *theorems* whose proofs involve only the axioms. For example, Axioms 1, 2, and 3 are all needed to prove the following:

10 Theorem. *For any* `r`*,* `m0`*, and* `m1`*:*

```
changeTempo r m0 +:+ m1 ≡ changeTempo r (m0 +:+ changeTempo (recip r)
                                         m1)
```

Proof.

```
changeTempo r (m0 +:+ changeTempo (recip r) m1)
= changeTempo r m0 +:+ changeTempo r (changeTempo (recip r) m1)
                                                -- by Axiom 1
= changeTempo r m0 +:+ changeTempo (r * recip r) m1 -- by Axiom 2
= changeTempo r m0 +:+ changeTempo 1 m1          -- simple arithmetic
= changeTempo r m0 +:+ m1                        -- by Axiom 3
```

□

For example, this fact justifies the equivalence of the two phrases shown in Figure 8.

Many other interesting transformations of Haskore musical objects can be stated and proved correct using equational reasoning. We leave as an exercise for the reader the proof of the following axioms (which include the above axioms as special cases).

The following axioms are additionally given in a way which allows automatic tests using the QuickCheck package. http://www.cs.chalmers.se/~rjmh/QuickCheck/ The properties are formulated as functions but they can translated one-by-one from the axioms stated in mathematical notation.

```
module Equivalence where

import           Haskore.Music hiding (repeat, reverse, dur)
import qualified Haskore.Music.GeneralMIDI  as MidiMusic
   -- should also work for general RhyMusic but is a bit more cumbersome
import qualified Haskore.Performance        as Performance
import qualified Haskore.Performance.Default as DefltPf
import qualified Haskore.Performance.Player  as Player

import qualified Haskore.Basic.Duration as Dur
import qualified Data.EventList.Relative.TimeTime as TimeListPad
import qualified Numeric.NonNegative.Wrapper as NonNeg
import Data.Tuple.HT (mapFst, )

import Control.Monad.Trans.Reader (runReader, )

import Test.QuickCheck
```

We define operators =?= and ==?== which play the role of our previously defined equivalence sign "≡". The operator =?= compares plain pieces of music, whereas the operator ==?== compares functions mapping to music. We will use the second one mainly in order to compare music transformers like `changeTempo` and `transpose`.

```
infix 4 =?=, ==?==

(=?=) :: MidiMusic.T -> MidiMusic.T -> Bool
(=?=) m0 m1 =
    let pl  = DefltPf.map :: Player.Map NonNeg.Rational Rational MidiMusic.Note
        perform m =
           mapFst TimeListPad.catMaybes $
           runReader (Performance.monadFromMusic pl m) DefltPf.context
    in  perform m0 == perform m1

(==?==) :: (a -> MidiMusic.T) -> (a -> MidiMusic.T) -> (a -> Bool)
(==?==) fm0 fm1 x  =  fm0 x =?= fm1 x
```

Here we repeat one of the simple axioms, now also with a test function ready for quick-checking.

11 Axiom. *Changing the tempo by* 1 *and transposing by* 0 *are identities. That is:*

$$\texttt{changeTempo}\ 1 \equiv \texttt{id}$$
$$\texttt{transpose}\ 0 \equiv \texttt{id}$$

```
propTempoNeutral, propTransposeNeutral :: MidiMusic.T -> Bool

propTempoNeutral = changeTempo 1 ==?== id

propTransposeNeutral = transpose 0 ==?== id
```

The first QuickCheck test function reads as: "The property of a neutral tempo change is that changing the tempo by one is equivalent to the identity function." It says everything we want to state and not more. It is available in a machine readable form ready both for static provers and for tests by execution. QuickCheck will call these functions on several randomly generated pieces of music. These songs might sound awful, so they should be exotically enough in order to check whether our axioms are not only true for common music.

12 Axiom. `changeTempo` *is* multiplicative *and* `transpose` *is* additive. *That is, for any* `r0`, `r1`, `p0`, `p1`*:*

```
changeTempo r0 .  changeTempo r1 ≡ changeTempo (r0*r1)
   transpose p0 .  transpose p1 ≡ transpose (p0+p1)
```

```
propTempoTempo ::
   Dur.Ratio -> Dur.Ratio -> MidiMusic.T -> Property
propTempoTempo r0 r1 m =
   r0 > 0 && r1 > 0   ==>
      (changeTempo r0 . changeTempo r1 ==?==
       changeTempo (r0*r1)) m

propTransposeTranspose ::
   Int -> Int -> MidiMusic.T -> Bool
propTransposeTranspose p0 p1 =
   transpose p0 . transpose p1 ==?== transpose (p0+p1)
```

The first equation needs the precondition of non-zero tempo changes. Changing the tempo to zero causes a division by zero when `Pf.fromMusic` recomputes the duration of a whole note. Because of the precondition we can no longer have `Bool` as function value but we must use `Property` which stores not only the result of the test but also if the precondition was fulfilled. Test cases where the precondition fail do not count in the maximum number of tests performed per test function.

13 Axiom. *Function composition is* commutative *with respect to both tempo scaling and transposition. That is, for any* `r0`, `r1`, `p0` *and* `p1`*:*

```
changeTempo r0 . changeTempo r1 ≡ changeTempo r1 . changeTempo r0
    transpose p0 . transpose p1 ≡ transpose p1 . transpose p0
  changeTempo r0 . transpose p0 ≡ transpose p0 . changeTempo r0
```

```
propTempoCommutativity :: Dur.Ratio -> Dur.Ratio -> MidiMusic.T -> Property
propTempoCommutativity r0 r1 m =
   r0 > 0 && r1 > 0   ==>
      (changeTempo r0 . changeTempo r1 ==?==
       changeTempo r1 . changeTempo r0) m

propTransposeCommutativity :: Int -> Int -> MidiMusic.T -> Bool
propTransposeCommutativity p0 p1 =
   transpose p0 . transpose p1 ==?== transpose p1 . transpose p0

propTempoTransposeCommutativity ::
   Dur.Ratio -> Int -> MidiMusic.T -> Property
propTempoTransposeCommutativity r p m =
   r > 0   ==>
      (changeTempo r . transpose p ==?==
       transpose p . changeTempo r) m
```

14 Axiom. *Tempo scaling and transposition are* distributive *over both sequential and parallel composition. That is, for any* `r`, `p`, `m0`, *and* `m1`*:*

```
changeTempo r (m0 +:+ m1) ≡ changeTempo r m0 +:+ changeTempo r m1
changeTempo r (m0 =:= m1) ≡ changeTempo r m0 =:= changeTempo r m1
   transpose p (m0 +:+ m1) ≡ transpose p m0 +:+ transpose p m1
   transpose p (m0 =:= m1) ≡ transpose p m0 =:= transpose p m1
```

```
propTempoSerial, propTempoParallel ::
   Dur.Ratio -> MidiMusic.T -> MidiMusic.T -> Property

propTempoSerial r m0 m1 =
   r > 0   ==>
      changeTempo r (m0 +:+ m1) =?=
      changeTempo r m0 +:+ changeTempo r m1

propTempoParallel r m0 m1 =
   r > 0   ==>
      changeTempo r (m0 =:= m1) =?=
      changeTempo r m0 =:= changeTempo r m1

propTransposeSerial, propTransposeParallel ::
   Int -> MidiMusic.T -> MidiMusic.T -> Bool
propTransposeSerial p m0 m1 =
   transpose p (m0 +:+ m1) =?= transpose p m0 +:+ transpose p m1
propTransposeParallel p m0 m1 =
   transpose p (m0 =:= m1) =?= transpose p m0 =:= transpose p m1
```

15 Axiom. *Sequential and parallel composition are* associative*. That is, for any* `m0`, `m1`, *and* `m2`*:*

```
m0 +:+ (m1 +:+ m2) ≡ (m0 +:+ m1) +:+ m2
m0 =:= (m1 =:= m2) ≡ (m0 =:= m1) =:= m2
```

```
propSerialAssociativity, propParallelAssociativity ::
   MidiMusic.T -> MidiMusic.T -> MidiMusic.T -> Bool
propSerialAssociativity m0 m1 m2 =
   m0 +:+ (m1 +:+ m2) =?= (m0 +:+ m1) +:+ m2
propParallelAssociativity m0 m1 m2 =
   m0 =:= (m1 =:= m2) =?= (m0 =:= m1) =:= m2
```

16 Axiom. *Parallel composition is* commutative*. That is, for any* `m0` *and* `m1`*:*

```
m0 =:= m1 ≡ m1 =:= m0
```

```
propParallelCommutativity ::
   MidiMusic.T -> MidiMusic.T -> Bool
propParallelCommutativity m0 m1 =
   m0 =:= m1 =?= m1 =:= m0
```

17 Axiom. `Rest 0` *is a* unit *for* `changeTempo` *and* `transpose`*, and a* zero *for sequential and parallel composition. That is, for any* `r`*,* `p`*, and* `m`*:*

```
changeTempo r (Rest 0) ≡ Rest 0
 transpose p (Rest 0) ≡ Rest 0
m +:+ Rest 0 ≡ m ≡ Rest 0 +:+ m
m =:= Rest 0 ≡ m ≡ Rest 0 =:= m
```

```
propTempoRest0 :: Dur.Ratio -> Property
propTempoRest0 r =
   r > 0   =⇒
      changeTempo r (rest 0) =?= rest 0
propTransposeRest0 :: Int -> Bool
propTransposeRest0 p = transpose p (rest 0) =?= rest 0

propSerialNeutral0, propSerialParallel0,
  propSerialNeutral1, propSerialParallel1 ::
    MidiMusic.T -> Bool
propSerialNeutral0 m  =  m +:+ rest 0 =?= m
propSerialNeutral1 m  =  rest 0 +:+ m =?= m
propSerialParallel0 m  =  m =:= rest 0 =?= m
propSerialParallel1 m  =  rest 0 =:= m =?= m
```

18 Exercise. *Establish the validity of each of the above axioms.*

3.3 Players

In the last section we saw how a performance involved the notion of a *player*. The reason for this is the same as for real players and their instruments: many of the note and phrase attributes (see Section 3.1.8) are player and instrument dependent. For example, how should "legato" be interpreted in a performance? Or "diminuendo"? Different players interpret things in different ways, of course, but even more fundamental is the fact that a pianist, for example, realizes legato in a way fundamentally different from the way a violinist does, because of differences in their instruments. Similarly, diminuendo on a piano and a harpsichord are different concepts.

With a slight stretch of the imagination, we can even consider a "notator" of a score as a kind of player: exactly how the music is rendered on the written page may be a personal, stylized process. For example, how many, and which staves should be used to notate a particular instrument?

In any case, to handle these issues, Haskore has a notion of a *player* which "knows" about differences with respect to performance and notation. A Haskore player is a 4-tuple consisting of a name and three functions: one for interpreting notes, one for phrases, and one for producing a properly notated score.

```
data Player time dyn note =
        PlayerCons { name            :: PlayerName,
                     playNote        :: NoteFun time dyn note,
                     interpretPhrase :: PhraseFun time dyn note,
                     notatePlayer    :: NotateFun }

instance (Show time, Show dyn) => Show (Player time dyn note) where
```

```
   show p = "Player.cons " ++ name p

type NoteFun time dyn note =
     Context time dyn note -> Music.Dur -> note -> T time dyn note
type PhraseFun time dyn note =
     PhraseAttribute -> Monad time dyn note -> Monad time dyn note
type NotateFun = ()
```

The last line above is because notation is currently not implemented. Note that both `NotateFun` and `PhraseFun` functions return a `Performance.T`.

```
module Haskore.Performance.Player where

import Haskore.Music (PhraseAttribute, )
import qualified Haskore.Music as Music
-- import qualified Haskore.Performance.Context as Context
   -- this import would cause a cycle
import qualified Haskore.Performance as Pf
-- import qualified Data.EventList.Relative.TimeBody    as TimeList
import qualified Data.EventList.Relative.TimeTime  as TimeListPad
import qualified Data.EventList.Relative.TimeMixed as TimeListPad
import qualified Haskore.Basic.Duration as Dur
import qualified Numeric.NonNegative.Class as NonNeg
import Haskore.Performance (eventDur, eventDynamics, )
import Data.Tuple.HT (mapFst, )

import Control.Monad.Trans.Reader(Reader, asks, )
import Control.Monad (liftM, )

type T    time dyn note = Pf.Player time dyn note
-- constructors can't be renamed, we might use a function instead
-- cons = Pf.PlayerCons

type Name               = Music.PlayerName
type Map  time dyn note = Pf.PlayerMap time dyn note


type PhraseInterpreter time dyn note =
   PhraseAttribute -> (Pf.T time dyn note, time) -> (Pf.T time dyn note, time)

type EventModifier time dyn note =
   Pf.Event time dyn note -> Pf.Event time dyn note

changeVelocity :: Num dyn => (dyn -> dyn) ->
   EventModifier time dyn note
changeVelocity f =
   (\e -> e {eventDynamics = f (eventDynamics e)})

changeDur :: Num time => (time -> time) ->
   EventModifier time dyn note
changeDur f =
   (\e -> e {eventDur = f (eventDur e)})
```

Figure 10 defines a relatively sophisticated player called `fancyPlayer` that knows all that `Player.deflt` knows, and much more.

All three articulations `Staccato`, `Legato`, `Slurred` are interpreted as changing the duration of the notes proportionally. That's why they have the suffix `Rel` for *relative*.

- The function `legatoRel` takes a ratio of each note's duration. In order to obtain a real Legato effect the value must be larger than 1.
- The function `slurredRel` is similar to `legatoRel` but it doesn't extend the duration of the *last* note(s).
- The function `staccatoRel` divides the note durations by constant factor. In order to obtain a real Staccato effect the value must be larger than 1.

```
staccatoRel, legatoRel, slurredRel :: (NonNeg.C time, Fractional time) =>
   Dur.T -> Pf.Monad time dyn note -> Pf.Monad time dyn note
staccatoRel x = mapEvents     (changeDur (/ Dur.toNumber x))
legatoRel   x = mapEvents     (changeDur (* Dur.toNumber x))
slurredRel  x = mapInitEvents (changeDur (* Dur.toNumber x))

mapInitEvents :: (NonNeg.C time, Num time) =>
   EventModifier time dyn note ->
       Pf.Monad time dyn note -> Pf.Monad time dyn note
mapInitEvents f =
   let -- modify durations of all notes except those with the latest start time
       aux =
          TimeListPad.flatten .
          TimeListPad.mapTimeInit
             (TimeListPad.mapBodyInit
                 (TimeListPad.mapBody (map (fmap f)))) .
          TimeListPad.collectCoincident
   in  liftM (mapFst aux)

mapEvents :: EventModifier time dyn note ->
                Pf.Monad time dyn note -> Pf.Monad time dyn note
mapEvents f = liftM (mapFst (TimeListPad.mapBody (fmap f)))
```

In contrast to the relative interpretations above, we feel that somehow absolute changes are more useful. That's why we make these functions the default for the fancy player. These function expect regular note durations, that is ratios of a whole note.

- The functions `legatoAbs` and `slurredAbs` prolong notes by a fix amount. That is the overlap (if no rests are between) is constant.
- `staccatoAbs` replaces the note durations by a fix amount.

```
staccatoAbs, legatoAbs, slurredAbs :: (NonNeg.C time, Fractional time) =>
   Dur.T -> Pf.Monad time dyn note -> Pf.Monad time dyn note
staccatoAbs dur pf =
   getDurModifier const dur >>= flip mapEvents pf
legatoAbs dur pf =
   getDurModifier (+)   dur >>= flip mapEvents pf
slurredAbs dur pf =
```

```
   getDurModifier (+)   dur >>= flip mapInitEvents pf

getDurModifier :: (Fractional time) =>
   (time -> time -> time) -> Dur.T ->
      Reader (Pf.Context time dyn note) (EventModifier time dyn note)
getDurModifier f dur =
   do tempo <- asks Pf.contextDur
      return (changeDur (f (Dur.toNumber dur * tempo)))
```

The behavior of `(Ritardando x)` can be explained as follows. We'd like to "stretch" the time of each event by a factor from 0 to x, linearly interpolated based on how far along the musical phrase the event occurs. I.e., given a start time t_0 for the first event in the phrase, total phrase duration D, and event time t, the new event time t' is given by:

$$t' = \left(1 + \frac{t - t_0}{D} \cdot x\right) \cdot (t - t_0) + t_0$$

Further, if d is the duration of the event, then the end of the event $t + d$ gets stretched to a new time t'_d given by:

$$t'_d = \left(1 + \frac{t + d - t_0}{D} \cdot x\right) \cdot (t + d - t_0) + t_0$$

The difference $t'_d - t'$ gives us the new, stretched duration d', which after simplification is:

$$d' = \left(1 + \frac{2 \cdot (t - t_0) + d}{D} \cdot x\right) \cdot d$$

`Accelerando` behaves in exactly the same way, except that it shortens event times rather than lengthening them. And, a similar but simpler strategy explains the behaviors of `Crescendo` and `Diminuendo`.

```
accent :: (Fractional dyn) =>
   Rational -> Pf.Monad time dyn note -> Pf.Monad time dyn note
accent x = mapEvents (changeVelocity (fromRational x +))
```

3.4 Conversion functions with default settings

3.4.1 Examples of Player Construction

A "default player" called `Default.player` (not to be confused with "deaf player"!) is defined for use when none other is specified in the score; it also functions as a base from which other players can be derived. `Default.player` responds only to the `Velocity` note attribute and to the `Accent`, `Staccato`, and `Legato` phrase attributes. It is defined in Figure 9. Before reading this code, recall how players are invoked by the `Performance.fromMusic` function defined in the last section; in particular, note the calls to `playNote` and `interpretPhase` defined above. Then note:

1. `defltPlayNote` is the only function (even in the definition of `Performance.fromMusic`) that actually generates an event. It also modifies that event based on an interpretation of each note attribute by the function `defltNasHandler`.

2. `defltNasHandler` only recognizes the `Velocity` attribute, which it uses to set the event velocity accordingly.

3. `defltInterpPhrase` calls (mutually recursively) `Performance.fromMusic` to interpret a phrase, and then modifies the result based on an interpretation of each phrase attribute by the function `defltInterpPhrase`.

4. `defltInterpPhrase` only recognizes the `Accent`, `Staccato`, and `Legato` phrase attributes. For each of these it uses the numeric argument as a "scaling" factor of the volume (for `Accent`) and duration (for `Staccato` and `Legato`). Thus `(Phrase (Legato 1.1) m)` effectively increases the duration of each note in `m` by 10% (without changing the tempo).

It should be clear that much of the code in Figure **??** can be re-used in defining a new player. For example, to define a player `weird` that interprets note attributes just like `Default.player` but behaves differently with respect to phrase attributes, we could write:

```
weird  = Performance.PlayerCons {
            pname           = "Weirdo",
            playNote        = defltPlayNote defltNasHandler,
            interpretPhrase = liftM . myPhraseInterpreter
            notatePlayer    = defltNotatePlayer ()
         }
```

and then supply a suitable definition of `myPhraseInterpreter`. That definition could also re-use code, in the following sense: suppose we wish to add an interpretation for `Crescendo`, but otherwise have `myPhraseInterpreter` behave just like `defltInterpPhrase`.

```
myPhraseInterpreter (Dyn (Crescendo x)) pf = ...
myPhraseInterpreter  pa                 pf = defltInterpPhrase pa pf
```

19 Exercise. *Fill in the* `...` *in the definition of* `myPhraseInterpreter` *according to the following strategy: Assume* $0 < \mathtt{x} < 1$. *Gradually scale the volume of each event by a factor of* 1.0 *through* $1.0 + \mathtt{x}$, *using linear interpolation.*

20 Exercise. *Choose some of the other phrase attributes and provide interpretations of them, such as* `Diminuendo`, `Slurred`, `Trill`, *etc. (The* `trill` *functions from Section 3.1.5 may be useful here.)*

```
module Haskore.Performance.Default where

import qualified Haskore.Music       as Music
import qualified Haskore.Performance as Performance
import qualified Haskore.Performance.Context as Context
import qualified Haskore.Performance.Player  as Player

import qualified Data.EventList.Relative.TimeBody    as TimeList
```

```
import qualified Haskore.Basic.Tempo    as Tempo
import qualified Haskore.Basic.Duration as Dur

import qualified Numeric.NonNegative.Class   as NonNeg
import qualified Numeric.NonNegative.Wrapper as NonNegW

import Prelude hiding (map)
```

```
fromMusic ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   Music.T note -> Performance.T time dyn note
fromMusic =
   Performance.fromMusic map context

fromMusicModifyContext ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   (Context.T time dyn note -> Context.T time dyn note) ->
   Music.T note ->
   Performance.T time dyn note
fromMusicModifyContext update =
   Performance.fromMusic
      map
      (update context)

floatFromMusic :: (Ord note) =>
   Music.T note -> Performance.T NonNegW.Float Float note
floatFromMusic = fromMusic

paddedFromMusic  ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   Music.T note -> Performance.Padded time dyn note
paddedFromMusic =
   Performance.paddedFromMusic map context

paddedFromMusicModifyContext ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   (Context.T time dyn note -> Context.T time dyn note) ->
   Music.T note ->
   Performance.T time dyn note
paddedFromMusicModifyContext update =
   Performance.fromMusic
      map
      (update context)
```

3.5 Conversion functions with default settings

```
module Haskore.Performance.Fancy where

import qualified Haskore.Music       as Music
import qualified Haskore.Performance as Performance
import qualified Haskore.Performance.Context as Context
import qualified Haskore.Performance.Player  as Player
import qualified Haskore.Performance.Default as DefltPf
```

```
-- default is a reserved keyword
player ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   Player.T time dyn note
player = map "Default"

-- a default PMap that makes everything into a Default.player
map ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   Player.Name -> Player.T time dyn note
map pname =
   Performance.PlayerCons {
      Performance.name            = pname,
      Performance.playNote        = playNote,
      Performance.interpretPhrase = interpretPhrase,
      Performance.notatePlayer    = notatePlayer ()
   }

playNote :: (Fractional time, Real time) =>
   Performance.NoteFun time dyn note
playNote
   (Performance.Context curDur _ curKey curVelocity) d note =
       TimeList.singleton 0
            (Performance.Event {
               Performance.eventDur       = Dur.toNumber d * curDur,
               Performance.eventTranspose = curKey,
               Performance.eventDynamics  = curVelocity,
               Performance.eventNote      = note } )

interpretPhrase ::
   (NonNeg.C time, Fractional time, Fractional dyn) =>
   Performance.PhraseFun time dyn note
interpretPhrase (Music.Dyn (Music.Accent   x)) = Player.accent x
interpretPhrase (Music.Art (Music.Staccato x)) = Player.staccatoAbs x
interpretPhrase (Music.Art (Music.Legato   x)) = Player.legatoAbs x
interpretPhrase _                              = id

notatePlayer :: () -> Performance.NotateFun
notatePlayer _ = ()

context ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   Context.T time dyn note
context =
   Performance.Context {
      Performance.contextPlayer     = player,
      Performance.contextDur        = Tempo.metro 60 Dur.qn,
      Performance.contextTranspose  = 0,
      Performance.contextDynamics   = 1
   }
```

Figure 9: Definition of default Player `Default.player`.

```
import Haskore.Performance (eventDur, )

-- import qualified Data.EventList.Relative.TimeBody  as TimeList
-- import qualified Data.EventList.Relative.TimeTime  as TimeListPad
import qualified Data.EventList.Relative.MixedTime as TimeListPad
import qualified Data.EventList.Relative.BodyTime  as BodyTimeList

import Control.Monad.Trans.State  (state, evalState, )
import Control.Monad.Trans.Reader (local, )

import qualified Numeric.NonNegative.Class   as NonNeg
import qualified Numeric.NonNegative.Wrapper as NonNegW

import Prelude hiding (map)
```

```
fromMusic ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   Music.T note -> Performance.T time dyn note
fromMusic =
   Performance.fromMusic map context

fromMusicModifyContext ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   (Context.T time dyn note -> Context.T time dyn note) ->
   Music.T note ->
   Performance.T time dyn note
fromMusicModifyContext update =
   Performance.fromMusic
      map
      (update context)

floatFromMusic :: (Ord note) =>
   Music.T note -> Performance.T NonNegW.Float Float note
floatFromMusic = fromMusic

paddedFromMusic  ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   Music.T note -> Performance.Padded time dyn note
paddedFromMusic =
   Performance.paddedFromMusic map context

doublePaddedFromMusic  ::
   (Ord note) =>
   Music.T note -> Performance.Padded NonNegW.Double Double note
doublePaddedFromMusic =
   Performance.paddedFromMusic map context

paddedFromMusicModifyContext ::
   (Ord note, NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn) =>
   (Context.T time dyn note -> Context.T time dyn note) ->
   Music.T note ->
   Performance.T time dyn note
paddedFromMusicModifyContext update =
   Performance.fromMusic
```

```
player :: (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   Player.T time dyn note
player = map "Fancy"

-- a PMap that makes everything into a fancyPlayer
map ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   String -> Player.T time dyn note
map pname =
   Performance.PlayerCons {
      Performance.name            = pname,
      Performance.playNote        = DefltPf.playNote,
      Performance.interpretPhrase = fancyInterpretPhrase,
      Performance.notatePlayer    = DefltPf.notatePlayer ()
   }

processPerformance :: (Num time) =>
   (time ->
      (time -> time -> time,
       time -> Performance.Event time dyn note -> Performance.Event time dyn note,
       time)) ->
   (Performance.PaddedWithRests time dyn note, time) ->
   (Performance.PaddedWithRests time dyn note, time)
processPerformance f (pf, dur) =
   let (fTime, fEvent, newDur) = f dur
       procPf =
          flip evalState 0 .
          BodyTimeList.mapM
             (\dt -> state $ \t -> (fTime  t dt, t+dt))
             (\ev -> state $ \t -> (fmap (fEvent t) ev, t))
   in  (TimeListPad.mapTimeTail procPf pf, newDur)

fancyInterpretDynamic ::
   (Fractional time, Real time, Fractional dyn) =>
   Music.Dynamic -> Performance.Monad time dyn note -> Performance.Monad time dyn note
fancyInterpretDynamic dyn =
 let loud x = local (Performance.updateDynamics (fromRational x *))
     inflate add x dur =
        let r = fromRational x / realToFrac dur
        in  (const id,
             \t -> Player.changeVelocity (add (realToFrac t * r)),
             dur)
 in  case dyn of
        Music.Accent x       -> Player.accent x
        Music.Loudness x     -> loud x
        Music.Crescendo x    -> fmap (processPerformance (inflate (+)      x))
        Music.Diminuendo x   -> fmap (processPerformance (inflate subtract x))
--        Music.Crescendo x    -> fmap (processPerformance (inflate x))
--        Music.Diminuendo x   -> fmap (processPerformance (inflate (-x)))

fancyInterpretTempo :: (Fractional time, Real time) =>
   Music.Tempo -> Performance.Monad time dyn note -> Performance.Monad time dyn note
fancyInterpretTempo tmp =
 let stretch add x dur =
        let x' = fromRational x
            r = x' / dur
            fac t dt = add 1 (r * (2*t + dt))
        in  (\t dt -> dt * fac t dt,
             \t (e@Performance.Event {eventDur = d}) ->
                e{eventDur = d * fac t d },
             dur * add 1 x')
 in  case tmp of
        Music.Ritardando  x  -> fmap (processPerformance (stretch (+) x))
        Music.Accelerando x  -> fmap (processPerformance (stretch (-) x))
```

```
      map
      (update context)
```

4 Interfaces to other musical software

4.1 Connect Performance to a Back-End

```
module Haskore.Performance.BackEnd where

import qualified Haskore.Performance    as Pf
import qualified Haskore.Music          as Music
import qualified Haskore.Basic.Pitch    as Pitch
import qualified Data.EventList.Relative.TimeBody    as TimeList
import qualified Data.EventList.Relative.TimeTime as TimeListPad

import Haskore.Music ((=:=), (+:+))
```

The performance data structure is still bound to music specific data. We still have to convert that into back-end specific data, such as MIDI events, CSound statements, SuperCollider messages or other. The new data type `Performance.BackEnd.T` is similar to `Performance.T`, but does not contain transposition or dynamics information any longer. Also music-specific data is converted to back-end specific data.

Later we have to provide converters from each type of music to each back-end. This requires combinatorial amount of implementation work but it is the most flexible way to do so. We expect only a few general types of music which fit to many back-ends, and many music types specialised to features of a particular back-end. It would be certainly less work to have an universal intermediate, but this restricts the flexibility.

```
type T      time note = TimeList.T    time (Event time note)
type Padded time note = TimeListPad.T time (Event time note)

data Event time note =
       Event {eventDur  :: time,
              eventNote :: note}
     deriving (Eq, Ord, Show)
```

Now we provide a function which simplifies conversion from a `Performance.Event` to a `Performance.BackEnd.Event` in case that this conversion does not depend on the event time and duration.

```
instance Functor (Event time) where
   fmap f e = e{eventNote = f (eventNote e)}

mapTime :: (time0 -> time1) -> T time0 note -> T time1 note
mapTime f =
   TimeList.mapBody
      (\ev -> ev{eventDur = f (eventDur ev)}) .
   TimeList.mapTime f

mapTimePadded ::
   (time0 -> time1) -> Padded time0 note -> Padded time1 note
```

```
mapTimePadded f =
   TimeListPad.mapBody
      (\ev -> ev{eventDur = f (eventDur ev)}) .
   TimeListPad.mapTime f

eventFromPerformanceEvent ::
   (dyn -> Pitch.Relative -> note -> backEndNote) ->
      Pf.Event time dyn note -> Event time backEndNote
eventFromPerformanceEvent f =
   \ (Pf.Event dur vel trans note)
            -> Event dur (f vel trans note)

fromPerformance ::
   (dyn -> Pitch.Relative -> note -> backEndNote) ->
      Pf.T time dyn note -> T time backEndNote
fromPerformance = TimeList.mapBody . eventFromPerformanceEvent

fromPaddedPerformance ::
   (dyn -> Pitch.Relative -> note -> backEndNote) ->
      Pf.Padded time dyn note -> Padded time backEndNote
fromPaddedPerformance = TimeListPad.mapBody . eventFromPerformanceEvent
```

For symmetry we also provide a function which converts a performance back to a music. This operation is not uniquely defined, and a satisfying implementation is a music theoretical challenge. A sophisticated algorithm would have to make assumptions about the structure of "common" music. So you will be able to construct examples of music that fool such an algorithm.

The opposite extreme is a version which simply maps the stream of notes to a big parallel composition where each parallel channel consists of one note. (The normal form as described in Hudak's Temporal Media paper.)

The following implementation tries to avoid obviously unnecessary parallelism by watching for non-overlapping notes. Nevertheless the conversion of general polyphonic music yields a music that is not very nicely structured. So, don't rely on the structure of the restored music, only assume that this functions reverts the performance generation.

```
toMusic :: T Music.Dur note -> Music.T note
toMusic =
   TimeList.switchL
      (Music.rest 0)
      (\ (t0, Event d mn) es0 ->
         let n = if d>=0
                   then Music.atom d (Just mn)
                   else error "Performance.toMusic: note of negative duration"
             rmd =
                TimeList.switchL n
                   (\(t1, re1) es1 ->
                      if t1 >= d
                        then n +:+ toMusic (TimeList.cons (t1-d) re1 es1)
                        else n =:= toMusic es0)
                   es0
         in  case compare t0 0 of
               EQ -> rmd
               GT -> Music.rest t0 +:+ rmd
```

```
            LT -> error "Performance.toMusic: events in wrong order")
```

4.2 Midi

Midi ("musical instrument digital interface") is a standard protocol adopted by most, if not all, manufacturers of electronic instruments. At its core is a protocol for communicating *musical events* (note on, note off, key press, etc.) as well as so-called *meta events* (select synthesizer patch, change volume, etc.). Beyond the logical protocol, the Midi standard also specifies electrical signal characteristics and cabling details. In addition, it specifies what is known as a *standard Midi file* which any Midi-compatible software package should be able to recognize.

Over the years musicians and manufacturers decided that they also wanted a standard way to refer to *common* or *general* instruments such as "acoustic grand piano", "electric piano", "violin", and "acoustic bass", as well as more exotic ones such as "chorus aahs", "voice oohs", "bird tweet", and "helicopter". A simple standard known as *General Midi* was developed to fill this role. It is nothing more than an agreed-upon list of instrument names along with a *program patch number* for each, a parameter in the Midi standard that is used to select a Midi instrument's sound.

Most "sound-blaster"-like sound cards on conventional PC's know about Midi, as well as General Midi. However, the sound generated by such modules, and the sound produced from the typically-scrawny speakers on most PC's, is often quite poor. It is best to use an outboard keyboard or tone generator, which are attached to a computer via a Midi interface and cables. It is possible to connect several Midi instruments to the same computer, with each assigned a different *channel*. Modern keyboards and tone generators are quite amazing little beasts. Not only is the sound quite good (when played on a good stereo system), but they are also usually *multi-timbral*, which means they are able to generate many different sounds simultaneously, as well as *polyphonic*, meaning that simultaneous instantiations of the same sound are possible.

If you decide to use the General Midi features of your sound-card, you need to know about another set of conventions known as "General Midi". The most important aspect of General Midi is that Channel 10 (9 in Haskore's 0-based numbering) is dedicated to *percussion*.

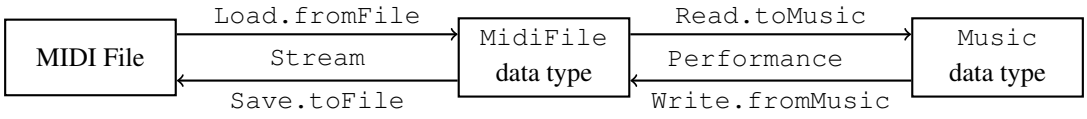
Haskore provides a way to specify a Midi channel number and General Midi instrument selection for each `Instr` in a Haskore composition. It also provides a means to generate a Standard Midi File, which can then be played using any conventional Midi software. Finally, it provides a way for existing Midi files to be read and converted into a `MidiMusic.T` object in Haskore. In this section the top-level code needed by the user to invoke this functionality will be described, along with the gory details.

```
module Haskore.Interface.MIDI.Write
          (fromRhythmicPerformance, fromRhythmicPerformanceMixed,
           fromGMPerformance,       fromGMPerformanceMixed,
           fromGMPerformanceAuto,   fromGMPerformanceMixedAuto,
           fromRhythmicMusic,       fromRhythmicMusicMixed,
           fromGMMusic,             fromGMMusicAuto,
           fromGMMusicMixed,        fromGMMusicMixedAuto,
           volumeHaskoreToMIDI, volumeMIDIToHaskore)
       where

import qualified Sound.MIDI.File             as MidiFile
import qualified Sound.MIDI.File.Event       as MidiFileEvent
```

```
import qualified Sound.MIDI.File.Event.Meta  as MetaEvent
import qualified Sound.MIDI.Message.Channel       as ChannelMsg
import qualified Sound.MIDI.Message.Channel.Voice as Voice
import qualified Haskore.Interface.MIDI.InstrumentMap as InstrMap
import qualified Haskore.Interface.MIDI.Note          as MidiNote

import qualified Haskore.Music.GeneralMIDI as MidiMusic
import qualified Haskore.Music.Rhythmic    as RhyMusic
import qualified Haskore.Performance as Performance
import qualified Haskore.Performance.Context as Context
import qualified Haskore.Performance.BackEnd as PerformanceBE
import qualified Haskore.Performance.Fancy   as FancyPf
import qualified Data.EventList.Relative.TimeBody  as TimeList
import qualified Data.EventList.Relative.MixedBody as TimeList
import qualified Data.EventList.Relative.BodyBody  as BodyBodyList
import qualified Haskore.Basic.Pitch           as Pitch

import qualified Numeric.NonNegative.Class as NonNeg

import qualified Haskore.General.Map as Map
import Data.Ord.HT (limit, )
import Data.Maybe (mapMaybe, )
import Control.Monad.Trans.State (state, evalState, )
import Control.Monad (liftM, )
```

Instead of converting a Haskore `Performance.T` directly into a Midi file, Haskore first converts it into a datatype that *represents* a Midi file, which is then written to a file in a separate pass. This separation of concerns makes the structure of the Midi file clearer, makes debugging easier, and provides a natural path for extending Haskore's functionality with direct Midi capability.

Here is the basic structure of the modules and functions:



Given instrument and drum maps (Section 4.2.2), a performance is converted to a datatype representing a Standard Midi File of type 0 (`Mixed` - one track containing data of all channels) or type 1 (`Parallel` - tracks played simultaneously) using the `from*PerformanceMixed` and `from*Performance` functions, respectively. The "`Mixed`" mode is the only one which can be used in principle for infinite music, since the number of tracks is stored explicitly in the MIDI file which depends on the number of instruments actually used in the song. Nevertheless such a stream can not be written to a pipe (not to speak of a physical disk), since the binary MIDI file format stores lengths of tracks.

The functions with names of the form `fromRhythmicPerformance*` convert from generic rhythmic music performances using appropriate tables. In contrast to that, for General MIDI music the instrument and drum maps are fixed. There are the two variants `fromGMPerformance*`, which allows explicit assignment of instruments to channels, and `fromGMPerformance*Auto`, which assigns the channels automatically one by one.

```
type Perf time dyn drum instr =
```

```
        Performance.T time dyn (RhyMusic.Note drum instr)

type NotePerfToBE dyn drum instr =
        dyn -> Pitch.Relative ->
           RhyMusic.Note drum instr -> MidiNote.T

getInstrument ::
   Performance.Event time dyn (RhyMusic.Note drum instr) -> Maybe instr
getInstrument =
   RhyMusic.maybeInstrument . RhyMusic.body . Performance.eventNote

fromRhythmicPerformance ::
   (NonNeg.C time, RealFrac time, RealFrac dyn,
    Eq drum, Eq instr) =>
   InstrMap.ChannelProgramPitchTable drum ->
   InstrMap.ChannelProgramTable instr ->
   Perf time dyn drum instr -> MidiFile.T
fromRhythmicPerformance dMap iMap =
   fromRhythmicPerformanceBase
      (const (MidiNote.fromRhyNote
         (InstrMap.lookup dMap) (InstrMap.lookup iMap)))

fromGMPerformance ::
   (NonNeg.C time, RealFrac time, RealFrac dyn) =>
   (MidiMusic.Instrument -> ChannelMsg.Channel) ->
      Performance.T time dyn MidiMusic.Note -> MidiFile.T
fromGMPerformance cMap =
   fromRhythmicPerformanceBase
      (const (MidiNote.fromGMNote cMap))

fromGMPerformanceAuto ::
   (NonNeg.C time, RealFrac time, RealFrac dyn) =>
   Performance.T time dyn MidiMusic.Note -> MidiFile.T
fromGMPerformanceAuto =
   fromRhythmicPerformanceBase
      (\instrs -> MidiNote.fromGMNote
            (InstrMap.fromInstruments instrs))

fromRhythmicPerformanceBase ::
   (NonNeg.C time, RealFrac time, Eq instr) =>
   ([instr] -> NotePerfToBE dyn drum instr) ->
      Perf time dyn drum instr -> MidiFile.T
fromRhythmicPerformanceBase makeNoteMap pf =
   let splitList = TimeList.slice getInstrument pf
       noteMap   = makeNoteMap (mapMaybe fst splitList)
       {- noteMap will always lookup instruments in a map
          although the instrument will be the same for each track. -}
       pfBEs     = map (PerformanceBE.fromPerformance noteMap)
                       (map snd splitList)
   in  MidiFile.Cons MidiFile.Parallel (MidiFile.Ticks division)
          (map trackFromPfBE pfBEs)


fromRhythmicPerformanceMixed ::
   (NonNeg.C time, RealFrac time, RealFrac dyn, Eq drum, Eq instr) =>
```

```
   InstrMap.ChannelProgramPitchTable drum ->
   InstrMap.ChannelProgramTable instr ->
   Perf time dyn drum instr -> MidiFile.T
fromRhythmicPerformanceMixed dMap iMap =
   fromRhythmicPerformanceMixedBase
      (MidiNote.fromRhyNote
         (InstrMap.lookup dMap) (InstrMap.lookup iMap))

fromGMPerformanceMixed ::
   (NonNeg.C time, RealFrac time, RealFrac dyn) =>
   (MidiMusic.Instrument -> ChannelMsg.Channel) ->
      Performance.T time dyn MidiMusic.Note -> MidiFile.T
fromGMPerformanceMixed cMap =
   fromRhythmicPerformanceMixedBase (MidiNote.fromGMNote cMap)

fromGMPerformanceMixedAuto ::
   (NonNeg.C time, RealFrac time, RealFrac dyn) =>
   Performance.T time dyn MidiMusic.Note -> MidiFile.T
fromGMPerformanceMixedAuto pf =
   let instrs = mapMaybe fst (TimeList.slice getInstrument pf)
       cMap = InstrMap.fromInstruments instrs
   in  fromRhythmicPerformanceMixedBase
          (MidiNote.fromGMNote cMap) pf

fromRhythmicPerformanceMixedBase ::
   (NonNeg.C time, RealFrac time, RealFrac dyn, Eq instr) =>
   NotePerfToBE dyn drum instr ->
   Perf time dyn drum instr -> MidiFile.T
fromRhythmicPerformanceMixedBase noteMap pf =
   MidiFile.Cons MidiFile.Mixed (MidiFile.Ticks division)
          [trackFromPfBE (PerformanceBE.fromPerformance noteMap pf)]
```

The more comfortable function `fromRhythmicMusic` turns a `MidiMusic.T` immediately into a `MidiFile.T`. Thus it needs also a `Context` and drum and instrument table. The signature of `fromGMMusic` is chosen so that it can be used as an inverse to `ReadMidi.toGMMusic`. The function `fromGMMusicAuto` is similar but doesn't need a `InstrMap.ChannelTable` because it creates one from the set of instruments actually used in the `MidiMusic.T`.

```
fromRhythmicMusic, fromRhythmicMusicMixed ::
   (NonNeg.C time, RealFrac time, RealFrac dyn,
    Ord drum, Ord instr) =>
      (InstrMap.ChannelProgramPitchTable drum,
       InstrMap.ChannelProgramTable instr,
       Context.T time dyn (RhyMusic.Note drum instr),
       RhyMusic.T drum instr) -> MidiFile.T

fromGMMusic, fromGMMusicMixed ::
   (NonNeg.C time, RealFrac time, RealFrac dyn) =>
      (InstrMap.ChannelTable MidiMusic.Instr,
       Context.T time dyn MidiMusic.Note, MidiMusic.T) -> MidiFile.T

fromGMMusicAuto, fromGMMusicMixedAuto ::
   (NonNeg.C time, RealFrac time, RealFrac dyn) =>
      (Context.T time dyn MidiMusic.Note, MidiMusic.T) -> MidiFile.T
```

```
fromRhythmicMusic      (dm,im,c,m) =
   fromRhythmicMusicBase (fromRhythmicPerformance dm im)      c m
fromRhythmicMusicMixed (dm,im,c,m) =
   fromRhythmicMusicBase (fromRhythmicPerformanceMixed dm im) c m
fromGMMusic       (cm,c,m) =
   fromRhythmicMusicBase  (fromGMPerformance      (InstrMap.lookup cm)) c m
fromGMMusicMixed  (cm,c,m) =
   fromRhythmicMusicBase  (fromGMPerformanceMixed (InstrMap.lookup cm)) c m
fromGMMusicAuto       (c,m) =
   fromRhythmicMusicBase  fromGMPerformanceAuto        c m
fromGMMusicMixedAuto  (c,m) =
   fromRhythmicMusicBase  fromGMPerformanceMixedAuto   c m

fromRhythmicMusicBase ::
   (NonNeg.C time, RealFrac time, Fractional dyn, Ord dyn,
    Ord drum, Ord instr) =>
   (Perf time dyn drum instr -> MidiFile.T) ->
      Context.T time dyn (RhyMusic.Note drum instr) ->
      RhyMusic.T drum instr -> MidiFile.T
fromRhythmicMusicBase p c m = p (Performance.fromMusic FancyPf.map c m)
```

General Midi specific definitions are imported from module `GeneralMidi` (see Section **??**). The Midi file datatype itself is imported from the module module `MidiFile`, functions for writing it to files are found in the module module `SaveMidi`, and functions for reading MIDI files come from the modules module `LoadMidi` and module `ReadMidi`. All these modules are described later in this section.

4.2.1 The Gory Details

Some preliminaries, otherwise known as constants:

```
division :: MidiFile.Tempo
division = 96    -- time-code division: 96 ticks per quarter note
```

When writing Type 1 Midi Files, we can associate each instrument with a separate track. So first we partition the event list into separate lists for each instrument. (Again, due to the limited number of MIDI channels, we can handle no more than 15 instruments.)

The crux of the conversion process is `trackFromPfBE`, which converts a `Performance.T` into a stream of `Midi.Events`.

As said before, we can't use absolute times, but the difficulties with relatively timed events are handled by the module `Data.EventList.Relative.TimeBody`. We first convert all Performance events to MIDI events preserving the time stamps from the Performance. In the second step we discretize the time stamps with `Data.EventList.Relative.TimeBody.resample`, yielding a perfect `Midi.Track`. On the one hand with this order of execution it may be that notes with equal duration can have slightly different durations in the MIDI file. On the other hand small rests between notes or small overlappings are avoided.[6]

[6] It would be better to define `rate = 4*division`, since this would map a quarter note to `division` ticks, as stated by the MIDI File specification. For compensation `SetTempo` could be set to 250000, meaning a quarter second per quarter note, or one second per whole note.

We manage a module `Map` which stores the active program number of each MIDI channel. If a note on a channel needs a new program or there was no note before, a `ProgChange` is inserted in the stream of MIDI events. The function `updateChannelMap` updates this map each time a note occurs and it returns the MIDI channel for the note and a `Maybe` that contains a program change if necessary.

```
trackFromPfBE :: (NonNeg.C time, RealFrac time) =>
   PerformanceBE.T time MidiNote.T -> MidiFile.Track
trackFromPfBE =
   uncurry TimeList.cons setTempo .
   TimeList.mapBody MidiFileEvent.MIDIEvent .
   TimeList.resample rate .
   TimeList.foldr TimeList.consTime addEvent TimeList.empty .
   progChanges

setTempo :: (MidiFile.ElapsedTime, MidiFileEvent.T)
setTempo =
   (0, MidiFileEvent.MetaEvent
          (MetaEvent.SetTempo MetaEvent.defltTempo))

getChanProg :: MidiNote.T -> (ChannelMsg.Channel, Voice.Program)
getChanProg note = (MidiNote.channel note, MidiNote.program note)

updateChannelMap ::
   (ChannelMsg.Channel, Voice.Program) ->
   Map.Map ChannelMsg.Channel Voice.Program ->
   (Maybe ChannelMsg.T,
       Map.Map ChannelMsg.Channel Voice.Program)
updateChannelMap (midiChan, progNum) cm =
   if Just progNum == Map.lookup cm midiChan
     then (Nothing, cm)
     else (Just (ChannelMsg.Cons midiChan (ChannelMsg.Voice
                     (Voice.ProgramChange progNum))),
           Map.insert midiChan progNum cm)

progChanges ::
   PerformanceBE.T time MidiNote.T
     -> PerformanceBE.T time (MidiNote.T, Maybe ChannelMsg.T)
progChanges =
   flip evalState Map.empty .
   TimeList.mapBodyM
      (\(PerformanceBE.Event dur note) ->
           liftM (\mn -> PerformanceBE.Event dur (note, mn))
              (state (updateChannelMap (getChanProg note))))

rate :: (Num a) => a
rate = 2 * fromIntegral division
--     ^ would be correctly 4 and the setTempo should be 250000
```

A source of incompatibility between Haskore and Midi is that Haskore represents notes with an onset and a duration, while Midi represents them as two separate events, an note-on event and a note-off event. Thus `addEvent` turns a Haskore `Event` into two `ChannelMsg.T`s, a `NoteOn` and a `NoteOff`.

The function `TimeList.insert` is used to insert a `NoteOff` into the sequence of following MIDI events. It looks a bit cumbersome to insert every single `NoteOff`. An alternative may be to `merge` the

list of NoteOn events with the list of NoteOff events. This won't work because the second one isn't ordered. Instead one could merge the two-element lists defined by NoteOn and NoteOff for each note using fold. But there might be infinitely many notes ...

```
addEvent ::
   (NonNeg.C time) =>
   PerformanceBE.Event time
      (MidiNote.T, Maybe ChannelMsg.T) ->
   TimeList.T time ChannelMsg.T ->
   BodyBodyList.T time ChannelMsg.T
addEvent ev mevs =
   let (note, progChange)
         = PerformanceBE.eventNote ev
       d = PerformanceBE.eventDur  ev
       (mec0, mec1) = MidiNote.toMIDIEvents note
   in  maybe (TimeList.consBody mec0)
          (\pcME ->
              TimeList.consBody pcME .
              TimeList.cons NonNeg.zero mec0)
          progChange
          (TimeList.insert d mec1 mevs)
```

The MIDI volume handling is still missing. One cannot express the Volume in terms of the velocity! Thus we need some new event constructor for changed controller values.

```
volumeHaskoreToMIDI :: (RealFrac a, Floating a) => a -> Int
volumeHaskoreToMIDI v = round (limit (0,127) (64 + 16 * logBase 2 v))

volumeMIDIToHaskore :: Floating a => Int -> a
volumeMIDIToHaskore v = 2 ** ((fromIntegral v - 64) / 16)
```

4.2.2 Instrument map

```
module Haskore.Interface.MIDI.InstrumentMap where

import Haskore.Music.Standard(Instr)
import qualified Sound.MIDI.Message.Channel as ChannelMsg
import qualified Sound.MIDI.General         as GeneralMidi

import qualified Haskore.General.Map as Map
import qualified Data.List as List
import Data.Tuple.HT (swap, )
import Data.Char (toLower, )
import Data.Maybe (fromMaybe, )
```

A InstrumentMap.ChannelProgramTable is a user-supplied table for mapping instrument names (Instrs) to Midi channels and General Midi patch names. The patch names are by default General Midi names, although the user can also provide a PatchMap for mapping Patch Names to unconventional Midi Program Change numbers.

```
type ChannelTable instr =
   [(instr,  ChannelMsg.Channel)]
```

```
type ChannelProgramTable instr =
   [(instr, (ChannelMsg.Channel, ChannelMsg.Program))]
type ChannelProgramPitchTable instr =
   [(instr, (ChannelMsg.Channel, ChannelMsg.Program, ChannelMsg.Pitch))]

type ToChannel instr =
   instr ->  ChannelMsg.Channel
type ToChannelProgram instr =
   instr -> (ChannelMsg.Channel, ChannelMsg.Program)
type ToChannelProgramPitch instr =
   instr -> (ChannelMsg.Channel, ChannelMsg.Program, ChannelMsg.Pitch)

type FromChannel instr =
    ChannelMsg.Channel -> Maybe instr
type FromChannelProgram instr =
   (ChannelMsg.Channel, ChannelMsg.Program) -> Maybe instr
type FromChannelProgramPitch instr =
   (ChannelMsg.Channel, ChannelMsg.Program, ChannelMsg.Pitch) -> Maybe instr
```

The `allValid` is used to test whether or not every instrument in a list is found in a `InstrumentMap.ChannelProgramTable`.

```
repair :: [Instr] -> ChannelProgramTable Instr -> ChannelProgramTable Instr
repair insts pMap =
   if allValid pMap insts
       then pMap
       else tableFromInstruments insts

allValid :: ChannelProgramTable Instr -> [Instr] -> Bool
allValid upm = all (\x -> any (partialMatch x . fst) upm)
```

If a Haskore user only uses General Midi instrument names as `Instr`s, we can define a function that automatically creates a `InstrumentMap.ChannelProgramTable` from these names. Note that, since there are only 15 Midi channels plus percussion, we can handle only 15 instruments. Perhaps in the future a function could be written to test whether or not two tracks can be combined with a Program Change (tracks can be combined if they don't overlap).

```
tableFromInstruments :: [Instr] -> ChannelProgramTable Instr
tableFromInstruments instrs =
   zip instrs (assignChannels GeneralMidi.instrumentChannels instrs)
        -- 10th channel (#9) is for percussion

assignChannels :: [ChannelMsg.Channel] -> [Instr] ->
      [(ChannelMsg.Channel, ChannelMsg.Program)]
assignChannels _ [] = []
assignChannels [] _ =
   error "Too many instruments; not enough MIDI channels."
assignChannels chans@(c:cs) (i:is) =
   let percList = ["percussion", "perc", "drum", "drums"]
   in  if map toLower i `elem` percList
         then (GeneralMidi.drumChannel, GeneralMidi.drumProgram)
                    : assignChannels chans is
         else (c, fromMaybe
            (error ("unknown instrument <<" ++ i ++ ">>"))
```

```
            (GeneralMidi.instrumentNameToProgram i))
                    : assignChannels cs is

fromInstruments :: Ord instr => [instr] -> ToChannel instr
fromInstruments instrs =
   let fm = Map.fromList (zip instrs GeneralMidi.instrumentChannels)
   in  Map.findWithDefault fm (error "More instruments than channels")
```

The following functions lookup `Instrs` in `InstrumentMap.ChannelProgramTables` to recover channel and program change numbers. Note that the function that does string matching ignores case, and that instrument name and search pattern match if one is a prefix of the other one. For example, `"chur"` matches `"Church Organ"`. Note also that the *first* match succeeds, so using a substring should be done with care to be sure that the correct instrument is selected.

```
partialMatch :: Instr -> Instr -> Bool
partialMatch "piano" "Acoustic Grand Piano" = True
partialMatch s1 s2 =
  let s1' = map toLower s1
      s2' = map toLower s2
  in  all (uncurry (==)) (zip s1' s2')

lookupIName :: [(Instr, a)] -> Instr -> a
lookupIName  ys x =
   maybe (error ("InstrumentMap.lookupIName: Instrument " ++ x ++ " unknown"))
         snd (List.find (partialMatch x . fst) ys)

lookup :: Eq instr => [(instr, a)] -> instr -> a
lookup ys x =
   fromMaybe (error ("InstrumentMap.lookup: Instrument unknown"))
             (List.lookup x ys)
```

```
reverseLookupMaybe :: Eq a => [(instr, a)] -> a -> Maybe instr
reverseLookupMaybe ys x =
   List.lookup x (map swap ys)

reverseLookup :: Eq a => [(instr, a)] -> a -> instr
reverseLookup ys x =
   let instr = reverseLookupMaybe ys x
       err   = error "InstrumentMap.reverseLookup: channel+program not found"
   in  fromMaybe err instr
```

A default `InstrumentMap.ChannelProgramTable`. Note: the PC sound card I'm using is limited to 9 instruments.

```
defltTable :: [(Instr, ChannelMsg.Channel, GeneralMidi.Instrument)]
defltTable =
   map (\(instr,chan,gmInstr) -> (instr, ChannelMsg.toChannel chan, gmInstr))
   [("piano",   1, GeneralMidi.AcousticGrandPiano),
    ("vibes",   2, GeneralMidi.Vibraphone),
    ("bass",    3, GeneralMidi.AcousticBass),
    ("flute",   4, GeneralMidi.Flute),
    ("sax",     5, GeneralMidi.TenorSax),
    ("guitar",  6, GeneralMidi.AcousticGuitarSteel),
```

```
    ("violin",  7, GeneralMidi.Viola),
    ("violins", 8, GeneralMidi.StringEnsemble1),
    ("drums",   9, GeneralMidi.AcousticGrandPiano)]
      -- the GM name for drums is unimportant, only channel 9

deflt :: ChannelProgramTable Instr
deflt =
   map (\(iName, chan, gmName) ->
         (iName, (chan, GeneralMidi.instrumentToProgram gmName))) defltTable

defltGM :: ChannelProgramTable GeneralMidi.Instrument
defltGM =
   map (\(_, chan, gmName) ->
         (gmName, (chan, GeneralMidi.instrumentToProgram gmName))) defltTable

defltCMap :: [(GeneralMidi.Instrument, ChannelMsg.Channel)]
defltCMap =
   map (\(_, chan, gmName) -> (gmName, chan)) defltTable
```

For a description of the MIDI file type and its loading and saving to disk, see the `midi` package.

4.2.3 Reading Midi files

Now that we have translated a raw Midi file into a `MidiFile.T` data type, we can translate that `MidiFile.T` into a `MidiMusic.T` object.

```
module Haskore.Interface.MIDI.Read (toRhyMusic, toGMMusic,
 {- debugging -} retrieveTracks)
  where

import qualified Haskore.Interface.MIDI.Note          as MidiNote
import qualified Haskore.Interface.MIDI.InstrumentMap as InstrMap
import           Sound.MIDI.File                  as MidiFile
import qualified Sound.MIDI.File.Event            as MidiFileEvent
import qualified Sound.MIDI.Message.Channel       as ChannelMsg
import qualified Sound.MIDI.Message.Channel.Voice as Voice
import qualified Sound.MIDI.General               as GeneralMidi
import Sound.MIDI.File.Event (T(MIDIEvent, MetaEvent), )
import Sound.MIDI.File.Event.Meta (T(SetTempo), defltTempo, )
import Sound.MIDI.Message.Channel (Body(Voice), Channel, )
import Sound.MIDI.Message.Channel.Voice (Program, )

import Haskore.Basic.Duration ((%+))
import qualified Data.EventList.Relative.TimeBody  as TimeList
import qualified Data.EventList.Relative.MixedBody as TimeList
import qualified Haskore.Music             as Music
import qualified Haskore.Music.GeneralMIDI as MidiMusic
import qualified Haskore.Music.Rhythmic    as RhyMusic
import qualified Haskore.Performance.Context  as Context
import qualified Haskore.Performance.BackEnd  as PfBE
import qualified Haskore.Performance.Default  as DefltPf
import qualified Haskore.Process.Optimization as Optimization

import qualified Numeric.NonNegative.Class as NonNeg
```

```
import Haskore.Music
             (line, chord, changeTempo, Dur, DurRatio)
import Data.Tuple.HT (mapPair, mapSnd, )
import qualified Data.List.HT as ListHT

import Haskore.General.Map (Map)
import qualified Haskore.General.Map as Map
import Data.Maybe (mapMaybe, fromMaybe)
```

The main function. Note that we need drum and instrument maps in order to restore a `Context.T` as well as a `RhyMusic.T` object.

```
toRhyMusic ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   InstrMap.ChannelProgramPitchTable drum ->
   InstrMap.ChannelProgramTable instr ->
   MidiFile.T ->
      (Context.T time dyn (RhyMusic.Note drum instr), RhyMusic.T drum instr)
toRhyMusic dMap iMap mf@(MidiFile.Cons _ d trks) =
  let cpm = makeCPM trks
      m   = Music.mapNote
               (MidiNote.toRhyNote
                  (InstrMap.reverseLookupMaybe dMap)
                  (InstrMap.reverseLookupMaybe iMap))
               (format (readFullTrack d cpm) (MidiFile.explicitNoteOff mf))
  in (context, m)

toGMMusic ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   MidiFile.T -> (InstrMap.ChannelTable MidiMusic.Instr,
                  Context.T time dyn MidiMusic.Note, MidiMusic.T)
toGMMusic mf@(MidiFile.Cons _ d trks) =
  let cpm     = makeCPM trks
      upm     = map (\(ch, progNum) ->
                       (GeneralMidi.instrumentFromProgram progNum, ch))
                    (Map.toList cpm)
      m       = Music.mapNote MidiNote.toGMNote
                   (format (readFullTrack d cpm)
                           (MidiFile.explicitNoteOff mf))
  in (upm, context, m)

context ::
   (NonNeg.C time, Fractional time, Real time, Fractional dyn) =>
   Context.T time dyn note
context =
   Context.setPlayer DefltPf.player $
   Context.setDur 2 $
   DefltPf.context

retrieveTracks :: MidiFile.T -> [[MidiMusic.T]]
retrieveTracks (MidiFile.Cons _ d trks) =
  let cpm = makeCPM trks
  in  map (map (Music.mapNote MidiNote.toGMNote
                . readTrack (MidiFile.ticksPerQuarterNote d) cpm . fst)
```

```
             . prepareTrack) trks

type ChannelProgramMap = Map ChannelMsg.Channel Voice.Program

readFullTrack ::
   Division -> ChannelProgramMap -> Track -> Music.T MidiNote.T
readFullTrack dv cpm =
  let readTempoTrack (t,r) =
        changeTempo r (readTrack (MidiFile.ticksPerQuarterNote dv) cpm t)
  in  Optimization.all . line . map readTempoTrack . prepareTrack

prepareTrack :: Track -> [(RichTrack, DurRatio)]
prepareTrack =
   map (extractTempo defltTempo) . segmentBeforeSetTempo .
   mergeNotes defltTempo . moveTempoToHead
```

Make one big music out of the individual tracks of a MidiFile, using different composition types depending on the format of the MidiFile.

```
format :: (Track -> Music.T note) -> MidiFile.T -> Music.T note
format tm (MidiFile.Cons typ _ trks) =
   let trks' = map tm trks
   in  case typ of
         MidiFile.Mixed ->
            case trks' of
               [trk] -> trk
               _ -> error ("toRhyMusic: Only one track allowed for MIDI file type 0.")
         MidiFile.Parallel -> chord trks'
         MidiFile.Serial   -> line  trks'
```

Look for Program Changes in the given tracks, in order to make a `ChannelProgramMap`.

```
makeCPM :: [Track] -> ChannelProgramMap
makeCPM =
   Map.fromList . concatMap (mapMaybe getPC . TimeList.getBodies)

getPC :: MidiFileEvent.T -> Maybe (Channel, Program)
getPC ev =
   do (ch, Voice.ProgramChange num) <- MidiFileEvent.maybeVoice ev
      Just (ch, num)
```

`moveTempoToHead` gets the information that occurs at the beginning of the piece: the default tempo and the default key signature. A `SetTempo` in the middle of the piece should translate to a tempo change (`Tempo r m`), but a `SetTempo` at time 0 should set the default tempo for the entire piece, by translating to `Context.T` tempo. It remains a matter of taste which tempo of several parallel tracks to use for the whole music. `moveTempoToHead` takes care of all events that occur at time 0 so that if any `SetTempo` appears at time 0, it is moved to the front of the list, so that it can be easily retrieved from the result of `segmentBeforeSetTempo`.

```
moveTempoToHead :: Track -> Track
moveTempoToHead es =
   let (tempo, track) = getHeadTempo es
   in  TimeList.cons 0 (MetaEvent (SetTempo tempo)) track
```

```
getHeadTempo :: Track -> (Tempo, Track)
getHeadTempo es =
   maybe
      (defltTempo, es)
      (\ ~(me,rest) ->
          case me of
             MetaEvent (SetTempo tempo) -> (tempo, rest)
             _ -> mapSnd (TimeList.cons 0 me) (getHeadTempo rest))
      (do ((0,me),rest) <- TimeList.viewL es
          return (me,rest))
```

Manages the tempo changes in the piece. It translates each MidiFile `SetTempo` into a ratio between the new tempo and the tempo at the beginning.

```
extractTempo :: Tempo -> RichTrack -> (RichTrack, DurRatio)
extractTempo d trk =
   fromMaybe
      (trk, 1)
      (do ((_, Event (MetaEvent (SetTempo tempo))), rest) <- TimeList.viewL trk
          return (rest, toInteger d %+ toInteger tempo))
```

`segmentBefore` is used to split a track into sub-tracks by tempo. We do not want to add this function to the `event-list` package, because the precise type would be `AlternatingList.Disparate (TimeList.T time body) (TimeList.Event time body)` and that's inconvenient for our application.

```
segmentBefore ::
   (body -> Bool) -> TimeList.T time body -> [TimeList.T time body]
segmentBefore p =
   map TimeList.fromPairList .
   ListHT.segmentBefore (p . snd) .
   TimeList.toPairList
```

```
isSetTempo :: RichEvent -> Bool
isSetTempo (Event (MetaEvent (SetTempo _))) = True
isSetTempo _                                = False

segmentBeforeSetTempo :: RichTrack -> [RichTrack]
segmentBeforeSetTempo = segmentBefore isSetTempo
```

`readTrack` is the heart of the `toRhyMusic` operation. It reads a track that has been processed by `mergeNotes`, and returns the track as `StdMusic.T`. A `RichEvent` consists either of a normal `MIDIEvent` or of a note, which in contrast to normal `MIDIEvents` contains the information of corresponding `NoteOn` and `NoteOff` events.

```
type RichTrack = TimeList.T ElapsedTime RichEvent
data RichEvent =
     Event MidiFileEvent.T
   | Note  ElapsedTime MidiNote.T

readTrack :: Tempo -> ChannelProgramMap ->
   RichTrack -> Music.T MidiNote.T
```

```
readTrack ticksPerQN cpm =
   PfBE.toMusic . trackTimeToStd ticksPerQN
    . richTrackToBE . applyProgChanges cpm
```

Take the division in ticks per quarterNote and a duration value in number of ticks and converts that to a common note duration (such as quarter note, eighth note, etc.).

```
fromTicks :: Tempo -> ElapsedTime -> Dur
fromTicks ticksPerQN d =
   toInteger d %+ (toInteger ticksPerQN * quarter)

quarter :: Integer
quarter = 4

trackTimeToStd :: Tempo ->
   PfBE.T ElapsedTime note -> PfBE.T Dur note
trackTimeToStd ticksPerQN =
   TimeList.mapBody
      (\(PfBE.Event d n) -> PfBE.Event (fromTicks ticksPerQN d) n)
      . TimeList.mapTime (fromTicks ticksPerQN)
```

Look up an instrument name from a `ChannelProgramMap` given its channel number.

```
lookupChannelProg :: ChannelProgramMap -> Channel -> Program
lookupChannelProg cpm =
   Map.findWithDefault cpm
      (error "Invalid channel in user patch map")
```

Implement a *Program Change*: a change in the `ChannelProgramMap` in which a channel changes from one instrument to another.

```
progChange :: Channel -> Program -> ChannelProgramMap -> ChannelProgramMap
progChange = Map.insert
-- progChange ch num cpm = Map.insert ch num cpm
```

Process all `ProgramChange` events in a track. That is, manage a patch map and insert in the appropriate program numbers into the `MidiNote.Ts`.

The function works the following way: Split the track into pieces, each beginning with a program change. Compute the patch maps that are active after each program change. Apply these patch maps to the track parts.

```
isProgChange :: RichEvent -> Bool
isProgChange (Event ev) =
   maybe False (const True) (getPC ev)
isProgChange _ = False

applyProgChanges :: ChannelProgramMap -> RichTrack -> RichTrack
applyProgChanges cpm track =
   let parts@(_:pcParts) = segmentBefore isProgChange track
{-
       updateCPM (Event (MIDIEvent ch (ProgramChange prog))) =
          progChange ch prog
       updateCPM _  =  error "TimeList.collectCoincident is buggy"
```

```
-}
       updateCPM =
          TimeList.switchL
             (error "TimeList.collectCoincident is buggy")
             (\ (_, Event ev) _ ->
                 maybe
                    (error "after segmentation, each part should start with ProgramChange event")
                    (uncurry progChange)
                    (getPC ev))
       cpms =
          scanl (flip id) cpm (map updateCPM pcParts)
       setProg localCPM (Note d n) =
          Note d (n{MidiNote.program =
                       lookupChannelProg localCPM (MidiNote.channel n)})
       setProg _ e = e
   in  TimeList.concat (zipWith (TimeList.mapBody . setProg) cpms parts)
```

Remove meta events from `RichTrack`, thus converting to a back-end performance.

```
richNoteToBE :: RichEvent -> PfBE.Event ElapsedTime MidiNote.T
richNoteToBE (Note d n) = PfBE.Event d n
richNoteToBE _ = error "richNoteToBE: only Note constructor allowed"

isRichNote :: RichEvent -> Bool
isRichNote (Note _ _) = True
isRichNote _          = False

richTrackToBE :: RichTrack -> PfBE.T ElapsedTime MidiNote.T
richTrackToBE =
   TimeList.mapBody richNoteToBE . fst
      . TimeList.partition isRichNote
```

The `mergeNotes` function changes the order of the events in a track so that they can be handled by readTrack: each `NoteOff` is put directly after its corresponding `NoteOn`. Its first and second arguments are the elapsed time and value (in microseconds per quarter note) of the `SetTempo` currently in effect.

```
mergeNotes :: Tempo -> Track -> RichTrack
mergeNotes stv =
   TimeList.mapTimeTail
      (TimeList.switchBodyL $ \ e rest ->
           uncurry TimeList.consBody $
           let deflt = (Event e, mergeNotes stv rest)
           in  case e of
                  MetaEvent (SetTempo newStv) ->
                     (Event e, mergeNotes newStv rest)
                  MIDIEvent chmsg@(ChannelMsg.Cons _ (Voice msg)) ->
                     if Voice.isNoteOn msg
                       then mapPair
                               (uncurry Note, mergeNotes stv)
                               (searchNoteOff 0 stv 1 chmsg rest)
                       else
                         if Voice.isNoteOff msg
                           then error "NoteOff before NoteOn"
                           else deflt
                  _ -> deflt)
```

The function `searchNoteOff` takes a track and looks through the list of events to find the `NoteOff` corresponding to the given `NoteOn`. A `NoteOff` corresponds to an earlier `NoteOn` if it is the first in the track to have the same channel and pitch. If between `NoteOn` and `NoteOff` are `SetTempo` events, it calculates what the elapsed-time is, expressed in the current tempo. This function takes a ridiculous number of arguments, I know, but I don't think it can do without any of the information. Maybe there is a simpler way.

```
searchNoteOff ::
      Double          {- ^ time interval between NoteOn and now,
                           in terms of the tempo at the NoteOn -}
   -> Tempo -> Double {- ^ SetTempo values: the one at the NoteOn and
                           the ratio between the current tempo and the first one. -}
   -> ChannelMsg.T    {- ^ channel and pitch of NoteOn (NoteOff must match) -}
   -> Track           {- ^ the track to be searched -}
   -> ((ElapsedTime, MidiNote.T), Track)
                      -- ^ the needed event and the remainder of the track

searchNoteOff int ost str chm0 =
   TimeList.switchL
      (error "ReadMidi.searchNoteOff: no corresponding NoteOff")
      (\(t1, mev1) es ->
          maybe
             -- if MIDI events don't match, then recourse
             (mapSnd (TimeList.cons t1 mev1) $
              searchNoteOff (addInterval str t1 int) ost
                 (case mev1 of
                    -- respect tempo changes
                    MetaEvent (SetTempo nst) ->
                          fromIntegral ost / fromIntegral nst
                    _ -> str)
                 chm0 es)
             -- if MIDI events match, construct a MidiNote.T
             (\note ->
                let d = round (addInterval str t1 int)
                in  ((d, note), TimeList.delay t1 es))
             -- check whether NoteOn and NoteOff matches
             (do chm1 <- MidiFileEvent.maybeMIDIEvent mev1
                 MidiNote.fromMIDIEvents (chm0, chm1)))

addInterval :: Double -> ElapsedTime -> Double -> Double
addInterval str t int = int + fromIntegral t * str
```

4.3 CSound

```
module Haskore.Interface.CSound where
```

[Note: if this module is loaded into Hugs98, the following error message may result:

```
  ERROR "CSound.lhs" (line 707):
  *** Cannot derive Eq OrcExp after 40 iterations.
  *** This may indicate that the problem is undecidable.  However,
  *** you may still try to increase the cutoff limit using the -c
```

```
  *** option and then try again.  (The current setting is -c40)
```

This is apparently due to the size of the `OrcExp` data type. For correct operation, start Hugs with a larger cutoff limit, such as `-c1000`.]

CSound is a software synthesizer that allows its user to create a virtually unlimited number of sounds and instruments. It is extremely portable because it is written entirely in C. Its strength lies mainly in the fact that all computations are performed in software, so it is not reliant on sophisticated musical hardware. The output of a CSound computation is a file representing the signal which can be played by an independent application, so there is no hard upper limit on computation time. This is important because many sophisticated signals take much longer to compute than to play. The purpose of this module is to create an interface between Haskore and CSound in order to give the Haskore user access to all the powerful features of a software sound synthesizer.

CSound takes as input two plain text files: a *score* (.sco) file and an *orchestra* (.orc) file. The score file is similar to a Midi file, and the orchestra file defines one or more *instrument*s that are referenced from the score file (the orchestra file can thus be thought of as the software equivalent of Midi hardware). The CSound program takes these two files as input, and produces a *sound file* as output, usually in `.wav` format. Sound files are generally much larger than Midi files, since they describe the actual sound to be generated, represented as a sequence of values (typically 44,100 of them for each second of music), which are converted directly into voltages that drive the audio speakers. Sound files can be played by any standard media player found on conventional PC's.

Each of these files is described in detail in the following sections.

Here are some common definitions:

```
newtype Instrument = Instrument Int
   deriving (Show, Eq)

instrument :: Int -> Instrument
instrument = Instrument

instruments :: [Instrument]
instruments = map instrument [1..]

instrumentToNumber :: Instrument -> Int
instrumentToNumber (Instrument n) = n

showInstrumentNumber :: Instrument -> String
showInstrumentNumber = show . instrumentToNumber

type Name = String

type Velocity  = Float
type PField    = Float
type Time      = Float
```

4.3.1 The Score File

```
module Haskore.Interface.CSound.Score where

import Haskore.Interface.CSound (Instrument, showInstrumentNumber, PField, Time)
import qualified Haskore.Interface.CSound.Note as CSNote
import qualified Haskore.Interface.CSound.Generator as Generator
import Haskore.Interface.CSound.Generator
          (compSine1, lineSeg1, randomTable, PStrength, RandDist(Uniform))

import qualified Haskore.Music.Rhythmic        as RhyMusic
import qualified Haskore.Performance           as Performance
import qualified Haskore.Performance.BackEnd   as PerformanceBE
import qualified Haskore.Performance.Context   as Context
import qualified Haskore.Performance.Fancy     as FancyPf
import qualified Data.EventList.Relative.TimeBody as TimeList
import qualified Data.EventList.Absolute.TimeBody as TimeListAbs
import qualified Haskore.Basic.Pitch           as Pitch
import qualified Haskore.Interface.CSound.InstrumentMap as InstrMap
import qualified Haskore.Interface.CSound.SoundMap as SoundMap

import qualified Numeric.NonNegative.Class as NonNeg
```

We will represent a score file as a sequence of *score statements*:

```
type T = [Statement]
```

The `Statement` data type is designed to simulate CSound's three kinds of score statements:

1. A *tempo* statement, which sets the tempo. In the absence of a tempo statement, the tempo defaults to 60 beats per minute.

2. A *note event*, which defines the start time, pitch, duration (in beats), volume (in decibels), and instrument to play a note (and is thus more like a Haskore `Event` than a Midi event, thus making the conversion to CSound easier than to Midi, as we shall see later). Each note event also contains a number of optional arguments called *p-fields*, which determine other properties of the note, and whose interpretation depends on the instrument that plays the note. This will be discussed further in a later section.

3. *Function table* definitions. A function table is used by instruments to produce audio signals. For example, sequencing through a table containing a perfect sine wave will produce a very pure tone, while a table containing an elaborate polynomial will produce a complex sound with many overtones. The tables can also be used to produce control signals that modify other signals. Perhaps the simplest example of this is a tremolo or vibrato effect, but more complex sound effects, and FM (frequency modulation) synthesis in general, is possible.

```
data Statement = Tempo Bpm
               | Note  Instrument StartTime Duration Pch Volume [PField]
               | Table Table CreatTime TableSize Normalize Generator.T
    deriving Show

type Bpm       = Int
```

```
type StartTime = Time
type Duration  = Time
data Pch       = AbsPch Pitch.Absolute | Cps Float deriving Show
type Volume    = Float
type Table     = Int
type CreatTime = Time
type TableSize = Int
type Normalize = Bool
```

This is all rather straightforward, except for function table generation, which requires further explanation.

Function Tables Each function table must have a unique integer ID (`Table`), creation time (usually 0), size (which must be a power of 2), and a `Normalize` flag. Most tables in CSound are normalized, i.e. rescaled to a maximum absolute value of 1. The normalization process can be skipped by setting the `Normalize` flag to `False`. Such a table may be desirable to generate a control or modifying signal, but is not very useful for audio signal generation.

Tables are simply arrays of floating point values. The values stored in the table are calculated by one of CSound's predefined *generating routines*, represented by the type `Generator.T`:

```
module Haskore.Interface.CSound.Generator where

import Haskore.Interface.CSound (Time)
import Haskore.General.Utility
   (flattenTuples2, flattenTuples3, flattenTuples4)

data T = Routine Number [Parameter]
       | SoundFile SFName SkipTime ChanNum
     deriving Show

type SFName    = String
type SkipTime  = Time
type ChanNum   = Float
type Number    = Int
type Parameter = Float
```

`Routine n args` refers to CSound's generating routine n (an integer), called with floating point arguments `args`. There is only one generating routine (called GEN01) in CSound that takes an argument type other than floating point, and thus we represent this using the special constructor `SoundFile`, whose functionality will be described shortly.

Knowing which of CSound's generating routines to use and with what arguments can be a daunting task. The newest version of CSound (version 4.01) provides 23 different generating routines, and each one of them assigns special meanings to its arguments. To avoid having to reference routines using integer ids, the following functions are defined for the most often-used generating routines. A brief discussion of each routine is also included. For a full description of these and other routines, refer to the CSound manual or consult the following webpage: `http://www.leeds.ac.uk/music/Man/Csound/Function/GENS.html`. The user familiar with CSound is free to write helper functions like the ones below to capture other generating routines.

GEN01. Transfers data from a soundfile into a function table. Recall that the size of the function table in CSound must be a power of two. If the soundfile is larger than the table size, reading stops when the table is full; if it is smaller, then the table is padded with zeros. One exception is allowed: if the file is of type AIFF and the table size is set to zero, the size of the function table is allocated dynamically as the number of points in the soundfile. The table is then unusable by normal oscillators, but can be used by a special `SampOsc` constructor (discussed in Section 4.3.2). The first argument passed to the GEN01 subroutine is a string containing the name of the source file. The second argument is skip time, which is the number of seconds into the file that the reading begins. Finally there is an argument for the channel number, with 0 meaning read all channels. GEN01 is represented in Haskore as `SoundFile SFName SkipTime ChanNum`, as discussed earlier. To make the use of `SoundFile` consistent with the use of other functions to be described shortly, we define a simple equivalent:

```
soundFile :: SFName -> SkipTime -> ChanNum -> T
soundFile = SoundFile
```

GEN02. Transfers data from its argument fields directly into the function table. We represent its functionality as follows:

```
tableValues :: [Parameter] -> T
tableValues gas = Routine 2 gas
```

GEN03. Fills the table by evaluating a polynomial over a specified interval and with given coefficients. For example, calling GEN03 with an interval of $(-1, 1)$ and coefficients 5, 4, 3, 2, 0, 1 will generate values of the function $5 + 4x + 3x^2 + 2x^3 + x^5$ over the interval -1 to 1. The number of values generated is equal to the size of the table. Let's express this by the following function:

```
polynomial :: Interval -> Coefficients -> T
polynomial (x1,x2) cfs = Routine 3 (x1:x2:cfs)

type Interval     = (Float, Float)
type Coefficients = [Float]
```

GEN05. Constructs a table from segments of exponential curves. The first argument is the starting point. The meaning of the subsequent arguments alternates between the length of a segment in samples, and the endpoint of the segment. The endpoint of one segment is the starting point of the next. The sum of all the segment lengths normally equals the size of the table: if it is less the table is padded with zeros, if it is more, only the first `TableSize` locations will be stored in the table.

```
exponential1 :: StartPt -> [(SegLength, EndPt)] -> T
exponential1 sp xs = Routine 5 (sp : flattenTuples2 xs)

type StartPt   = Float
type SegLength = Float
type EndPt     = Float
```

GEN25. Similar to GEN05 in that it produces segments of exponential curves, but instead of representing the lengths of segments and their endpoints, its arguments represent (x, y) coordinates in the table, and the subroutine produces curves between successive locations. The x-coordinates must be in increasing order.

```
exponential2 :: [Point] -> T
exponential2 pts = Routine 25 (flattenTuples2 pts)

type Point = (Float,Float)
```

GEN06. Generates a table from segments of cubic polynomial functions, spanning three points at a time. We define a function `cubic` with two arguments: a starting position and a list of segment length (in number of samples) and segment endpoint pairs. The endpoint of one segment is the starting point of the next. The meaning of the segment endpoint alternates between a local minimum/maximum and point of inflexion. Whether a point is a maximum or a minimum is determined by its relation to the next point of inflexion. Also note that for two successive minima or maxima, the inflexion points will be jagged, whereas for alternating maxima and minima, they will be smooth. The slope of the two segments is independent at the point of inflection and will likely vary. The starting point is a local minimum or maximum (if the following point is greater than the starting point, then the starting point is a minimum, otherwise it is a maximum). The first pair of numbers will in essence indicate the position of the first inflexion point in (x, y) coordinates. The folowing pair will determine the next local minimum/maximum, followed by the second point of inflexion, etc.

```
cubic ::  StartPt -> [(SegLength, EndPt)] -> T
cubic sp pts = Routine 6 (sp : flattenTuples2 pts)
```

GEN07. Similar to GEN05, except that it generates straight lines instead of exponential curve segments. All other issues discussed about GEN05 also apply to GEN07. We represent it as:

```
lineSeg1 :: StartPt -> [(SegLength, EndPt)] -> T
lineSeg1 sp pts = Routine 7 (sp : flattenTuples2 pts)
```

GEN27. As with GEN05 and GEN25, produces straight line segments between points whose locations are given as (x, y) coordinates, rather than a list of segment length, endpoint pairs.

```
lineSeg2 :: [Point] -> T
lineSeg2 pts = Routine 27 (flattenTuples2 pts)
```

GEN08. Produces a smooth piecewise cubic spline curve through the specified points. Neighboring segments have the same slope at the common points, and it is that of a parabola through that point and its two neighbors. The slope is zero at the ends.

```
cubicSpline :: StartPt -> [(SegLength, EndPt)] -> T
cubicSpline sp pts = Routine 8 (sp : flattenTuples2 pts)
```

GEN10. Produces a composite sinusoid. It takes a list of relative strengths of harmonic partials 1, 2, 3, etc. Partials not required should be given strength of zero.

```
compSine1 :: [PStrength] -> T
compSine1 pss = Routine 10 pss

type PStrength = Float
```

GEN09. Also produces a composite sinusoid, but requires three arguments to specify each contributing partial. The arguments specify the partial number, which doesn't have to be an integer (i.e. inharmonic partials are allowed), the relative partial strength, and the initial phase offset of each partial, expressed in degrees.

```
compSine2 :: [(PNum, PStrength, PhaseOffset)] -> T
compSine2 args = Routine 9 (flattenTuples3 args)

type PNum = Float
type PhaseOffset = Float
```

GEN19. Provides all of the functionality of GEN09, but in addition a DC offset must be specified for each partial. The DC offset is a vertical displacement, so that a value of 2 will lift a 2-strength partial from range $[-2, 2]$ to range $[0, 4]$ before further scaling.

```
compSine3 :: [(PNum, PStrength, PhaseOffset, DCOffset)] -> T
compSine3 args = Routine 19 (flattenTuples4 args)

type DCOffset = Float
```

GEN11. Produces an additive set of harmonic cosine partials, similar to GEN10. We will represent it by a function that takes three arguments: the number of harmonics present, the lowest harmonic present, and a multiplier in an exponential series of harmonics amplitudes (if the x'th harmonic has strength coefficient of A, then the $(x + n)$'th harmonic will have a strength of $A * (r^n)$, where r is the multiplier).

```
cosineHarms :: NHarms -> LowestHarm -> Mult -> T
cosineHarms n l m = Routine 11 [fromIntegral n, fromIntegral l, m]

type NHarms = Int
type LowestHarm = Int
type Mult = Float
```

GEN21. Produces tables having selected random distributions.

```
randomTable :: RandDist -> T
randomTable rd = Routine 21 [fromIntegral (fromEnum rd + 1)]

data RandDist =
       Uniform
```

```
    | Linear
    | Triangular
    | Expon
    | BiExpon
    | Gaussian
    | Cauchy
    | PosCauchy
  deriving (Eq, Ord, Enum, Show)
```

```
toStatementWords :: T -> [String]
toStatementWords (Routine gn gas)     = show gn : map show gas
toStatementWords (SoundFile nm st cn) = ["1", nm, show st, "0", show cn]
```

Common Tables For convenience, here are some common function tables, which take as argument the identifier integer:

```
simpleSine, square, sawtooth, triangle, whiteNoise :: Table -> Statement

simpleSine n = Table n 0 8192 True
                     (compSine1 [1])
square     n = Table n 0 1024 True
                     (lineSeg1 1 [(256, 1), (0, -1), (512, -1), (0, 1), (256, 1)])
sawtooth   n = Table n 0 1024 True
                     (lineSeg1 0 [(512, 1), (0, -1), (512, 0)])
triangle   n = Table n 0 1024 True
                     (lineSeg1 0 [(256, 1), (512, -1), (256, 0)])
whiteNoise n = Table n 0 1024 True
                     (randomTable Uniform)
```

The following function for a composite sine has an extra argument, a list of harmonic partial strengths:

```
compSine :: Table -> [PStrength] -> Statement
compSine _ s = Table 6 0 8192 True (compSine1 s)
```

Naming Instruments and Tables In CSound, each table and instrument has a unique identifying integer associated with it. Haskore, on the other hand, uses strings to name instruments. What we need is a way to convert Haskore instrument names to identifier integers that CSound can use. Similar to Haskore's player maps, we define a notion of a *CSound name map* for this purpose.

```
module Haskore.Interface.CSound.InstrumentMap where

import Haskore.Interface.CSound (PField, Instrument, instruments)

import qualified Data.List as List

type SoundTable instr = [(instr, Instrument)]
```

A name map can be provided directly in the form `[("name1", int1), ("name2", int2), ...]`, or the programmer can define auxiliary functions to make map construction easier. For example:

```
tableFromInstruments :: [instr] -> SoundTable instr
tableFromInstruments nms = zip nms $ instruments
```

The following function will add a name to an existing name map. If the name is already in the map, an error results.

```
addToTable :: (Eq instr) =>
   instr -> Instrument -> SoundTable instr -> SoundTable instr
addToTable nm i instrMap =
   if elem nm (map fst instrMap)
     then ((nm,i) : instrMap)
     else (error ("CSound.addToTable: instrument already in the map"))
```

Note the use of the function `lookup` imported from module `List`.

```
type ToSound instr = instr -> ([PField], Instrument)

lookup :: (Eq instr) => SoundTable instr -> ToSound instr
lookup table instr =
   maybe (error "CSound.InstrMap.lookup: instrument not found")
         ((,) [])
         (List.lookup instr table)
```

Converting Haskore Music.T to a CSound Score File To convert a `Music.T` value into a CSound score file, we need to:

1. Convert the `Music.T` value to a `Performance.T`.
2. Convert the `Performance.T` value to a `Score.T`.
3. Write the `Score.T` value to a CSound score file.

We already know how to do the first step. Steps two and three will be achieved by the following two functions:

```
fromPerformanceBE :: (NonNeg.C time, Num time) =>
   (time -> Time) ->
   PerformanceBE.T time CSNote.T -> T

saveIA :: T -> IO ()
```

The three steps can be put together in whatever way the user wishes, but the most general way would be this:

```
fromRhythmicMusic ::
   (RealFrac time, NonNeg.C time, RealFrac dyn, Ord drum, Ord instr) =>
      Tables ->
      (InstrMap.SoundTable drum,
       InstrMap.SoundTable instr,
       Context.T time dyn (RhyMusic.Note drum instr),
```

```
       RhyMusic.T drum instr) -> T
fromRhythmicMusic tables (dMap, iMap, cont, m) =
   tables ++ fromRhythmicPerformance dMap iMap
                (Performance.fromMusic FancyPf.map cont m)

type Tables = T
```

The `Tables` argument is a user-defined set of function tables, represented as a sequence of `Statements` (specifically, `Table` constructors). (See Section 4.3.1.)

From Performance.T to Score.T The translation between `Performance.Events` and score `CSoundScore.Notes` is straightforward, the only tricky part being:

- The unit of time in a `Performance.T` is the second, whereas in a `Score.T` it is the beat. However, the default CSound tempo is 60 beats per minute, or one beat per second, as was already mentioned, and we use this default for our *score* files. Thus the two are equivalent, and no translation is necessary.
- CSound wants to get pitch information in the form 'a.b' but it interprets them very different. Sometimes it is considered as 'octave.pitchclass' sometimes it is considered as fraction frequency. We try to cope with it using the two-constructor type Pch.
- Like for MIDI data we must distinguish between Velocity and Volume. Velocity is instrument dependent and different velocities might result in different flavors of a sound. As a quick work-around we turn the velocity information into volume. Cf. `dbamp` in the CSound manual.

```
fromPerformanceBE timeMap =
   map (\(time, event) ->
      noteToStatement timeMap time
         (PerformanceBE.eventDur  event)
         (PerformanceBE.eventNote event)) .
   TimeListAbs.toPairList .
   TimeList.toAbsoluteEventList NonNeg.zero

fromRhythmicPerformance ::
   (RealFrac time, NonNeg.C time, RealFrac dyn, Ord drum, Ord instr) =>
   InstrMap.SoundTable drum ->
   InstrMap.SoundTable instr ->
      Performance.T time dyn (RhyMusic.Note drum instr) -> T
fromRhythmicPerformance dMap iMap =
   fromPerformanceBE realToFrac .
   PerformanceBE.fromPerformance
      (CSNote.fromRhyNote
         (InstrMap.lookup dMap)
         (InstrMap.lookup iMap))

fromRhythmicPerformanceMap ::
   (RealFrac time, NonNeg.C time, RealFrac dyn) =>
   InstrMap.ToSound drum ->
   InstrMap.ToSound instr ->
      Performance.T time dyn (RhyMusic.Note drum instr) -> T
fromRhythmicPerformanceMap dMap iMap =
```

```
   fromPerformanceBE realToFrac .
   PerformanceBE.fromPerformance (CSNote.fromRhyNote dMap iMap)

fromRhythmicPerformanceWithAttributes ::
   (RealFrac time, NonNeg.C time, RealFrac dyn) =>
   SoundMap.DrumTableWithAttributes out drum ->
   SoundMap.InstrumentTableWithAttributes out instr ->
      Performance.T time dyn (RhyMusic.Note drum instr) -> T
fromRhythmicPerformanceWithAttributes dMap iMap =
   fromRhythmicPerformanceMap
      (SoundMap.lookupDrum dMap)
      (SoundMap.lookupInstrument iMap)

noteToStatement ::
   (time -> Time) -> time -> time ->
      CSNote.T -> Statement
noteToStatement timeMap t d (CSNote.Cons pfs v i p) =
   Note i (timeMap t) (timeMap d)
          (maybe (Cps 0 {- dummy -}) AbsPch p) v pfs
```

From Score to Score File Now that we have a value of type `Score`, we must write it into a plain text ASCII file with an extension `.sco` in a way that CSound will recognize. This is done by the following function:

```
saveIA s =
   do putStr "\nName your score file "
      putStr "(.sco extension will be added): "
      name   <- getLine
      save (name ++ ".sco") s

save :: FilePath -> T -> IO ()
save name s = writeFile (name ++ ".sco") (toString s)
```

This function asks the user for the name of the score file, opens that file for writing, writes the score into the file using the function `toString`, and then closes the file.

The score file is a plain text file containing one statement per line. Each statement consists of an opcode, which is a single letter that determines the action to be taken, and a number of arguments. The opcodes we will use are “e” for end of score, “t” to set tempo, “f” to create a function table, and “i” for note events.

```
toString :: T -> String
toString s = unlines (map statementToString s ++ ["e"])   -- end of score
```

Finally, the `statementToString` function:

```
statementToString :: Statement -> String
statementToString = unwords . statementToWords

statementToWords :: Statement -> [String]
statementToWords (Tempo t) =
   ["t", "0", show t]
statementToWords (Note i st d p v pfs) =
```

```
   ["i", showInstrumentNumber i, show st, show d,
    pchToString p, show v] ++ map show pfs
statementToWords (Table t ct s n gr)   =
   ["f", show t, show ct, show s,
    (if n then id else ('-':))
       (unwords (Generator.toStatementWords gr))]

-- it's exciting whether CSound knows what we mean with the values
-- (0 < note) is for compatibility with older CSound example files
pchToString :: Pch -> String
pchToString (AbsPch ap) =
   let (oct, note) = divMod ap 12
   in  show oct ++ "." ++
       (if 0 < note && note < 10 then "0" else "") ++
       show note
pchToString (Cps freq) = show freq
```

4.3.2 The Orchestra File

```
module Haskore.Interface.CSound.Orchestra (
   T(Cons), InstrBlock(..), Header, AudRate, CtrlRate,
   -- SigTerm(ConstFloat, ConstInt, TableNumber, PField, Str,
   --         Read, Tap, Result, Conditional,
   --         Infix, Prefix, SigGen),
   SigExp, DelayLine, Boolean,
   -- DelayLine(DelayLine), Boolean(Operator, Comparison),
   GlobalSig(Global), Output(..), Mono(Mono), Stereo(Stereo), Quad(Quad),
   EvalRate(NR, CR, AR), Instrument, Name,
   sigGen, tableNumber, readGlobal, rec,

   -- assorted functions
   toString, saveIA, save,
   channelCount, getMultipleOutputs,

   -- variables dealing with PFields
   noteDur, notePit, noteVel, p1, p2, p3, p4, p5, p6, p7, p8, p9, pField,

   -- functions for dealing with Booleans and Conditional SigExps
   (<*), (<=*), (>*), (>=*), (==*), (/=*), (&&*), (||*), ifthen,
   constInt, constFloat, constEnum,

   -- functions for creating signal expressions
   pchToHz, dbToAmp, line, expon, lineSeg, exponSeg, env, phasor,
   IndexMode(..), tblLookup, tblLookupI, osc, oscI,
   fmOsc, fmOscI, sampOsc, random, randomH, randomI, genBuzz, buzz,
   pluck, PluckDecayMethod(..), delay, vdelay, comb, alpass, reverb,
   delTap, delTapI,

   -- monad-related functions
   Orc, mkSignal, addInstr, mkOrc,

   -- assorted examples
   orc1, test, test1) where
```

```
import Haskore.Interface.CSound
          (Name, Instrument, instrument, instruments, showInstrumentNumber)
import Haskore.Interface.CSound.OrchestraFunction

import qualified Haskore.General.LoopTreeRecursiveGen as TreeRec
import qualified Haskore.General.LoopTreeTaggedGen    as TreeTag

import Control.Monad.Trans.State (State, state, modify, execState, )
import Control.Applicative (liftA, liftA2, liftA3, pure)
import Data.Foldable (Foldable(foldMap))
import Data.Traversable (Traversable(sequenceA))
import qualified Data.Traversable as Traversable

import Haskore.General.Utility (flattenTuples2, )
import Data.List.HT (partition, )
import Data.Tuple.HT (mapSnd, )
import Data.Maybe.HT (toMaybe, )
import Data.Maybe (mapMaybe, )
import Data.List (nub, intersperse, (\\), )
```

The orchestra file consists of two parts: a *header*, and one or more *instrument blocks*. The header sets global parameters controlling sampling rate and control rate. The instrument blocks define instruments, each identified by a unique integer ID, and containing statements modifying or generating various audio signals. Each note statement in a score file passes all its arguments—including the p-fields—to its corresponding instrument in the orchestra file. While some properties vary from note to note, and should therefore be designed as p-fields, many can be defined within the instrument; the choice is up to the user.

The orchestra file is represented as:

```
data Output out =>
        T out = Cons Header [InstrBlock out] deriving (Show, Eq)
```

The orchestra header sets the audio rate, control rate, and number of output channels:

```
type Header = (AudRate, CtrlRate)

type AudRate  = Int  -- samples per second
type CtrlRate = Int  -- samples per second
```

Digital computers represent continuous analog audio waveforms as a sequence of discrete samples. The audio rate (`AudRate`) is the number of these samples calculated each second. Theoretically, the maximum frequency that can be represented is equal to one-half the audio rate. Audio CDs contain 44,100 samples per second of music, giving them a maximum sound frequency of 22,050 Hz, which is as high as most human ears are able to hear.

Computing 44,100 values each second can be a demanding task for a CPU, even by today's standards. However, some signals used as inputs to other signal generating routines don't require such a high resolution, and can thus be generated at a lower rate. A good example of this is an amplitude envelope, which changes relatively slowly, and thus can be generated at a rate much lower than the audio rate. This rate is called the *control rate* (`CtrlRate`), and is set in the orchestra file header. The audio rate is usually a multiple of the control rate, but this is not a requirement.

Each instrument block contains four things: a unique identifying integer; an expression giving the amount of extra time the instrument should be granted, usually used for reverb; an `Output` expression that gives the outputs in terms of *orchestra expressions*, called `SigExp`s; and a list of global signals and the `SigExp`s that are written out to those signals.

```
type Reverb = SigExp
data InstrBlock a =
       InstrBlock {instrBlockInstr   :: Instrument,
                   instrBlockReverb  :: Reverb,
                   instrBlockOutput  :: a,
                   instrBlockGlobals :: [(GlobalSig, SigExp)]}
     deriving (Show, Eq)
```

Recall that `Instrument` is a type synonym for an `Int`. This value may be obtained from a string name and a name map using the function `getId :: NameMap -> Name -> Maybe Int` discussed earlier.

Orchestra Expressions The data type `SigExp` is the largest deviation that we will make from the actual CSound design. In CSound, instruments are defined using a sequence of statements that, in a piecemeal manner, define the various oscillators, summers, constants, etc. that make up an instrument. These pieces can be given names, and these names can be referenced from other statements. But despite this rather imperative, statement-oriented approach, it is acually completely functional. In other words, every CSound instrument can be rewritten as a single expression. It is this "expression language" that we capture in `SigExp`. A pleasant attribute of the result is that CSound's ad hoc naming mechanism is replaced with Haskell's conventional way of naming things.

The entire `SigExp` data type declaration, as well as the declarations for related datatypes, is shown in Figure 11. In what follows, we describe each of the various constructors in turn.

Constants `ConstFloat x` represents the floating-point constant `x`.

P-field Arguments `pField n` refers to the nth p-field argument. Recall that all note characteristics, including pitch, volume, and duration, are passed into the orchestra file as p-fields. For example, to access the pitch, one would write `pField 4`. To make the access of these most common p-fields easier, we define the following constants:

```
noteDur, notePit, noteVel :: SigExp
noteDur = pField 3
notePit = pField 4
noteVel = pField 5

pField :: Int -> SigExp
pField n = TreeRec.Branch (PField n)
```

It is also useful to define the following standard names, which are identical to those used in CSound:

```
p1,p2,p3,p4,p5,p6,p7,p8,p9 :: SigExp
p1 = pField 1
p2 = pField 2
```

```
type Function = String
type OutCount = Integer
type Table    = Int

type Boolean = BooleanTerm SigExp
data BooleanTerm tree =
           Operator   Function (BooleanTerm tree) (BooleanTerm tree)
         | Comparison Function tree tree
     deriving (Show, Eq)

data GlobalSig =
   Global EvalRate (SigExp -> SigExp -> SigExp) Int
instance Show GlobalSig where
   show (Global rt _ n) = "Global " ++ show rt ++ " <function> " ++ show n
instance Eq GlobalSig where
   Global r1 _ n1 == Global r2 _ n2  =  r1 == r2 && n1 == n2

type DelayLine = DelayLineTerm SigExp
data DelayLineTerm tree = DelayLine tree tree
     deriving (Show, Eq)

data SigTerm tree =
              ConstFloat Float
            | ConstInt Int
            | TableNumber Table
            | PField Int
            | Str String
            | Read GlobalSig
            | Tap Function (DelayLineTerm tree) [tree]
            | Result (DelayLineTerm tree)
            | Conditional (BooleanTerm tree) tree tree
            | Infix  Function tree tree
            | Prefix Function tree
            | SigGen Function EvalRate OutCount [tree]
            | Index OutCount (SigTerm tree)
     deriving (Show, Eq)

instance Functor BooleanTerm where
   fmap f branch =
      case branch of
        Operator   nm left right -> Operator   nm (fmap f left) (fmap f right)
        Comparison nm left right -> Comparison nm (     f left) (     f right)

instance Functor DelayLineTerm where
   fmap f (DelayLine x y) = DelayLine (f x) (f y)

instance Functor SigTerm where
   fmap f branch =
     case branch of
        {- The first cases look like they could be handled
           by returning just 'branch'. But this does not work,
           because the result have a different type in general. -}
        ConstFloat x       -> ConstFloat x
        ConstInt n    -> ConstInt n
        TableNumber t -> TableNumber t
        PField n      -> PField n
        Str str       -> Str str
        Read t        -> Read t
        Tap nm del xs -> Tap nm (fmap f del) (map f xs)
        Result del    -> Result (fmap f del)
        Conditional b true false ->
           Conditional (fmap f b) (f true) (f false)
        Infix  nm left right -> Infix nm (f left) (f right)
        Prefix nm arg        -> Prefix nm (f arg)
        SigGen nm rate cnt args ->
           SigGen nm rate cnt (map f args)
        Index cnt x -> Index cnt (fmap f x)

instance TreeTag CollShow SigTerm where
```

```
p3 = pField 3
p4 = pField 4
p5 = pField 5
p6 = pField 6
p7 = pField 7
p8 = pField 8
p9 = pField 9
```

Strings `Str s` represents a string argument in CSound — a type of argument that is very rarely used, but is included here for the sake of completeness.

Reading and Writing Global Signals `Read g` is the counterpart to the `(GlobalSig, SigExp)` pairs in the `InstrBlock` statements, reading instead of writing global signals. Together, they allow for audio and control signals to be passed from instrument to instrument, and used for things like panning or overall envelopes.

Logical and Conditional Statements You probably noticed that `Boolean` was defined alongside `SigExp` in Figure **??**. `Boolean` is a type of expression used in the `Conditional SigExp` — basically, it's a comparison or some logical function of two comparisons. In other words, a `Boolean` is an expression that evaluates to a boolean. The syntax is fairly simple — a `Boolean` is either a `Comparison`, a function comparing two `SigExp`s and returning a `Boolean`; or an `Operator`, a function from two `Boolean`s to a third `Boolean`, such as the logical "and" operator. Thus we can express, for example, a query about whether a certain p-value lies within a range by evaluating this expression:

```
Operator "&&" (Comparison "<" 1 p2) (Comparison "<" p2 3)
```

The above expression will create a CSound expression that is true when p2 lies between 1 and 3.

`Boolean`s can be used inside of a `Conditional` expression in order to choose one of two values based on the trueness or falseness of the `Boolean`. For example:

```
Conditional (Comparison ">" p1 p2) p1 p2
```

will return the maximum of the two values p1 and p2. We are including several functions that will perform this automatically:

```
(<*), (<=*), (>*), (>=*), (==*), (/=*) ::
   -- SigExp -> SigExp -> Boolean
   TreeTerm term =>
      TreeRec.T term -> TreeRec.T term -> BooleanTerm (TreeRec.T term)
(<* ) = comparisonTerm "<"
(<=*) = comparisonTerm "<="
(>* ) = comparisonTerm ">"
(>=*) = comparisonTerm ">="
(==*) = comparisonTerm "=="
(/=*) = comparisonTerm "!="

(&&*), (||*) :: Boolean -> Boolean -> Boolean
```

```
(&&*) = operator   "&&"
(||*) = operator   "||"

operator :: String -> Boolean -> Boolean -> Boolean
operator = Operator
```

Arithmetic and Transcendental Functions Arithmetic functions are represented in various ways, depending on the type of function. The standard binary operators — plus and times, for instance — are infix operators, and so they can be crafted in this module using the Infix constructor, specifying the name of the function (the text used to express it in CSound) and the two arguments to the function. The other mathematical operators, such as `sin`, `log`, or `sqrt`, can be expressed with a `Prefix` constructor, passing the name of the function in CSound (usually the same as the name in Haskell, although not always) and the argument to the given function. Examples of this are:

```
Infix "+" (PField 1) (Prefix "sin" 1 (ConstFloat 3.0))
Prefix "sqrt" (Infix "*" (PField 3) (PField 4))
```

To facilitate the use of these arithmetic functions, we can make `SigExp` an instance of certain numeric type classes, thus providing more conventional names for the various operations.

```
sigGen :: Function -> EvalRate -> OutCount -> [SigExp] -> SigExp
sigGen nm rate cnt args = TreeRec.Branch (SigGen nm rate cnt args)

constFloat :: Float -> SigExp
constFloat = TreeRec.Branch . ConstFloat

constInt :: Int -> SigExp
constInt = TreeRec.Branch . ConstInt

constEnum :: Enum a => a -> SigExp
constEnum = TreeRec.Branch . ConstInt . fromEnum

class TreeTerm term where
  constTerm  :: Float    -> TreeRec.T term
  prefixTerm :: Function -> TreeRec.T term -> TreeRec.T term
  infixTerm  :: Function -> TreeRec.T term -> TreeRec.T term -> TreeRec.T term
  comparisonTerm  :: Function -> TreeRec.T term -> TreeRec.T term ->
                                                   BooleanTerm (TreeRec.T term)
  ifthen :: BooleanTerm (TreeRec.T term) ->
                          TreeRec.T term -> TreeRec.T term -> TreeRec.T term

instance TreeTerm SigTerm where
  constTerm          x   = TreeRec.Branch (ConstFloat          x)
  prefixTerm      nm x   = TreeRec.Branch (Prefix      nm x)
  infixTerm       nm x y = TreeRec.Branch (Infix       nm x y)
  comparisonTerm  nm x y = Comparison nm x y
  ifthen          b  x y = TreeRec.Branch (Conditional b  x y)
```

We can not request `term == SigTerm TreeRec.T` that's why we have to define the `TreeTerm` class and the instance for `SigTerm`.

```
instance (TreeTag.CollShow term, TreeTag.CollEq term,
          Functor term, TreeTerm term) =>
         Num (TreeRec.T term) where
  (+)      = infixTerm "+"
  (-)      = infixTerm "-"
  (*)      = infixTerm "*"
  negate   = prefixTerm "-"
  abs      = prefixTerm "abs"
  signum x = ifthen (x <* 0) (-1) (ifthen (x >* 0) 1 0)
  fromInteger = constTerm . fromInteger

instance (TreeTag.CollShow term, TreeTag.CollEq term,
          Functor term, TreeTerm term) =>
         Fractional (TreeRec.T term) where
  (/) = infixTerm "/"
  fromRational = constTerm . fromRational
{-
  fromRational x =
     fromInteger (numerator x) /
     fromInteger (denominator x)
-}

instance (TreeTag.CollShow term, TreeTag.CollEq term,
          Functor term, TreeTerm term) =>
         Floating (TreeRec.T term) where
  exp     = prefixTerm "exp"
  log     = prefixTerm "log"
  sqrt    = prefixTerm "sqrt"
  (**)    = infixTerm "^"
  pi      = constTerm pi
  sin     = prefixTerm "sin"
  cos     = prefixTerm "cos"
  tan     = prefixTerm "tan"
  asin    = prefixTerm "sininv"
  acos    = prefixTerm "cosinv"
  atan    = prefixTerm "taninv"
  sinh    = prefixTerm "sinh"
  cosh    = prefixTerm "cosh"
  tanh    = prefixTerm "tanh"
  asinh x = log (sqrt (x*x+1) + x)
  acosh x = log (sqrt (x*x-1) + x)
  atanh x = (log (1+x) - log (1-x)) / 2
```

Now we can write simpler code, such as: `noteDur + sin p6 ** 2`.

Other `Prefix`s `sin`, `log`, and `sqrt` aren't the only functions that use `Prefix` as a constructor — `Prefix` is used for all functions in CSound that take a single argument and are represented like normal mathematical functions. Most of these functions are, indeed, mathematical, such as the function converting a CSound pitch value to the number of cycles per second, or the function converting decibels to the corresponding amplitude.

For convenience, we will define a few common operators here:

```
> pchToHz, dbToAmp :: SigExp -> SigExp
> pchToHz = prefixTerm "cpspch"
> dbToAmp = prefixTerm "ampdb"
```

Now, when we want to convert a pitch to its hertz value or a decibel level to the desired amplitude, we can simply say `pchToHz notePit` or `dbToAmp noteVel`.

Signal Generation and Modification The most sophisticated `SigExp` constructor is `sigGen`, which drives most of the functions used for signal generation and modification. The constructor takes four arguments: the name of the function to be used, such as *`envlpx`* or *`oscili`*; the rate of output; the number of outputs (covered in a later section); and a list of all the arguments to be passed.

Most of these we have seen before. But what is the rate of output? Well, signals in CSound can be generated at three rates: the note rate (i.e., with, every note event), the control rate, and the audio rate (we discussed the latter two earlier). Many of the signal generating routines can produce signals at more than one rate, so the rate must be specified as an argument. The following simple data structure serves this purpose:

```
data EvalRate = NR  -- note rate
              | CR  -- control rate
              | AR  -- audio rate
    deriving (Show, Eq, Ord)
```

All right, so now we know what the arguments are. But what does the `sigGen` constructor actually do? Like the other kinds of `SigExp`s, it has an input and an output. In Haskore, it acts just the same as any other kind of function. But when written to a CSound Orchestra file, each `sigGen` receives a variable name that it is assigned to, and each `sigGen` is written to a single line of the CSound file.

`sigGen`s can be used for all sorts of things — CSound has a very large variety of functions, most of which are actually `sigGen`s. They can do anything from generating a simple sine wave to generating complex signals. Most of them, however, have to do with signal generation; hence the name `sigGen`. For the user's sake, we will outline a few of the CSound functions here:

1. The CSound statement `line evalrate start duration finish`, produces values along a straight line from `start` to `finish`. The values can be generated either at control or audio rate, and the line covers a period of time equal to `duration` seconds. We can translate this into CSound like so:

   ```
   line, expon :: EvalRate -> SigExp -> SigExp -> SigExp -> SigExp
   line rate start duration finish =
      sigGen "line" rate 1 [start, duration, finish]
   ```

2. *`expon`* is similar to *`line`*, but the code `expon evalrate start duration finish` produces an exponential curve instead of a straight line.

   ```
   expon rate start duration finish =
      sigGen "expon" rate 1 [start, duration, finish]
   ```

3. If a more elaborate signal is required, one can use the CSound functions *linseg* or *expseg*, which take any odd number of arguments greater than or equal to three. The first three arguments work as before, but only for the first of a number of segments. The subsequent segment lengths and endpoints are given in the rest of the arguments. A signal containing both straight line and exponential segments can be obtained by adding a *linseg* signal and *expseg* signal together in an appropriate way.

   The Haskore code is more complicated for this, because there are an arbitrary but odd number of arguments. So we will give the first three arguments as we did with the *line* and *expon* functions, and then have a list of pairs, which will be flattened into an argument list:

```
lineSeg, exponSeg :: EvalRate -> SigExp -> SigExp -> SigExp
                     -> [(SigExp, SigExp)] -> SigExp
lineSeg rate y0 x1 y1 lst =
   sigGen "linseg" rate 1 ([y0, x1, y1] ++ flattenTuples2 lst)
exponSeg rate y0 x1 y1 lst =
   sigGen "expseg" rate 1 ([y0, x1, y1] ++ flattenTuples2 lst)
```

4. The Haskore code `env rate rshape sattn dattn steep dtime rtime durn sig` modifies the signal `sig` by applying an envelope to it.[7] `rtime` and `dtime` are the rise time and decay time, respectively (in seconds), and `durn` is the overall duration. `rshape` is the identifier integer of a function table storing the rise shape. `sattn` is the pseudo-steady state attenuation factor. A value between 0 and 1 will cause the signal to exponentially decay over the steady period, a value greater than 1 will cause the signal to exponentially rise, and a value of 1 is a true steady state maintained at the last rise value. `steep`, whose value is usually between -0.9 and $+0.9$, influences the steepness of the exponential trajectory. `dattn` is the attenuation factor by which the closing steady state value is reduced exponentially over the decay period, with value usually around 0.01.

   In Haskore, this becomes a fairly simple function, going from an `EvalRate` and eight `SigExp`s to one single `SigExp`:

```
env :: EvalRate -> SigExp -> SigExp -> SigExp -> SigExp -> SigExp
        -> SigExp -> SigExp -> SigExp -> SigExp
env rate rshape sattn dattn steep dtime rtime durn sig =
   sigGen "envlpx" rate 1
      [sig, rtime, durn, dtime, rshape, sattn, dattn, steep]
```

5. Typing `phasor phase freq` into CSound generates a signal moving from 0 to 1 at a given frequency and starting at the given initial phase offset. When used properly as the index to a table lookup unit, the function can simulate the behavior of an oscillator. We implement it in Haskore thus:

```
phasor :: EvalRate -> SigExp -> SigExp -> SigExp
phasor rate phase freq = sigGen "phasor" rate 1 [freq, phase]
```

6. CSound table lookup functions *table* and *tablei* both take `index`, `table`, and `indexmode` arguments. The `indexmode` is either 0 or 1, differentiating between raw index and normalized index (zero to one); for convenience we define:

[7] Although this function is widely-used in CSound, the same effect can be accomplished by creating a signal that is a combination of straight line and exponential curve segments, and multiplying it by the signal to be modified.

```
data IndexMode =
     RawIndex
   | NormalIndex
       deriving (Show, Eq, Enum)
```

Both *table* and *tablei* return values stored in the specified table at the given index. The difference is that *tablei* uses the fractional part of the index to interpolate between adjacent table entries, which generates a smoother signal at a small cost in execution time. The equivalent Haskore code to the CSound functions is:

```
tblLookup, tblLookupI ::
   EvalRate -> IndexMode -> SigExp -> SigExp -> SigExp
tblLookup  rate mode table ix =
   sigGen "table"  rate 1 [ix, table, constEnum mode]
tblLookupI rate mode table ix =
   sigGen "tablei" rate 1 [ix, table, constEnum mode]
```

As mentioned, the output of *phasor* can be used as input to a table lookup to simulate an oscillator whose frequency is controlled by the note pitch. This can be accomplished easily by the following piece of Haskore code:

```
oscil = let index = phasor AR (pchToHz notePit) 0.0
        in  tblLookupI AR NormalIndex table index
```

where `table` is some given function table ID. If `oscil` is given as argument to an output constructor such as `MonoOut`, then this `Output` coupled with an instrument ID number (say, 1) produces a complete instrument block:

```
i1 = (1, MonoOut oscil)
```

Adding a suitable `Header` would then give us a complete, though somewhat sparse, `CSound.Orchestra.T` value.

7. Instead of the above design we could use one of the built-in CSound oscillators, *oscil* and *oscili*, which differ in the same way as *table* and *tablei*. Both CSound functions take the following arguments: raw amplitude, frequency, and the index of a table. The result is a signal that oscillates through the function table at the given frequency. Let the Haskore functions be as follows:

```
osc, oscI :: EvalRate -> SigExp -> SigExp -> SigExp -> SigExp
osc  rate table amp freq = sigGen "oscil"  rate 1 [amp, freq, table]
oscI rate table amp freq = sigGen "oscili" rate 1 [amp, freq, table]
```

Now, the following statement is equivalent to `osc`, defined above:

```
```

8. It is often desirable to use the output of one oscillator to modulate the frequency of another, a process known as *frequency modulation*. The Haskore code `fmOsc table modindex carfreq modfreq amp freq` produces a signal whose effective modulating frequency is `freq*modfreq`, and whose carrier frequency is `freq*carfreq`. `modindex` is the *index of modulation*, usually a

value between 0 and 4, which determines the timbre of the resulting signal. *oscili* behaves similarly to *oscil*, except that it, like *tablei* and *oscili*, interpolates between values.

Interestingly enough, these two functions are the first listed here that work at audio rate only; thus, we do not have to pass the rate as an argument to the helper function, because the rate is always AR. Thus, the Haskore code is:

```
fmOsc, fmOscI :: SigExp -> SigExp -> SigExp -> SigExp -> SigExp
                 -> SigExp -> SigExp
fmOsc  table modindex carfreq modfreq amp freq =
   sigGen "foscil"  AR 1 [amp, freq, carfreq, modfreq, modindex, table]
fmOscI table modindex carfreq modfreq amp freq =
   sigGen "foscili" AR 1 [amp, freq, carfreq, modfreq, modindex, table]
```

9. `sampOsc table amp freq` oscillates through a table containing an AIFF sampled sound segment. This is the only time a table can have a length that is not a power of two, as mentioned earlier. Like `fmOsc`, `sampOsc` can only generate values at the audio rate:

```
sampOsc :: SigExp -> SigExp -> SigExp -> SigExp
sampOsc table amp freq = sigGen "loscil" AR 1 [amp, freq, table]
```

10. The Haskore code `random rate amp` produces a random number series between `-amp` and `+amp` at either control or audio rate. `randomH rate quantRate amp` does the same but will hold each number for `quantRate` cycles before generating a new one. `randomI rate quantRate amp` will in addition provide straight line interpolation between successive numbers:

```
random :: EvalRate -> SigExp -> SigExp
random rate amp = sigGen "rand" rate 1 [amp]

randomH, randomI :: EvalRate -> SigExp -> SigExp -> SigExp
randomH rate quantRate amp = sigGen "randh" rate 1 [amp, quantRate]
randomI rate quantRate amp = sigGen "randi" rate 1 [amp, quantRate]
```

The remaining functions covered in this file only operate at audio rate, and thus their Haskore equivalents do not have `rate` arguments.

11. `genBuzz table multiplier loharm numharms amp freq` generates a signal that is an additive set of harmonically related cosine partials. `freq` is the fundamental frequency, `numharms` is the number of harmonics, and `loharm` is the lowest harmonic present. The amplitude coefficients of the harmonics are given by the exponential series `a`, `a * multiplier`, `a * multiplier^2`, ..., `a * multiplier^(numharms-1)`. The value `a` is chosen so that the sum of the amplitudes is `amp`. `table` is a function table containing a cosine wave.

```
genBuzz :: SigExp -> SigExp -> SigExp -> SigExp -> SigExp
           -> SigExp -> SigExp
genBuzz table multiplier loharm numharms amp freq =
   sigGen "gbuzz" AR 1 [amp, freq, numharms, loharm, multiplier, table]
```

12. `buzz` is a special case of `genBuzz` in which `loharm = 1.0` and `multiplier = 1.0`. `table` is a function table containing a sine wave:

```
buzz :: SigExp -> SigExp -> SigExp -> SigExp -> SigExp
buzz table numharms amp freq =
   sigGen "buzz" AR 1 [amp, freq, numharms, table]
```

Note that the above two constructors have an analog in the generating routine GEN11 and the related function `cosineHarms` (see Section 4.3.1). `cosineHarms` stores into a table the same waveform that would be generated by `buzz` or `genBuzz`. However, although `cosineHarms` is more efficient, it has fixed arguments and thus lacks the flexibility of `buzz` and `genBuzz` in being able to vary the argument values with time.

13. `pluck table freq2 decayMethod amp freq` is an audio signal that simulates a plucked string or drum sound, constructed using the Karplus-Strong algorithm. The signal has amplitude `amp` and frequency `freq2`. It is produced by iterating through an internal buffer that initially contains a copy of `table` and is smoothed with frequency `freq` to simulate the natural decay of a plucked string. If 0.0 is used for `table`, then the initial buffer is filled with a random sequence. There are six possible decay modes:

    (a) *simple smoothing*, which ignores the two arguments;

    (b) *stretched smoothing*, which stretches the smoothing time by a factor of `decarg1`, ignoring `decarg2`;

    (c) *simple drum*, where `decarg1` is a "roughness factor" (0 for pitch, 1 for white noise; a value of 0.5 gives an optimal snare drum sound);

    (d) *stretched drum*, which contains both roughness (`decarg1`) and stretch (`decarg2`) factors;

    (e) *weighted smoothing*, in which `decarg1` gives the weight of the current sample and `decarg2` the weight of the previous one (`decarg1+decarg2` must be ≤ 1); and

    (f) *recursive filter smoothing*, which ignores both arguments.

Here again are some helpful constants:

```
data PluckDecayMethod =
     PluckSimpleSmooth
   | PluckStretchSmooth SigExp
   | PluckSimpleDrum SigExp
   | PluckStretchDrum SigExp SigExp
   | PluckWeightedSmooth SigExp SigExp
   | PluckFilterSmooth
```

And here is the Haskore code for the CSound pluck function:

```
pluck :: SigExp -> SigExp -> PluckDecayMethod
          -> SigExp -> SigExp -> SigExp
pluck table freq2 decayMethod amp freq =
   sigGen "pluck" AR 1
          ([amp, freq, freq2, table] ++
           case decayMethod of
             PluckSimpleSmooth ->
                [constInt 1]
             PluckStretchSmooth stretch ->
                [constInt 2, stretch]
```

```
            PluckSimpleDrum roughness ->
               [constInt 3, roughness]
            PluckStretchDrum roughness stretch ->
               [constInt 4, roughness, stretch]
            PluckWeightedSmooth weightCur weightPrev ->
               [constInt 5, weightCur, weightPrev]
            PluckFilterSmooth ->
               [constInt 6])
```

14. `delay delayTime sig` takes a signal `sig` and delays it by `delayTime` — basically making it start `delayTime` later than it normally would have. This is a simple version of delay lines and delay taps, capable of performing all of the effects that don't involve feeding the result of a delay or a tap back into the input. This topic is more complicated and will be considered in the next section. In constrast to `delay`, the function `vdelay` also allows for a controlled delay. But for memory allocation reasons it must also know the maximum possible delay (in seconds).

```
delay :: SigExp -> SigExp -> SigExp
delay delayTime sig = sigGen "delay" AR 1 [sig, delayTime]

vdelay :: SigExp -> SigExp -> SigExp -> SigExp
vdelay maxDelay delayTime sig =
   sigGen "vdelay" AR 1 [sig, delayTime, maxDelay*1000]
```

15. Reverberation can be added to a signal using the CSound functions `comb looptime revtime sig`, `alpass looptime revtime sig`, and `reverb revtime sig`. `revtime` is the time in seconds it takes a signal to decay to 1/1000th of its original amplitude, and `looptime` is the echo density. `comb` produces a "colored" reverb, `alpass` a "flat" reverb, and `reverb` a "natural room" reverb:

```
comb :: SigExp -> SigExp -> SigExp -> SigExp
comb looptime revtime sig =
   sigGen "comb" AR 1 [sig, revtime, looptime]

alpass :: SigExp -> SigExp -> SigExp -> SigExp
alpass looptime revtime sig =
   sigGen "alpass" AR 1 [sig, revtime, looptime]

reverb :: SigExp -> SigExp -> SigExp
reverb revtime sig =
   sigGen "reverb" AR 1 [sig, revtime]
```

Delay Lines and Tapping `DelayLine deltime audiosig` establishes a digital delay line, where `audiosig` is the source, and `deltime` is the delay time in seconds. That `DelayLine` can either be simply read, by the `Result delayline` constructor, or tapped, by the `Tap tapname delayline args` constructor.

The most common tap functions are *`deltap`* and *`deltapi`*, where *`deltapi`* is the interpolating version of *`deltap`*. Thus we will include helper functions for both of those functions:

```
delTap, delTapI :: DelayLine -> SigExp -> SigExp
delTap  dl tap = TreeRec.Branch (Tap "deltap"  dl [tap])
delTapI dl tap = TreeRec.Branch (Tap "deltapi" dl [tap])
```

Recursive Statements In some cases, the user may want their instrument to have certain special effects — such as an infinite echo, going back and forth but getting fainter and fainter. It would seem logical that the user would, in that case, write something like this:

```
x = sig + delay (0.5 * x) 1.0
```

Unfortunately, the translation process cannot handle statements like that, and any kind of statement which is defined in terms of itself must be written a different way. *Within* Haskore, recursive statements are handled using three constructors: `Loop`, `Var`, and `Rec`. However, these three constructors are not available to the users, and so we offer a very simple solution: the `rec` function:

```
rec :: (SigExp -> SigExp) -> SigExp
rec = TreeRec.recourse
```

In order to perform the infinite echo listed above, we would write this code:

```
x = rec (\y -> sig + delay (0.5 * y) 1.0)
```

Thus `rec`, in some ways, is a bit like `fix`, although it doesn't actually do the computation — instead, it juggles some code around and passes the problem off to CSound.

When the `SigExp` is processed, all `Rec` constructors are converted into a `SigExp` with `Loop` and `Var` constructors. Each `Loop` has some number of matching `Var` statements, with the same unique integer referring to both. This is done through the `runFix` function and its various helper functions:

```
type SigFixed = TreeTag.T TreeRec.Tag SigTerm

runFix, simpleFix :: SigExp -> SigFixed
runFix = addEqTree . TreeRec.toTaggedUnique 1
{- some expressions need no loop unwinding,
   toTagged does unwinding anyway, but with less overhead
   and shared loop ids -}
simpleFix = TreeRec.toTagged 0

instance Foldable BooleanTerm where
   foldMap = Traversable.foldMapDefault

instance Traversable BooleanTerm where
   sequenceA branch =
      case branch of
        Operator   nm left right ->
           liftA2 (Operator   nm) (sequenceA left) (sequenceA right)
        Comparison nm left right ->
           liftA2 (Comparison nm) (          left) (          right)

instance Foldable DelayLineTerm where
   foldMap = Traversable.foldMapDefault
```

```
instance Traversable DelayLineTerm where
   sequenceA (DelayLine x y) = liftA2 DelayLine x y

instance Foldable SigTerm where
   foldMap = Traversable.foldMapDefault

instance Traversable SigTerm where
   sequenceA branch =
     case branch of
        {- compare with Functor instance -}
        ConstFloat x  -> pure $ ConstFloat x
        ConstInt n    -> pure $ ConstInt n
        TableNumber t -> pure $ TableNumber t
        PField n      -> pure $ PField n
        Str str       -> pure $ Str str
        Read t        -> pure $ Read t
        Tap nm del xs -> liftA2 (Tap nm) (sequenceA del) (sequenceA xs)
        Result del    -> liftA  Result   (sequenceA del)
        Conditional b true false ->
           liftA3 Conditional (sequenceA b) true false
        Infix  nm left right    -> liftA2 (Infix nm)  left right
        Prefix nm arg           -> liftA  (Prefix nm) arg
        SigGen nm rate cnt args ->
           liftA (SigGen nm rate cnt) (sequenceA args)
        Index cnt x -> liftA (Index cnt) (sequenceA x)

-- fixSig (Rec (LoopFunction f)) =
--    do n <- get; put (n + 1); fixSig (Loop n (addEq (f (Var n))))

addEqTree :: SigFixed -> SigFixed
addEqTree (TreeTag.Branch x)  = TreeTag.Branch (fmap addEqTree x)
addEqTree (TreeTag.Tag t x) = TreeTag.Tag t (addEqTree (addEq x))
addEqTree (TreeTag.Loop t)  = TreeTag.Loop t

addEq :: SigFixed -> SigFixed
addEq ex =
   case ex of
      TreeTag.Branch (SigGen _ _ _ _) -> ex
      TreeTag.Branch (Tap _ _ _)      -> ex
      TreeTag.Branch (Result _)       -> ex
      _ -> TreeTag.Branch (SigGen "="
              (if CR == getRate ex
                 then CR else AR) 1 [ex])

getRate :: SigFixed -> EvalRate
getRate (TreeTag.Branch branch) = getRateTerm branch
getRate (TreeTag.Tag _ arg) = getRate arg
getRate (TreeTag.Loop _) = error "getRate: undefined rate"

getRateTerm :: SigTerm SigFixed -> EvalRate
getRateTerm branch =
   case branch of
      Tap _ _ _         -> AR
      Result _          -> AR
```

```
      Conditional _ a b -> max (getRate a) (getRate b)
      Infix _ a b       -> max (getRate a) (getRate b)
      Prefix _ arg      -> getRate arg
      SigGen _ rt _ _   -> rt
      Index _ arg       -> getRateTerm arg
      _                 -> NR
```

Note that the `addEq` function is used to add an equal sign to the statement being looped, provided that the statement is not already one of the signal generating ones. Also note that if the rate of the statement is `NR`, the new rate will be `AR` — this is because you cannot have an infinitely recursive statement at the note rate.

Ideally, all `SigExp` statements should have `runFix` applied to them. So we have the `getFixedExpressions` function, used as a replacement to the standard `getChannels` of the `Output` class:

```
getFixedExpressions :: Output a => a -> [SigFixed]
getFixedExpressions = map (aux . runFix) . getChannels
   where aux ex =
            if AR == getRate ex
              then ex
              else TreeTag.Branch (SigGen "=" AR 1 [ex])
```

Signal Generators with Multiple Outputs When looking through the CSound documentation, you may notice that there are certain functions, such as *`convolve`* or *`babo`* that do not have the same structure in CSound as the most of the rest of the functions. This is because those are two operators that actually return multiple outputs. While this type of function is not extremely common, we have included code that can, in fact, handle such functions. The third argument to the `sigGen` constructor actually specifies the number of arguments to be returned. In most cases, this should simply be set to one; in a few cases, such as *`convolve`* or *`babo`*, this should be set to however many outputs you want returned from the function.

But how do you get to those outputs? Well, the `Index` constructor is used from within the code, but the user cannot access that. So we have the following function:

```
getMultipleOutputs :: SigExp -> [SigExp]
getMultipleOutputs (TreeRec.Branch ex@(SigGen _ _ outCount _)) =
   if outCount==1
     then error ("cannot get multiple outputs from a function with one output")
     else map (TreeRec.Branch . flip Index ex) [1..outCount]
getMultipleOutputs _ =
   error ("cannot get multiple outputs from a non-SigGen")
```

Which can be called on any `sigGen` statement returning multiple arguments, and returns a list of the outputs. In other words, you could write something like this:

```
[a1, a2] = getMultipleOutputs
             (LineStatement "babo" AR 2 [sig, 0, 0, 0, 5, 5, 5])
```

Haskell would then pattern-match, and leave you with two variables, `a1` and `a2`.

Output Operators Now that we've got all of those interesting methods of signal generation under our belts, we need some way to make CSound play these interesting sound waves. Hence, the *output statements*, all of which must be instances of the `Output` class:

```
class (Show c, Eq c) => Output c where
   getChannels :: c -> [SigExp]
   getName :: c -> String
   getChannelCount :: c -> Int
```

The `getChannelCount` could be pre-defined with `length . getChannels` but this would require that we have actually an `Output` value at hand when calling `getChannelCount`.

We have defined several common types of output, including `Mono`, which allows for the writing of one output channel; `Stereo`, which allows for two; and `Quad`, which, unsurprisingly, allows four:

```
data Mono   = Mono   SigExp deriving (Show, Eq)
data Stereo = Stereo SigExp SigExp deriving (Show, Eq)
data Quad   = Quad   SigExp SigExp SigExp SigExp deriving (Show, Eq)

instance Output Mono where
   getChannels (Mono x) = [x]
   getName _ = "out"
   getChannelCount _ = 1

instance Output Stereo where
   getChannels (Stereo x1 x2) = [x1, x2]
   getName _ = "outs"
   getChannelCount _ = 2

instance Output Quad where
   getChannels (Quad x1 x2 x3 x4) = [x1, x2, x3, x4]
   getName _ = "outq"
   getChannelCount _ = 4
```

The user is welcome to add more by declaring them instances of the `Output` class and then filling out the required methods.

Converting Orchestra Values to Orchestra Files We must now convert the `SigExp` values into a form which can be written into a CSound `.sco` file. As mentioned earlier, each signal generation or modification statement in CSound assigns its result a string name. This name is used whenever another statement takes the signal as an argument. Names of signals generated at note rate must begin with the letter `i`, control rate with letter `k`, and audio rate with letter `a`. The output statements do not generate a signal so they do not have a result name.

The function `mkList` is shown in Figure 12, and generates a list containing every single sub-expression of the given `SigExp`. It uses the following auxiliary functions:

```
type DelayLineFixed = DelayLineTerm SigFixed
type BooleanFixed   = BooleanTerm SigFixed

mkListAll :: [SigFixed] -> [SigFixed]
mkListAll = concatMap mkList
```

```
mkList :: SigFixed -> [SigFixed]
mkList ex@(TreeTag.Branch n)  = ex : mkListTerm n
mkList ex@(TreeTag.Tag _ x) = ex : mkList x
mkList    (TreeTag.Loop _)  = []

mkListTerm :: SigTerm SigFixed -> [SigFixed]
mkListTerm term =
   case term of
     Tap _ dl lst      -> mkListDL dl ++ mkListAll lst
     Result dl         -> mkListDL dl
     Conditional a b c -> mkListBool a ++ mkListAll [b, c]
     Infix _ a b       -> mkListAll [a, b]
     Prefix _ x        -> mkList x
     SigGen _ _ outCount lst ->
        if outCount == 1
          then mkListAll lst
          else map (TreeTag.Branch . flip Index term) [1..outCount]
                 ++ mkListAll lst
               -- cf. getMultipleOutputs
     Index _ expr      -> mkListTerm expr
     _                 -> []
```

Figure 12: The `mkList` Function

```
mkListDL :: DelayLineFixed -> [SigFixed]
mkListDL (DelayLine x1 x2) = mkListAll [x1, x2]

mkListBool :: BooleanFixed -> [SigFixed]
mkListBool (Operator   _ a b) = concatMap mkListBool [a, b]
mkListBool (Comparison _ a b) = mkListAll [a, b]

mkListOut :: Output a => InstrBlock a -> [SigFixed]
mkListOut (InstrBlock _ xtim chnls lst) =
   mkListAll (simpleFix xtim : getFixedExpressions chnls ++
                map (simpleFix . snd) lst)
              -- there should not be any loop to be unwind in lst
```

Once we have the list of all of the expressions, we need to find the signal-generating ones, like `Taps` and `sigGens`, and convert them into a list of `StatementDefs`, with their associated rates. This is done using the function `getLineRates`.

```
type LineFunctionRates = [(EvalRate, StatementDef)]

data StatementDef = StatementDef Function [SigFixed]
                  | TapDef Function DelayLineFixed [SigFixed]
                  | DelayDef DelayLineFixed
                  | DelayWriteDef DelayLineFixed
                  | MultiDef Function [SigFixed]
                             OutCount (SigTerm SigFixed)
                  | IndexDef OutCount (SigTerm SigFixed)
     deriving (Show, Eq)

getLineRates :: [SigFixed] -> LineFunctionRates
getLineRates = mapMaybe aux
   where
      aux (TreeTag.Branch n) =
         case n of
           Tap nm dl lst       -> Just (AR, TapDef nm dl lst)
```

```
          Result dl           -> Just (AR, DelayDef dl)
          SigGen nm rt ct lst -> Just (rt,
                                   if ct==1
                                     then StatementDef nm lst
                                     else MultiDef nm lst ct n)
          Index ct ex@(SigGen _ rt _ _) ->
                                  Just (rt, IndexDef ct ex)
          _                   -> Nothing
      aux _ = Nothing
```

DelayLines and Taps are a rather complex problem in Haskore. In CSound, there is no such thing as an explicit delay line; you establish a delay line with a *delayr* opcode, and then all taps that occur between that line and the matching *delayw* line belong to that particular delay line. Thus the translation from the Haskore concept of delay lines to the CSound concept is somewhat difficult. Hence procDelay and its various helper functions, which gather all of the taps together and add the requisite DelayWriteDef to the end of them:

```
procDelay :: LineFunctionRates -> LineFunctionRates
procDelay lst@((_, DelayDef dl) : _)   = setUpDelays lst dl
procDelay lst@((_, TapDef _ dl _) : _) = setUpDelays lst dl
procDelay (hd : tl)                    = hd : procDelay tl
procDelay []                           = []

setUpDelays :: LineFunctionRates -> DelayLineFixed -> LineFunctionRates
setUpDelays lst dl =
   let aux (_, DelayDef dl2)   = dl == dl2
       aux (_, TapDef _ dl2 _) = dl == dl2
       aux _                   = False
       (dels, rest) = partition aux lst
    in procTaps dels dl ++ procDelay rest

procTaps :: LineFunctionRates -> DelayLineFixed -> LineFunctionRates
procTaps lst dl =
   [(AR, DelayDef dl)] ++ filter aux lst ++ [(AR, DelayWriteDef dl)]
   where aux (_, TapDef _ _ _) = True
         aux _                 = False
```

Putting all of the above together, here is a function that converts an SigExp into a list of proper name / StatementDef pairs. Each one of these will eventually result in one statement in the CSound orchestra file. (The result of getLineRates is reversed to ensure that a definition exists before it is used; and this must be done *before* nub is applied (which removes duplicates), for the same reason.)

```
type StatementDefs = [(Name, StatementDef)]

extractFunctions :: [SigFixed] -> StatementDefs
extractFunctions =
   zipWith giveName [1 ..] . nub . procDelay . reverse . getLineRates

giveName :: Int -> (EvalRate, StatementDef) -> (Name, StatementDef)
giveName n (er,x) =
   let var = case er of
               AR -> 'a'
               CR -> 'k'
```

```
                NR -> 'i'
   in (var : show n, x)
```

The functions that follow are used to write the orchestra file. `saveIA` is similar to `Score.saveIA`: it asks the user for a file name, opens the file, writes the given orchestra value to the file, and then closes the file.

```
saveIA :: Output a => T a -> IO ()
saveIA orch =
   do putStr "\nName your orchestra file "
      putStr "(.orc extension will be added): "
      name <- getLine
      save name orch

save :: Output a => FilePath -> T a -> IO ()
save name orch =
   writeFile (name ++ ".orc") (toString orch)
```

`CSound.Orchestra.toString` splits the task of writing the orchestra into two parts: writing the header, and writing the instrument blocks.

```
toString :: Output a => T a -> String
toString orc@(Cons hdr ibs) =
   let glob = getGlobal ibs
   in  unlines $
       headerToString hdr (channelCount orc) ++
       maybe [] writeGlobalHeader glob ++
       concatMap instrBlockToString ibs ++
       maybe [] resetGlobals glob
```

Writing the header is relatively simple, and is accomplished by the following function:

```
headerToString :: Header -> Int -> [String]
headerToString (a,k) nc =
  ["sr     = " ++ show a,
   "kr     = " ++ show k,
   "ksmps  = " ++ show (fromIntegral a / fromIntegral k :: Double),
   "nchnls = " ++ show nc]

channelCount :: Output a => T a -> Int
channelCount (Cons _ instrBlock) =
   getChannelCount (instrBlockOutput (head instrBlock))
```

If the instance of `getChannelCount` does not rely on `getChannels` the `instrBlock` can be empty.

`instrBlockToString` writes a single instrument block.

```
instrBlockToString :: Output a => InstrBlock a -> [String]
instrBlockToString ib@(InstrBlock num xtim _ _) =
   let ses = mkListOut ib
       noes = extractFunctions ses
       lps = getLoops noes ses
    in "" :
```

```
       showInstrument num :
       writeLoops lps ++
       concatMap (writeExp noes lps) noes ++
       writeOut noes lps ib ++
       (if xtim /= 0
         then ["xtratim " ++ showExp noes lps (simpleFix xtim)]
         else []) ++
       "endin" :
       []

showInstrument :: Instrument ->  String
showInstrument instr = "instr " ++ showInstrumentNumber instr
```

`Loop` statements require special handling, including initialization at the top of each instrument and a special set of loop definitions which are also passed to most of the writing functions. This is handled by the following two functions:

```
type LoopDefs = [(TreeRec.Tag, String)]

writeLoops :: LoopDefs -> [String]
writeLoops = map ((++ " init 0") . snd)

getLoops :: StatementDefs -> [SigFixed] -> LoopDefs
getLoops noes =
   let extractTag (TreeTag.Tag n ex) = Just (n, ex)
       extractTag _                  = Nothing
   in  map (mapSnd (showExp noes []))
           . nub . mapMaybe extractTag
     -- map and mapMaybe are separated for efficiency achieved by nub
```

Globals, too, require special handling: they need both a header at the top of the CSound orchestra file, and an instrument in which to reset their values. Those requirements are fulfilled by the following functions, which are called from the `instrBlockToString` function.

```
globalRate :: EvalRate -> String
globalRate AR = "a"
globalRate CR = "k"
globalRate NR = error ("you cannot use init-rate globals")

globalWrite, globalRead :: GlobalSig -> String
globalWrite (Global rate _ n) = "g" ++ globalRate rate ++ "w" ++ show n
globalRead  (Global rate _ n) = "g" ++ globalRate rate ++ "r" ++ show n

resetGlobals :: ([GlobalSig], Instrument) -> [String]
resetGlobals (gs,num) =
   let aux g =
          (globalRead g ++ " = " ++ globalWrite g) :
          (globalWrite g ++ " = 0") :
          []
   in  "" :
       showInstrument num :
       concatMap aux gs ++
       "endin" :
       []
```

```
numGlobalInstrs :: Output a => [InstrBlock a] -> Instrument
numGlobalInstrs lst =
   head (instruments \\ map instrBlockInstr lst)

getGlobals :: Output a => [InstrBlock a] -> [GlobalSig]
getGlobals = concatMap (map fst . instrBlockGlobals)

getGlobal :: Output a => [InstrBlock a] -> Maybe ([GlobalSig], Instrument)
getGlobal lst =
   let gs = getGlobals lst
   in  toMaybe (not (null gs)) (gs, numGlobalInstrs lst)

writeGlobalHeader :: ([GlobalSig], Instrument) -> [String]
writeGlobalHeader (gs,num) =
   let globInit g =
          (globalWrite g ++ " init 0") :
          (globalRead  g ++ " init 0") :
          []
       contents =
          concatMap globInit gs ++
          ("turnon " ++ showInstrumentNumber num) :
          []
   in  "" : contents ++ "" : []

writeOutGlobals :: StatementDefs -> LoopDefs ->
                      [(GlobalSig, SigFixed)] -> [String]
writeOutGlobals noes lps =
   let aux (g, oe) =
          globalWrite g ++ " = " ++ globalWrite g ++ " + " ++
          writeArgs noes lps [oe]
   in  map aux
```

Recall that after processing, the `SigExp` becomes a list of `(Name, StatementDef)` pairs. The last few functions write each of these named `StatementDef`s as a statement in the orchestra file. Whenever a signal generation/modification constructor is encountered in an argument list of another constructor, the argument's string name is used instead, as found in the list of `(Name, StatementDef)` pairs.

The `Orc` Monad The global signals can be somewhat difficult to handle, especially when there are quite a few of them. After all, they must all be different; otherwise, the user may have two instruments writing completely different things to the same signal, and using the same signals for completely different things. However, there is an easier way to do this — a monad that allows for a much simpler way of getting global signals:

```
type Orc a b = State (OrcState a) b
data OrcState a = OrcState [InstrBlock a] Int deriving (Show, Eq)

mkSignalPlain :: EvalRate -> (SigExp -> SigExp -> SigExp) -> OrcState a
                    -> (GlobalSig, OrcState a)
mkSignalPlain rate func (OrcState ibs gCount) =
   (Global rate func gCount, OrcState ibs (gCount + 1))

mkSignal :: Output a => EvalRate -> (SigExp -> SigExp -> SigExp)
```

```
writeOut :: Output a => StatementDefs -> LoopDefs -> InstrBlock a -> [String]
writeOut noes lps (InstrBlock _ _ chnls lst) =
   (getName chnls ++ " " ++ writeArgs noes lps (getFixedExpressions chnls)) :
   writeOutGlobals noes lps (map (mapSnd simpleFix) lst)

writeExp :: StatementDefs -> LoopDefs -> (Name, StatementDef) -> [String]
writeExp noes lps (name, stmt) =
   case stmt of
     StatementDef funcName args ->
        [ifAllowedArgs funcName args
          (name ++ " " ++ funcName ++ " " ++ writeArgs noes lps args)]
     DelayDef (DelayLine _ del) ->
        [name ++ " delayr " ++ showExp noes lps del]
     TapDef funcName _ args ->
        [ifAllowedArgs funcName args
          (name ++ " " ++ funcName ++ " " ++ writeArgs noes lps args)]
     DelayWriteDef (DelayLine sig _) ->
        ["delayw " ++ showExp noes lps sig]
     IndexDef _ _ -> []
     MultiDef funcName args outCount ex {- 'ex' is always a SigGen -} ->
        [ifAllowedArgs funcName args
           (concat (intersperse ", "
              (map (\x -> showExp noes lps
                            (TreeTag.Branch (Index x ex)))
                   [1..outCount]))
            ++ " " ++ funcName ++ " " ++ writeArgs noes lps args)]

ifAllowedArgs :: String -> [SigFixed] -> String -> String
ifAllowedArgs funcName args str =
   if allowedArgs argCountTable funcName (length args)
     then str
     else error ("writeExp: wrong number of arguments " ++
                          "passed to function " ++ funcName)

writeArgs :: StatementDefs -> LoopDefs -> [SigFixed] -> String
writeArgs noes lps =
   concat . intersperse ", " . map (showExp noes lps)
```

Figure 13: The Function `writeExp`

```
showExp :: StatementDefs -> LoopDefs -> SigFixed -> String
showExp noes lps (TreeTag.Branch oe) =
   case oe of
     ConstFloat x  -> show x
     ConstInt n    -> show n
     TableNumber n -> show n
     PField p      -> "p" ++ show p
     Str s         -> show s
     Read var      -> globalRead var
     Conditional b tr fa ->
        "(" ++ showBool noes lps b ++ " ? "
            ++ showExp noes lps tr ++ " : "
            ++ showExp noes lps fa ++ ")"
     Infix nm x1 x2 ->
        "(" ++ showExp noes lps x1 ++ " " ++ nm ++ " "
            ++ showExp noes lps x2 ++ ")"
     Prefix nm x -> nm ++ "(" ++ showExp noes lps x ++ ")"
     SigGen nm _ _ args ->
        lookupDef noes (StatementDef nm args) oe
     Result dl      -> lookupDef noes (DelayDef dl) oe
     Tap nm dl args -> lookupDef noes (TapDef nm dl args) oe
     Index x ex -> lookupDef noes (IndexDef x ex) oe
showExp noes lps (TreeTag.Tag _ ex) =
   showExp noes lps ex
showExp _    lps (TreeTag.Loop s) =
   maybe (error "loop not found") id (lookup s lps)

lookupDef :: (Show a, Eq c) => [(b, c)] -> c -> a -> b
lookupDef noes def oe =
   maybe (error ("showExp " ++ show oe ++ ": constructor not found\n"))
         id (lookup def (map (\(x, y) -> (y, x)) noes))

showBool :: StatementDefs -> LoopDefs -> BooleanFixed -> String
showBool noes lps bool =
   case bool of
      Operator name x1 x2 ->
         "(" ++ showBool noes lps x1 ++ " " ++ name ++ " "
             ++ showBool noes lps x2 ++ ")"
      Comparison name x1 x2 ->
         "(" ++ showExp noes lps x1 ++ " " ++ name ++ " "
             ++ showExp noes lps x2 ++ ")"
```

Figure 14: The Function `showExp`

```
                -> Orc a GlobalSig
mkSignal rate func = state (mkSignalPlain rate func)

addInstrPlain :: Output a => InstrBlock a -> OrcState a -> OrcState a
addInstrPlain ib (OrcState ibs gCount) =
   OrcState (ibs ++ [ib]) gCount

addInstr :: Output a => InstrBlock a -> Orc a ()
addInstr ib = modify (addInstrPlain ib)

runOrc :: Orc a () -> [InstrBlock a]
runOrc comp =
   case execState comp (OrcState [] 1) of
      (OrcState ibs _) -> ibs

mkOrc :: Output a => Header -> Orc a () -> T a
mkOrc hdr = Cons hdr . runOrc
```

The user can call `mkSignal` to get a unique global line, or `addInstr` to add an instrument to the structure. For example:

```
test :: IO ()
test =
   let a1 = oscI AR (tableNumber 1) 1000 440
       comp =
          do h <- mkSignal AR (+)
             addInstr (InstrBlock (instrument 1) 0 (Mono a1) [(h, a1)])
             addInstr (InstrBlock (instrument 2) 0 (Mono (readGlobal h)) [])
   in  saveIA (mkOrc (44100, 4410) comp)
```

The above example has the first instrument writing a simple oscillation to the given audio-rate global signal, and then has the second instrument reading from the same global.

An Orchestra Example Figure 15 shows a typical CSound orchestra file. Figure 16 shows how this same functionality would be achieved in Haskore using an `CSound.Orchestra.T` value. Finally, Figure 17 shows the result of applying `Orchestra.saveIA` to `orc1` shown in Figure 16. Figures 15 and 17 should be compared: you will note that except for name changes, they are the same, as they should be.

4.3.3 Tutorial

```
module Haskore.Interface.CSound.Tutorial where

import Haskore.Interface.CSound.Orchestra
          (SigExp, Mono(Mono), Stereo(Stereo), Output, Name,
           pchToHz, dbToAmp, sigGen, rec, tableNumber, EvalRate(AR, CR),
           osc, oscI, randomI, expon, reverb, vdelay, comb, lineSeg,
           PluckDecayMethod(..), pluck, buzz)
import Haskore.Interface.CSound.Generator
          (compSine1, compSine2, cubicSpline, lineSeg1)
import Haskore.Interface.CSound.Score     as Score
```

```
sr = 48000
kr = 24000
ksmps = 2
nchnls = 2

instr 4

inote = cpspch(p5)

k1 envlpx ampdb(p4), .001, p3, .05, 6, -.1, .01
k2 envlpx ampdb(p4), .0005, .1, .1, 6, -.05, .01
k3 envlpx ampdb(p4), .001, p3, p3, 6, -.3, .01

a1 oscili k1, inote, 1
a2 oscili k1, inote * 1.004, 1
a3 oscili k2, inote * 16, 1
a4 oscili k3, inote, 5
a5 oscili k3, inote * 1.004, 5

outs  (a2 + a3 + a4) * .75, (a1 + a3 + a5) * .75

endin
```

Figure 15: Sample CSound Orchestra File

```
orc1 :: T Stereo
orc1 =
  let hdr   = (48000, 24000)
      inote = pchToHz p5
      k1    = env  CR 6 (-0.1)  0.01 0 0.05 0.001  p3  (dbToAmp p4)
      k2    = env  CR 6 (-0.05) 0.01 0 0.1  0.0005 0.1 (dbToAmp p4)
      k3    = env  CR 6 (-0.3)  0.01 0 p3   0.001  p3  (dbToAmp p4)
      t1    = tableNumber 1
      t5    = tableNumber 5
      a1    = oscI AR t1 k1  inote
      a2    = oscI AR t1 k1 (inote*1.004)
      a3    = oscI AR t1 k2 (inote*16)
      a4    = oscI AR t5 k3  inote
      a5    = oscI AR t5 k3 (inote*1.004)
      out   = Stereo ((a2+a3+a4) * 0.75) ((a1+a3+a5) * 0.75)
      ib    = InstrBlock (instrument 4) 0 out []
  in Cons hdr [ib]

test1 :: StatementDefs
test1 = extractFunctions $ mkListOut (head ((\(Cons _ x) -> x) orc1))
```

Figure 16: Haskore Orchestra Definition

```
sr     = 48000
kr     = 24000
ksmps  = 2.0
nchnls = 2

instr 4
k1 envlpx ampdb(p4), 1.0e-3, p3, p3, 6.0, -(0.3), 1.0e-2, 0.0
a2 oscili k1, (cpspch(p5) * 1.004), 5
k3 envlpx ampdb(p4), 5.0e-4, 0.1, 0.1, 6.0, -(5.0e-2), 1.0e-2, 0.0
a4 oscili k3, (cpspch(p5) * 16.0), 1
k5 envlpx ampdb(p4), 1.0e-3, p3, 5.0e-2, 6.0, -(0.1), 1.0e-2, 0.0
a6 oscili k5, cpspch(p5), 1
a7 oscili k1, cpspch(p5), 5
a8 oscili k5, (cpspch(p5) * 1.004), 1
outs (((a8 + a4) + a7) * 0.75), (((a6 + a4) + a2) * 0.75)
endin
```

Figure 17: Result of `Orchestra.saveIA orc1`

```
import qualified Haskore.Interface.CSound.Orchestra as Orchestra
import qualified Haskore.Interface.CSound.SoundMap as SoundMap
import qualified Haskore.Interface.CSound as CSound

import qualified Haskore.Performance         as Performance
import qualified Haskore.Performance.Context as Context
import qualified Haskore.Performance.Fancy   as FancyPerformance

import qualified Haskore.Music          as Music
import qualified Haskore.Music.Rhythmic as RhyMusic

import qualified Numeric.NonNegative.Wrapper as NonNeg

import Haskore.Basic.Duration
import Haskore.Music ((+:+), (=:=), qnr)
import Haskore.Melody as Melody

import System.Cmd (system, )
import System.Exit (ExitCode, )
```

This brief tutorial is designed to introduce the user to the capabilities of the CSound software synthesizer and sound synthesis in general.

Additive Synthesis The first part of the tutorial introduces *additive synthesis*. Additive synthesis is the most basic, yet the most powerful synthesis technique available, giving complete control over the sound waveform. The basic premiss behind additive sound synthesis is quite simple – defining a complex sound by specifying each contributing sine wave. The computer is very good at generating pure tones, but these are not very interesting. However, any sound imaginable can be reproduced as a sum of pure tones. We can define an instrument of pure tones easily in Haskore. First we define a *Function table* containing a lone

sine wave. We can do this using the `simpleSine` function defined in the module `CSound.Orchestra` module:

```
pureToneTN :: Score.Table
pureToneTN = 1
pureToneTable :: SigExp
pureToneTable = tableNumber pureToneTN
pureTone :: Score.Statement
pureTone = Score.Table pureToneTN 0 8192 True (compSine1 [1.0])

oscPure :: SigExp -> SigExp -> SigExp
oscPure = osc AR pureToneTable
```

`pureToneTN` is the table number of the simple sine wave. We will adopt the convention in this tutorial that variables ending with `TN` represent table numbers. Recall that `compSine1` is defined in the module module `CSound` as a sine wave generating routine (GEN10). In order to have a complete score file, we also need a tune. Here is a simple example:

```
type TutMelody params = Melody.T (TutAttr params)

data TutAttr params =
     TutAttr {attrVelocity   :: Rational,
              attrParameters :: params}

tune1 :: TutMelody ()
tune1 = Music.line (map ($ TutAttr 1.5 ())
                [ c 1 hn, e 1 hn,  g 1 hn,
                  c 2 hn, a 1 hn,  c 2 qn,
                  a 1 qn, g 1 dhn ] ++ [qnr])
```

The next step is to convert the melody into a music. In our simple tutorial we have only one instrument per song in all but one case. So we could skip this step, but we want to include it in order to show the general processing steps. We use the general data type for rhythmic music, with no drum definitions (null type `()`) and a custom instrument definition `Instrument`. We use only the instrument numbers 1 and 2 but the numbers are associated with different sounds in the examples.

```
data Instrument =
       Instr1p0
     | Instr2p0
     | Instr1p2 Float Float
     | Instr1p4 Float Float Float Float
   deriving (Eq, Ord, Show)

musicFromMelody :: (params -> Instrument) ->
   TutMelody params -> RhyMusic.T () Instrument
musicFromMelody instr =
   RhyMusic.fromMelody
      (\(TutAttr vel params) -> (vel, instr params))
```

The melody contains instrument specific parameters. They will be embedded in `Instrument` values by the following functions. These functions can be used as `instr` arguments to `musicFromMelody`.

```
type Pair      = (Float, Float)
```

```
type Quadruple = (Float, Float, Float, Float)

attrToInstr1p0 :: () -> Instrument
attrToInstr1p0 () = Instr1p0

attrToInstr2p0 :: () -> Instrument
attrToInstr2p0 () = Instr2p0

attrToInstr1p2 :: Pair -> Instrument
attrToInstr1p2 = uncurry Instr1p2

attrToInstr1p4 :: Quadruple -> Instrument
attrToInstr1p4 (x,y,z,w) = Instr1p4 x y z w
```

There is nothing special about the conversion from the music to the performance.

```
performanceFromMusic :: RhyMusic.T () Instrument ->
   Performance.T NonNeg.Float Float (RhyMusic.Note () Instrument)
performanceFromMusic =
   FancyPerformance.fromMusicModifyContext (Context.setDur 1)
```

Now we convert from the performance to the CSound score. To this end we must convert the instruments represented by `Instrument` to sound numbers and parameter fields. A `SoundMap.InstrumentTableWithAttributes out Instrument` must be generated for the conversion. The functions like `instrAssoc1p0` generate one entry for the table which assigns an instrument number and a sound algorithm to a constructor of `Instrument`.

```
type TutOrchestra out =
   (Orchestra.Header, SoundMap.InstrumentTableWithAttributes out Instrument)

instrNum1, instrNum2 :: CSound.Instrument
instrNum1 = CSound.instrument 1
instrNum2 = CSound.instrument 2

instrAssoc1p0 :: SoundMap.InstrumentSigExp out ->
   SoundMap.InstrumentAssociation out Instrument
instrAssoc1p0 =
   SoundMap.instrument instrNum1
      (\i -> do Instr1p0 <- Just i; Just ())

instrAssoc2p0 :: SoundMap.InstrumentSigExp out ->
   SoundMap.InstrumentAssociation out Instrument
instrAssoc2p0 =
   SoundMap.instrument instrNum2
      (\i -> do Instr2p0 <- Just i; Just ())

instrAssoc1p2 :: (SigExp -> SigExp -> SoundMap.InstrumentSigExp out) ->
   SoundMap.InstrumentAssociation out Instrument
instrAssoc1p2 =
   SoundMap.instrument2 instrNum1
      (\i -> do Instr1p2 x y <- Just i; Just (x,y))

instrAssoc1p4 :: (SigExp -> SigExp -> SigExp -> SigExp -> SoundMap.InstrumentSigExp out) ->
   SoundMap.InstrumentAssociation out Instrument
```

```
instrAssoc1p4 =
   SoundMap.instrument4 instrNum1
      (\i -> do Instr1p4 x y z w <- Just i; Just (x,y,z,w))
```

The function `scored` puts the chain from melody to CSound score together. Finally the function `example` collects music and instrument definitions, that is a complete example.

```
scored :: TutOrchestra out -> (params -> Instrument) ->
             TutMelody params -> Score.T
scored (_,sndMap) instr =
   Score.fromRhythmicPerformanceWithAttributes
      (error "no drum map defined") sndMap .
   performanceFromMusic .
   musicFromMelody instr

example :: Name -> (TutOrchestra out -> Score.T) -> TutOrchestra out ->
              (Name, Score.T, TutOrchestra out)
example name mkScore orc = (name, mkScore orc, orc)
```

Let's define an instrument in the orchestra file that will use the function table `pureTone`:

```
oe1 :: SoundMap.InstrumentSigExp Mono
oe1 _noteDur noteVel notePit =
      let signal = oscPure (dbToAmp noteVel) (pchToHz notePit)
      in  Mono signal

score1 orc = pureTone : scored orc attrToInstr1p0 tune1
```

This instrument will simply oscillate through the function table containing the sine wave at the appropriate frequency given by `notePit`, and the resulting sound will have an amplitude given by `noteVel`. Note that the `oe1` expression above is a `Mono`, not a complete `TutOrchestra`. We need to define a *header* and associate `oe1` with the instrument that's playing it:

```
hdr :: Orchestra.Header
hdr = (44100, 4410)

o1, o2, o3, o4, o7, o8, o9, o13, o14, o15, o19, o22
   :: TutOrchestra Mono
o5, o6, o10, o11, o12, o16, o17, o18, o20, o21
   :: TutOrchestra Stereo

tut1, tut2, tut3, tut4, tut7, tut8, tut9, tut13, tut14, tut15, tut19, tut22
   :: (Name, Score.T, TutOrchestra Mono)
tut5, tut6, tut10, tut11, tut12, tut16, tut17, tut18, tut20, tut21
   :: (Name, Score.T, TutOrchestra Stereo)

score1, score2, score3, score4, score5, score6, score7, score8, score9
   :: TutOrchestra out -> [Score.Statement]

o1 = (hdr, [instrAssoc1p0 oe1])
```

The header above indicates that the audio signals are generated at 44,100 Hz (CD quality), the control signals are generated at 4,410 Hz, and there are 2 output channels for stereo sound. Now we have a complete score and orchestra that can be converted to a sound file by CSound and played as follows:

```
csoundDir :: Name
csoundDir = "/tmp"
-- csoundDir = "C:/TEMP/csound"

tut1 = example "tut01" score1 o1
```

If you listen to the tune, you will notice that it sounds very thin and uninteresting. Most musical sounds are not pure. Instead they usually contain a sine wave of dominant frequency, called a *fundamental*, and a number of other sine waves called *partials*. Partials with frequencies that are integer multiples of the fundamental are called *harmonics*. In musical terms, the first harmonic lies an octave above the fundamental, second harmonic a fifth above the first one, the third harmonic lies a major third above the second harmonic etc. This is the familiar *overtone series*. We can add harmonics to our sine wave instrument easily using the `compSine` function defined in the module `CSound.Orchestra` module. The function takes a list of harmonic strengths as arguments. The following creates a function table containing the fundamental and the first two harmonics at two thirds and one third of the strength of the fundamental:

```
twoHarmsTN :: Score.Table
twoHarmsTN = 2
twoHarms :: Score.Statement
twoHarms = Score.Table twoHarmsTN 0 8192 True (compSine1 [1.0, 0.66, 0.33])
```

We can again proceed to create complete score and orchestra files as above:

```
score2 orc = twoHarms : scored orc attrToInstr1p0 tune1

oe2 :: SoundMap.InstrumentSigExp Mono
oe2 _noteDur noteVel notePit =
   let signal = osc AR (tableNumber twoHarmsTN)
                    (dbToAmp noteVel) (pchToHz notePit)
   in  Mono signal

o2 = (hdr, [instrAssoc1p0 oe2])

tut2 = example "tut02" score2 o2
```

The orchestra file is the same as before – a single oscillator scanning a function table at a given frequency and volume. This time, however, the tune will not sound as thin as before since the table now contains a function that is an addition of three sine waves. (Note that the same effect could be achieved using a simple sine wave table and three oscillators). Not all musical sounds contain harmonic partials exclusively, and never do we encounter instruments with static amplitude envelope like the ones we have seen so far. Most sounds, musical or not, evolve and change throughout their duration. Let's define an instrument containing both harmonic and nonharmonic partials, that starts at maximum amplitude with a straight line decay. We will use the function `compSine2` from the module `CSound.Orchestra` module to create the function table. `compSine2` takes a list of triples as an argument. The triples specify the partial number as a multiple of the fundamental, relative partial strength, and initial phase offset:

```
manySinesTN :: Score.Table
manySinesTN = 3
manySinesTable :: SigExp
manySinesTable = tableNumber manySinesTN
```

```
manySines :: Score.Statement
manySines = Score.Table manySinesTN 0 8192 True (compSine2 [(0.5, 0.9, 0.0),
                    (1.0, 1.0, 0.0), (1.1, 0.7, 0.0), (2.0, 0.6, 0.0),
                    (2.5, 0.3, 0.0), (3.0, 0.33, 0.0), (5.0, 0.2, 0.0)])
```

Thus this complex will contain the second, third, and fifth harmonic, nonharmonic partials at frequencies of 1.1 and 2.5 times the fundamental, and a component at half the frequency of the fundamental. Their strengths relative to the fundamental are given by the second argument, and they all start in sync with zero offset. Now we can proceed as before to create score and orchestra files. We will define an *amplitude envelope* to apply to each note as we oscillate through the table. The amplitude envelope will be a straight line signal ramping from 1.0 to 0.0 over the duration of the note. This signal will be generated at *control rate* rather than audio rate, because the control rate is more than sufficient (the audio signal will change volume 4,410 times a second), and the slower rate will improve performance.

```
score3 orc = manySines : scored orc attrToInstr1p0 tune1

lineCS :: EvalRate -> SigExp -> SigExp
              -> SigExp -> SigExp
lineCS  = Orchestra.line

oe3 :: SoundMap.InstrumentSigExp Mono
oe3 noteDur noteVel notePit =
   let ampEnv = lineCS CR 1.0 noteDur 0.0
       signal = osc AR manySinesTable
                    (ampEnv * dbToAmp noteVel) (pchToHz notePit)
   in  Mono signal

o3 = (hdr, [instrAssoc1p0 oe3])

tut3 = example "tut03" score3 o3
```

Not only do musical sounds usually evolve in terms of overall amplitude, they also evolve their *spectra*. In other words, the contributing partials do not usually all have the same amplitude envelope, and so their contribution to the overall sound isn't static. Let us illustrate the point using the same set of partials as in the above example. Instead of creating a table containing a complex waveform, however, we will use multiple oscillators going through the simple sine wave table we created at the beginning of this tutorial at the appropriate frequencies. Thus instead of the partials being fused together, each can have its own amplitude envelope, making the sound evolve over time. The score will be score1, defined above.

```
oe4 :: SoundMap.InstrumentSigExp Mono
oe4 noteDur noteVel notePit =
   let pitch      = pchToHz notePit
       amp        = dbToAmp noteVel
       mkLine t   = lineSeg CR 0 (noteDur*t) 1 [(noteDur * (1-t), 0)]
       aenv1      = lineCS CR 1 noteDur 0
       aenv2      = mkLine 0.17
       aenv3      = mkLine 0.33
       aenv4      = mkLine 0.50
       aenv5      = mkLine 0.67
       aenv6      = mkLine 0.83
       aenv7      = lineCS CR 0 noteDur 1
       mkOsc ae p = oscPure (ae * amp) (pitch * p)
```

```
       a1          = mkOsc aenv1 0.5
       a2          = mkOsc aenv2 1.0
       a3          = mkOsc aenv3 1.1
       a4          = mkOsc aenv4 2.0
       a5          = mkOsc aenv5 2.5
       a6          = mkOsc aenv6 3.0
       a7          = mkOsc aenv7 5.0
       out         = 0.5 * (a1 + a2 + a3 + a4 + a5 + a6 + a7)
   in  Mono out

o4 = (hdr, [instrAssoc1p0 oe4])

tut4 = example "tut04" score1 o4
```

So far, we have only used function tables to generate audio signals, but they can come very handy in *modifying* signals. Let us create a function table that we can use as an amplitude envelope to make our instrument more interesting. The envelope will contain an immediate sharp attack and decay, and then a second, more gradual one, so we'll have two attack/decay events per note. We'll use the cubic spline curve generating routine to do this:

```
coolEnvTN :: Score.Table
coolEnvTN = 4
coolEnvTable :: SigExp
coolEnvTable = tableNumber coolEnvTN
coolEnv :: Score.Statement
coolEnv = Score.Table coolEnvTN 0 8192 True
             (cubicSpline 1 [(1692, 0.2), (3000, 1), (3500, 0)])

oscCoolEnv :: SigExp -> SigExp -> SigExp
oscCoolEnv = osc CR coolEnvTable
```

Let us also add some *p-fields* to the notes in our score. The two p-fields we add will be used for *panning* – the first one will be the starting percentage of the left channel, the second one the ending percentage (1 means all left, 0 all right, 0.5 middle. Pfields of 1 and 0 will cause the note to pan completely from left to right for example)

```
tune2 :: TutMelody Pair
tune2 =
   let attr start end = TutAttr 1.4 (start, end)
   in  c 1 hn (attr 1.0 0.75) +:+
       e 1 hn (attr 0.75 0.5) +:+
       g 1 hn (attr 0.5 0.25) +:+
       c 2 hn (attr 0.25 0.0) +:+
       a 1 hn (attr 0.0 1.0)  +:+
       c 2 qn (attr 0.0 0.0)  +:+
       a 1 qn (attr 1.0 1.0)  +:+
      (g 1 dhn (attr 1.0 0.0) =:=
       g 1 dhn (attr 0.0 1.0))+:+ qnr
```

So far we have limited ourselves to using only sine waves for our audio output, even though Csound places no such restrictions on us. Any repeating waveform, of any shape, can be used to produce pitched sounds. In essence, when we are adding sinewaves, we are changing the shape of the wave. For example, adding

odd harmonics to a fundamental at strengths equal to the inverse of their partial number (ie. third harmonic would be 1/3 the strength of the fundamental, fifth harmonic 1/5 the fundamental etc) would produce a *square* wave which has a raspy sound to it. Another common waveform is the *sawtooth*, and the more mellow sounding *triangle*. The module `CSound.Orchestra` module already contains functions to create these common waveforms. Let's use them to create tables that we can use in an instrument:

```
triangleTN, squareTN, sawtoothTN :: Score.Table
triangleTN = 5
squareTN   = 6
sawtoothTN = 7
triangleT, squareT, sawtoothT :: Score.Statement
triangleT = triangle triangleTN
squareT   = square   squareTN
sawtoothT = sawtooth sawtoothTN

score4 orc = squareT : triangleT : sawtoothT : coolEnv :
                   scored orc attrToInstr1p2 (Music.changeTempo 0.5 tune2)

oe5 :: SigExp -> SigExp -> SoundMap.InstrumentSigExp Stereo
oe5 panStart panEnd noteDur noteVel notePit =
   let pitch  = pchToHz notePit
       amp    = dbToAmp noteVel
       pan    = lineCS CR panStart noteDur panEnd
       oscF   = 1 / noteDur
       ampen  = oscCoolEnv amp oscF
       signal = osc AR (tableNumber squareTN) ampen pitch
       left   = signal * pan
       right  = signal * (1-pan)
   in  Stereo left right

o5 = (hdr, [instrAssoc1p2 oe5])

tut5 = example "tut05" score4 o5
```

This will oscillate through a table containing the square wave. Check out the other waveforms too and see what they sound like. This can be done by specifying the table to be used in the orchestra file. As our last example of additive synthesis, we will introduce an orchestra with multiple instruments. The bass will be mostly in the left channel, and will be the same as the third example instrument in this section. It will play the tune two octaves below the instrument in the right channel, using an orchestra identical to `oe3` with the addition of the panning feature:

```
score5 orc = manySines : pureTone : scored orc attrToInstr1p0 tune1 ++
                                    scored orc attrToInstr2p0 tune1

oe6 :: SoundMap.InstrumentSigExp Stereo
oe6 noteDur noteVel notePit =
   let ampEnv = lineCS CR 1.0 noteDur 0.0
       signal = osc AR manySinesTable
                    (ampEnv * dbToAmp noteVel) (pchToHz (notePit - 2))
       left   = 0.8 * signal
       right  = 0.2 * signal
   in  Stereo left right

oe7 :: SoundMap.InstrumentSigExp Stereo
```

```
oe7 noteDur noteVel notePit =
   let pitch      = pchToHz notePit
       amp        = dbToAmp noteVel
       mkLine t   = lineSeg CR 0 (noteDur*t) 0.5 [(noteDur * (1-t), 0)]
       aenv1      = lineCS CR 0.5 noteDur 0
       aenv2      = mkLine 0.17
       aenv3      = mkLine 0.33
       aenv4      = mkLine 0.50
       aenv5      = mkLine 0.67
       aenv6      = mkLine 0.83
       aenv7      = lineCS CR 0 noteDur 0.5
       mkOsc ae p = oscPure (ae * amp) (pitch * p)
       a1         = mkOsc aenv1 0.5
       a2         = mkOsc aenv2 1.0
       a3         = mkOsc aenv3 1.1
       a4         = mkOsc aenv4 2.0
       a5         = mkOsc aenv5 2.5
       a6         = mkOsc aenv6 3.0
       a7         = mkOsc aenv7 5.0
       left       = 0.2 * (a1 + a2 + a3 + a4 + a5 + a6 + a7)
       right      = 0.8 * (a1 + a2 + a3 + a4 + a5 + a6 + a7)
   in  Stereo left right

o6 = (hdr, [instrAssoc1p0 oe6, instrAssoc2p0 oe7])

tut6 = example "tut06" score5 o6
```

Additive synthesis is the most powerful tool in computer music and sound synthesis in general. It can be used to create any sound imaginable, whether completely synthetic or a simulation of a real-world sound, and everyone interested in using the computer to synthesize sound should be well versed in it. The most significant drawback of additive synthesis is that it requires huge amounts of control data, and potentially thousands of oscillators. There are other synthesis techniques, such as *modulation synthesis*, that can be used to create rich and interesting timbres at a fraction of the cost of additive synthesis, though no other synthesis technique provides quite the same degree of control.

Modulation Synthesis While additive synthesis provides full control and great flexibility, it is quiet clear that the enormous amounts of control data make it impractical for even moderately complicated sounds. There is a class of synthesis techniques that use *modulation* to produce rich, time-varying timbres at a fraction of the storage and time cost of additive synthesis. The basic idea behind modulation synthesis is controlling the amplitude and/or frequency of the main periodic signal, called the *carrier*, by another periodic signal, called the *modulator*. The two main kinds of modulation synthesis are *amplitude modulation* and *frequency modulation* synthesis. Let's start our discussion with the simpler one of the two – amplitude synthesis. We have already shown how to supply a time varying amplitude envelope to an oscillator. What would happen if this amplitude envelope was itself an oscillating signal? Supplying a low frequency ($<$ 20Hz) modulating signal would create a predictable effect – we would hear the volume of the carrier signal go periodically up and down. However, as the modulator moves into the audible frequency range, the carrier changes timbre as new frequencies appear in the spectrum. The new frequencies are equal to the sum and difference of the carrier and modulator. So for example, if the frequency of the main signal (carrier) is C = 500Hz, and the frequency of the modulator is M = 100Hz, the audible frequencies will be the carrier

C (500Hz), C + M (600Hz), and C - M (400Hz). The amplitude of the two new sidebands depends on the amplitude of the modulator, but will never exceed half the amplitude of the carrier. The following is a simple example that demonstrates amplitude modulation. The carrier will be a 10 second pure tone at 500Hz. The frequency of the modulator will increase linearly over the 10 second duration of the tone from 0 to 200 Hz. Initially, you will be able to hear the volume of the signal fluctuate, but after a couple of seconds the volume will seem constant as new frequencies appear. Let us first create the score file. It will contain a sine wave table, and a single note event:

```
score6 _ =
   pureTone : [ Score.Note instrNum1 0.0 10.0 (Cps 500.0) 10000.0 [] ]
```

The orchestra will contain a single AM instrument. The carrier will simply oscillate through the sine wave table at frequency given by the note pitch (500Hz, see the score above), and amplitude given by the modulator. The modulator will oscillate through the same sine wave table at frequency ramping from 0 to 200Hz. The modulator should be a periodic signal that varies from 0 to the maximum volume of the carrier. Since the sine wave goes from -1 to 1, we will need to add 1 to it and half it, before multiplying it by the volume supplied by the note event. This will be the modulating signal, and the carrier's amplitude input. (note that we omit the conversion functions dbToAmp and notePit, since we supply the amplitude and frequency in their raw units in the score file)

```
oe8 :: SoundMap.InstrumentSigExp Mono
oe8 noteDur noteVel notePit =
   let modFreq = lineCS CR 0.0 noteDur 200.0
       modAmp  = oscPure 1.0 modFreq
       modSig  = (modAmp + 1.0) * 0.5 * noteVel
       carrier = oscPure modSig notePit
   in  Mono carrier

o7 = (hdr, [instrAssoc1p0 oe8])

tut7 = example "tut07" score6 o7
```

Next synthesis technique on the palette is *frequency modulation*. As the name suggests, we modulate the frequency of the carrier. Frequency modulation is much more powerful and interesting than amplitude modulation, because instead of getting two sidebands, FM gives a *number* of spectral sidebands. Let us begin with an example of a simple FM. We will again use a single 10 second note and a 500Hz carrier. Remember that when we talked about amplitude modulation, the amplitude of the sidebands was dependent upon the amplitude of the modulator. In FM, the modulator amplitude plays a much bigger role, as we will see soon. To negate the effect of the modulator amplitude, we will keep the ratio of the modulator amplitude and frequency constant at 1.0 (we will explain shortly why). The frequency and amplitude of the modulator will ramp from 0 to 200 over the duration of the note. This time, though, unlike with AM, we will hear a whole series of sidebands. The orchestra is just as before, except we modulate the frequency instead of amplitude.

```
oe9 :: SoundMap.InstrumentSigExp Mono
oe9 noteDur noteVel notePit =
   let modFreq = lineCS CR 0.0 noteDur 200.0
       modAmp  = modFreq
       modSig  = oscPure modAmp modFreq
       carrier = oscPure noteVel (notePit + modSig)
```

```
   in  Mono carrier

o8 = (hdr, [instrAssoc1p0 oe9])

tut8 = example "tut08" score6 o8
```

The sound produced by FM is a little richer but still very bland. Let us talk now about the role of the *depth* of the frequency modulation (the amplitude of the modulator). Unlike in AM, where we only had one spectral band on each side of the carrier frequency (ie we heard C, C+M, C-M), FM gives a much richer spectrum with many sidebands. The frequencies we hear are C, C+M, C-M, C+2M, C-2M, C+3M, C-3M etc. The amplitudes of the sidebands are determined by the *modulation index* I, which is the ratio between the amplitude (also referred to as depth) and frequency of the modulator (I = D / M). As a rule of thumb, the number of significant sideband pairs (at least 1number of sidebands) increases, energy is "stolen" from the carrier and distributed among the sidebands. Thus if I=1, we have 2 significant sideband pairs, and the audible frequencies will be C, C+M, C-M, C+2M, C-2M, with C, the carrier, being the dominant frequency. When I=5, we will have a much richer sound with about 6 significant sideband pairs, some of which will actually be louder than the carrier. Let us explore the effect of the modulation index in the following example. We will keep the frequency of the carrier and the modulator constant at 500Hz and 80 Hz respectively. The modulation index will be a stepwise function from 1 to 10, holding each value for one second. So in effect, during the first second (I = D/M = 1), the amplitude of the modulator will be the same as its frequency (80). During the second second (I = 2), the amplitude will be double the frequency (160), then it will go to 240, 320, etc:

```
oe10 :: SoundMap.InstrumentSigExp Mono
oe10 _noteDur noteVel notePit =
   let modInd  = lineSeg CR 1 1 1 [(0,2), (1,2), (0,3), (1,3), (0,4),
                                   (1,4), (0,5), (1,5), (0,6), (1,6),
                                   (0,7), (1,7), (0,8), (0,9), (1,9),
                                   (0,10), (1,10)]
       modAmp  = 80.0 * modInd
       modSig  = oscPure modAmp 80.0
       carrier = oscPure noteVel (notePit + modSig)
   in  Mono carrier

o9 = (hdr, [instrAssoc1p0 oe10])

tut9 = example "tut09" score6 o9
```

Notice that when the modulation index gets high enough, some of the sidebands have negative frequencies. For example, when the modulation index is 7, there is a sideband present in the sound with a frequency C - 7M = 500 - 560 = -60Hz. The negative sidebands get reflected back into the audible spectrum but are *phase shifted* 180 degrees, so it is an inverse sine wave. This makes no difference when the wave is on its own, but when we add it to its inverse, the two will cancel out. Say we set the frequency of the carrier at 100Hz instead of 80Hz. Then at I=6, we would have present two sidebands of the same frequency - C-4M = 100Hz, and C-6M = -100Hz. When these two are added, they would cancel each other out (if they were the same amplitude; if not, the louder one would be attenuated by the amplitude of the softer one). The following flexible instrument will sum up simple FM. The frequency of the modulator will be determined by the C/M ratio supplied as p6 in the score file. The modulation index will be a linear slope going from 0 to p7 over the duration of each note. Let us also add panning control as in additive synthesis - p8 will be the initial left

channel percentage, and p9 the final left channel percentage:

```
oe11 :: SigExp -> SigExp -> SigExp -> SigExp -> SoundMap.InstrumentSigExp Stereo
oe11 modFreqRatio modIndEnd panStart panEnd noteDur noteVel notePit =
   let carFreq = pchToHz notePit
       carAmp  = dbToAmp noteVel
       modFreq = carFreq * modFreqRatio
       modInd  = lineCS CR 0 noteDur modIndEnd
       modAmp  = modFreq * modInd
       modSig  = oscPure modAmp modFreq
       carrier = oscPure carAmp (carFreq + modSig)
       mainAmp = oscCoolEnv 1.0 (1/noteDur)
       pan     = lineCS CR panStart noteDur panEnd
       left    = mainAmp * pan * carrier
       right   = mainAmp * (1 - pan) * carrier
   in  Stereo left right

o10 = (hdr, [instrAssoc1p4 oe11])
```

Let's write a cool tune to show off this instrument. Let's keep it simple and play the chord progression Em - C - G - D a few times, each time changing some of the parameters:

```
emChord, cChord, gChord, dChord ::
   Float -> Float -> Float -> Float ->
      TutMelody Quadruple

quickChord ::
   [Music.Dur -> TutAttr Quadruple -> TutMelody Quadruple] ->
   Float -> Float -> Float -> Float ->
      TutMelody Quadruple
quickChord ns x y z w = Music.chord $
   map (\p -> p wn (TutAttr 1.4 (x, y, z, w))) ns

emChord = quickChord [e 0, g  0, b 0]
cChord  = quickChord [c 0, e  0, g 0]
gChord  = quickChord [g 0, b  0, d 1]
dChord  = quickChord [d 0, fs 0, a 0]

tune3 :: TutMelody Quadruple
tune3 =
    Music.transpose (-12) $
        emChord 3.0 2.0 0.0 1.0  +:+  cChord  3.0  5.0 1.0 0.0  +:+
        gChord  3.0 8.0 0.0 1.0  +:+  dChord  3.0 12.0 1.0 0.0  +:+
        emChord 3.0 4.0 0.0 0.5  +:+  cChord  5.0  4.0 0.5 1.0  +:+
        gChord  8.0 4.0 1.0 0.5  +:+  dChord 10.0  4.0 0.5 0.0  +:+
        (emChord 4.0 6.0 1.0 0.0  =:=  emChord 7.0  5.0 0.0 1.0)  +:+
        (cChord  5.0 9.0 1.0 0.0  =:=  cChord  9.0  5.0 0.0 1.0)  +:+
        (gChord  5.0 5.0 1.0 0.0  =:=  gChord  7.0  7.0 0.0 1.0)  +:+
        (dChord  2.0 3.0 1.0 0.0  =:=  dChord  7.0 15.0 0.0 1.0)
```

Now we can create a score. It will contain two wave tables – one containing the sine wave, and the other containing an amplitude envelope, which will be the table coolEnv which we have already seen before

```
score7 orc = pureTone : coolEnv :
               scored orc attrToInstr1p4 (Music.changeTempo 0.5 tune3)
```

```
tut10 = example "tut10" score7 o10
```

Note that all of the above examples of frequency modulation use a single carrier and a single modulator, and both are oscillating through the simplest of waveforms – a sine wave. Already we have achieved some very rich and interesting timbres using this simple technique, but the possibilities are unlimited when we start using different carrier and modulator waveshapes and multiple carriers and/or modulators. Let us include a couple more examples that will play the same chord progression as above with multiple carriers, and then with multiple modulators. The reason for using multiple carriers is to obtain /em formant regions in the spectrum of the sound. Recall that when we modulate a carrier frequency we get a spectrum with a central peak and a number of sidebands on either side of it. Multiple carriers introduce additional peaks and sidebands into the composite spectrum of the resulting sound. These extra peaks are called formant regions, and are characteristic of human voice and most musical instruments

```
oe12 :: SigExp -> SigExp -> SigExp -> SigExp -> SoundMap.InstrumentSigExp Stereo
oe12 modFreqRatio modIndEnd panStart panEnd noteDur noteVel notePit =
   let car1Freq = pchToHz notePit
       car2Freq = pchToHz (notePit + 1)
       car1Amp  = dbToAmp noteVel
       car2Amp  = dbToAmp noteVel * 0.7
       modFreq  = car1Freq * modFreqRatio
       modInd   = lineCS CR 0 noteDur modIndEnd
       modAmp   = modFreq * modInd
       modSig   = oscPure modAmp modFreq
       carrier1 = oscPure car1Amp (car1Freq + modSig)
       carrier2 = oscPure car2Amp (car2Freq + modSig)
       mainAmp  = oscCoolEnv 1.0 (1/noteDur)
       pan      = lineCS CR panStart noteDur panEnd
       left     = mainAmp * pan * (carrier1 + carrier2)
       right    = mainAmp * (1 - pan) * (carrier1 + carrier2)
   in  Stereo left right

o11 = (hdr, [instrAssoc1p4 oe12])

tut11 = example "tut11" score7 o11
```

In the above example, there are two formant regions – one is centered around the note pitch frequency provided by the score file, the other an octave above. Both are modulated in the same way by the same modulator. The sound is even richer than that obtained by simple FM. Let us now turn to multiple modulator FM. In this case, we use a signal to modify another signal, and the modified signal will itself become a modulator acting on the carrier. Thus the wave that wil be modulating the carrier is not a sine wave as above, but is itself a complex waveform resulting from simple FM. The spectrum of the sound will contain a central peak frequency, surrounded by a number of sidebands, but this time each sideband will itself also by surrounded by a number of sidebands of its own. So in effect we are talking about "double" modulation, where each sideband is a central peak in its own little spectrum. Multiple modulator FM thus provides extremely rich spectra

```
oe13 :: SigExp -> SigExp -> SigExp -> SigExp -> SoundMap.InstrumentSigExp Stereo
oe13 modFreqRatio modIndEnd panStart panEnd noteDur noteVel notePit =
   let carFreq  = pchToHz notePit
       carAmp   = dbToAmp noteVel
```

```
       mod1Freq = carFreq * modFreqRatio
       mod2Freq = mod1Freq * 2.0
       modInd   = lineCS CR 0 noteDur modIndEnd
       mod1Amp  = mod1Freq * modInd
       mod2Amp  = mod1Amp * 3.0
       mod1Sig  = oscPure mod1Amp mod1Freq
       mod2Sig  = oscPure mod2Amp (mod2Freq + mod1Sig)
       carrier  = oscPure carAmp  (carFreq  + mod2Sig)
       mainAmp  = oscCoolEnv 1.0 (1/noteDur)
       pan      = lineCS CR panStart noteDur panEnd
       left     = mainAmp * pan * carrier
       right    = mainAmp * (1 - pan) * carrier
   in  Stereo left right

o12 = (hdr, [instrAssoc1p4 oe13])

tut12 = example "tut12" score7 o12
```

In fact, the spectra produced by multiple modulator FM are so rich and complicated that even the moderate values used as arguments in our tune produce spectra that are saturated and otherworldly. And we did this while keeping the ratios of the two modulators frequencies and amplitudes constant; introducing dynamics in those ratios would produce even crazier results. It is quite amazing that from three simple sine waves, the purest of all tones, we can derive an unlimited number of timbres. Modulation synthesis is a very powerful tool and understanding how to use it can prove invaluable. The best way to learn how to use FM effectively is to dabble and experiment with different ratios, formant regions, dynamic relationships betweeen ratios, waveshapes, etc. The possibilities are limitless.

Other Capabilities Of CSound In our examples of additive and modulation synthesis we only used a limited number of functions and routines provided us by CSound, such as Osc (oscillator), Line and LineSig (line and line segment signal generators) etc. This tutorial intends to briefly explain the functionality of some of the other features of CSound. Remember that the CSound manual should be the ultimate reference when it comes to using these functions. Let us start with the two functions `buzz` and `genBuzz`. These functions will produce a set of harmonically related cosines. Thus they really implement simple additive synthesis, except that the number of partials can be varied dynamically through the duration of the note, rather than staying fixed as in simple additive synthesis. As an example, let us perform the tune defined at the very beginning of the tutorial using an instrument that will play each note by starting off with the fundamental and 70 harmonics, and ending with simply the sine wave fundamental (note that cosine and sine waves sound the same). We will use a straight line signal going from 70 to 0 over the duration of each note for the number of harmonics. The score used will be score1, and the orchestra will be:

```
oe14 :: SoundMap.InstrumentSigExp Mono
oe14 noteDur noteVel notePit =
   let numharms = lineCS CR 70 noteDur 0
       signal   = buzz pureToneTable numharms
                       (dbToAmp noteVel) (pchToHz notePit)
   in  Mono signal

o13 = (hdr, [instrAssoc1p0 oe14])

tut13 = example "tut13" score1 o13
```

Let's invert the line of the harmonics, and instead of going from 70 to 0, make it go from 0 to 70. This will produce an interesting effect quite different from the one just heard:

```
oe15 :: SoundMap.InstrumentSigExp Mono
oe15 noteDur noteVel notePit =
   let numharms = lineCS CR 0 noteDur 70
       signal   = buzz pureToneTable numharms
                       (dbToAmp noteVel) (pchToHz notePit)
   in  Mono signal

o14 = (hdr, [instrAssoc1p0 oe15])

tut14 = example "tut14" score1 o14
```

The `buzz` expression takes the overall amplitude, fundamental frequency, number of partials, and a sine wave table and generates a wave complex. In recent years there has been a lot of research conducted in the area of *physical modelling*. This technique attempts to approximate the sound of real world musical instruments through mathematical models. One of the most widespread, versatile and interesting of these models is the *Karplus-Strong algorithm* that simulates the sound of a plucked string. The algorithm starts off with a buffer containing a user-determined waveform. On every pass, the waveform is "smoothed out" and flattened by the algorithm to simulate the decay. There is a certain degree of randomness involved to make the string sound more natural. There are six different "smoothing methods" available in CSound, as mentioned in the CSound module. The `pluck` constructor accepts the note volume, pitch, the table number that is used to initialize the buffer, the smoothing method used, and two parameters that depend on the smoothing method. If zero is given as the initializing table number, the buffer starts off containing a random waveform (white noise). This is the best table when simulating a string instrument because of the randomness and percussive attack it produces when used with this algorithm, but you should experiment with other waveforms as well. Here is an example of what Pluck sounds like with a white noise buffer and the simple smoothing method. This method ignores the parameters, which we set to zero.

```
oe16 :: SoundMap.InstrumentSigExp Mono
oe16 _noteDur noteVel notePit =
   let signal = pluck 0 (pchToHz notePit)
                      PluckSimpleSmooth
                      (dbToAmp noteVel) (pchToHz notePit)
   in  Mono signal

o15 = (hdr, [instrAssoc1p0 oe16])

tut15 = example "tut15" score1 o15
```

The second smoothing method is the *stretched smooth*, which works like the simple smooth above, except that the smoothing process is stretched by a factor determined by the first parameter. The second parameter is ignored. The third smoothing method is the *snare drum* method. The first parameter is the "roughness" parameter, with 0 resulting in a sound identical to simple smooth, 0.5 being the perfect snare drum, and 1.0 being the same as simple smooth again with reversed polarity (like a graph flipped around the x-axis). The fourth smoothing method is the *stretched drum* method which combines the roughness and stretch factors – the first parameter is the roughness, the second is the stretch. The fifth method is *weighted average* – it combines the current sample (ie. the current pass through the buffer) with the previous one, with their weights being determined by the parameters. This is a way to add slight reverb to the plucked sound. Finally, the

last method filters the sound so it doesn't sound as bright. The parameters are ignored. You can modify the instrument `oe16` easily to listen to all these effects by simply replacing the variable `simpleSmooth` by `stretchSmooth, simpleDrum, stretchDrum, weightedSmooth` or `filterSmooth`. Here is another simple instrument example. This combines a snare drum sound with a stretched plucked string sound. The snare drum as a constant amplitude, while we apply an amplitude envelope to the string sound. The envelope is a spline curve with a hump in the middle, so both the attack and decay are gradual. The drum roughness factor is 0.3, so a pitch is still discernible (with a factor of 0.5 we would get a snare drum sound with no pitch, just a puff of white noise). The drum sound is shifted towards the left channel, while the string sound is shifted towards the right.

```
midHumpTN :: Score.Table
midHumpTN = 8
midHump :: Score.Statement
midHump = Score.Table midHumpTN 0 8192 True
             (cubicSpline 0.0 [(4096, 1.0), (4096, 0.0)])

score8 orc = pureTone : midHump : scored orc attrToInstr1p0 tune1

oe17 :: SoundMap.InstrumentSigExp Stereo
oe17 noteDur noteVel notePit =
   let string = pluck 0 (pchToHz notePit)
                        (PluckStretchSmooth 1.5)
                        (dbToAmp noteVel) (pchToHz notePit)
       drum   = pluck 0 (pchToHz notePit)
                        (PluckSimpleDrum 0.3)
                        6000 (pchToHz notePit)
       ampEnv = osc CR (tableNumber midHumpTN) 1.0 (1 / noteDur)
       left   = (0.65 * drum) + (0.35 * ampEnv * string)
       right  = (0.35 * drum) + (0.65 * ampEnv * string)
   in  Stereo left right

o16 = (hdr, [instrAssoc1p0 oe17])

tut16 = example "tut16" score8 o16
```

Let us now turn our attention to the effects we can achieve using a *delay line*. Let's define a simple percussive instrument. It's strong attack let us easily perceive the reverberation.

```
ping :: SigExp -> SigExp -> SigExp
ping noteVel notePit =
   let ampEnv = expon CR 1.0 1.0 (1/100)
   in  osc AR manySinesTable
           (ampEnv * dbToAmp noteVel) (pchToHz notePit)
```

There is still the problem, that subsequent notes truncate preceding ones. This would suppress the reverb. In order to avoid this we add a *legato* effect to the music. That is we prolong the notes such that they overlap.

```
score9 orc = manySines : scored orc attrToInstr1p0 (Music.legato 1 tune1)
```

Here we take the ping sound and add a little echo to it using delay:

```
oe18 :: SoundMap.InstrumentSigExp Stereo
oe18 _noteDur noteVel notePit =
```

```
   let ping'  = ping noteVel notePit
       dping1 = Orchestra.delay 0.05 ping'
       dping2 = Orchestra.delay 0.1  ping'
       left   = (0.65 * ping') + (0.35 * dping2) + (0.5 * dping1)
       right  = (0.35 * ping') + (0.65 * dping2) + (0.5 * dping1)
   in  Stereo left right

o17 = (hdr, [instrAssoc1p0 oe18])

tut17 = example "tut17" score9 o17
```

The constructor `delay` establishes a *delay line*. A delay line is essentially a buffer that contains the signal to be delayed. The first argument to the `delay` constructor is the length of the delay (which determines the size of the buffer), and the second argument is the signal to be delayed. So for example, if the delay time is 1.0 seconds, and the sampling rate is 44,100 Hz (CD quality), then the delay line will be a buffer containing 44,100 samples of the delayed signal. The buffer is rewritten at the audio rate. Once `Delay t sig` writes t seconds of the signal `sig` into the buffer, the buffer can be *tapped* using the `delTap` or the `delTapI` constructors. `delTap t dline` will extract the signal from `dline` at time `t` seconds. In the exmaple above, we set up a delay line containing 0.1 seconds of the audio signal, then we tapped it twice – once at 0.05 seconds and once at 0.1 seconds. The output signal is a combination of the original signal (left channel), the signal delayed by 0.05 seconds (middle), and the signal delayed by 0.1 seconds (right channel). CSound provides other ways to reverberate a signal besides the delay line just demonstrated. One such way is achieved via the Reverb constructor introduced in the module `CSound.Orchestra` module. This constructor tries to emulate natural room reverb, and takes as arguments the signal to be reverberated, and the reverb time in seconds. This is the time it takes the signal to decay to 1/1000 its original amplitude. In this example we output both the original and the reverberated sound.

```
oe19 :: SoundMap.InstrumentSigExp Stereo
oe19 _noteDur noteVel notePit =
   let ping'  = ping noteVel notePit
       rev    = reverb 0.3 ping'
       left   = (0.65 * ping') + (0.35 * rev)
       right  = (0.35 * ping') + (0.65 * rev)
   in  Stereo left right

o18 = (hdr, [instrAssoc1p0 oe19])

tut18 = example "tut18" score9 o18
```

The other two reverb functions are `comb` and `alpass`. Each of these requires as arguments the signal to be reverberated, the reverb time as above, and echo loop density in seconds. Here is an example of an instrument using `comb`.

```
oe20 :: SoundMap.InstrumentSigExp Mono
oe20 _noteDur noteVel notePit =
   Mono (comb 0.22 4.0 (ping noteVel notePit))

o19 = (hdr, [instrAssoc1p0 oe20])

tut19 = example "tut19" score9 o19
```

Delay lines can be used for effects other than simple echo and reverberation. Once the delay line has been established, it can be tapped at times that vary at control or audio rates. This can be taken advantage of to produce effects like chorus, flanger, or the Doppler effect. Here is an example of the flanger effect. This instrument adds a slight flange to `oe11`.

```
oe21 :: SigExp -> SigExp -> SigExp -> SigExp -> SoundMap.InstrumentSigExp Stereo
oe21 modFreqRatio modIndEnd panStart panEnd noteDur noteVel notePit =
   let carFreq = pchToHz notePit
       ampEnv  = oscCoolEnv 1.0 (1/noteDur)
       carAmp  = dbToAmp noteVel * ampEnv
       modFreq = carFreq * modFreqRatio
       modInd  = lineCS CR 0 noteDur modIndEnd
       modAmp  = modFreq * modInd
       modSig  = oscPure modAmp modFreq
       carrier = oscPure carAmp (carFreq + modSig)
       ftime   = oscPure (1/10) 2
       flanger = ampEnv * vdelay 1 (0.5 + ftime) carrier
       signal  = carrier + flanger
       pan     = lineCS CR panStart noteDur panEnd
       left    = pan * signal
       right   = (1 - pan) * signal
   in  Stereo left right

o20 = (hdr, [instrAssoc1p4 oe21])

tut20 = example "tut20" score7 o20
```

The last two examples use generic delay lines. That is we do not rely on special echo effects but build our own ones by delaying a signal, filtering it by low pass or high pass filters and feeding the result back to the delay function.

```
lowPass, highPass :: EvalRate -> SigExp -> SigExp -> SigExp
lowPass  rate cutOff sig = sigGen "tone"  rate 1 [sig, cutOff]
highPass rate cutOff sig = sigGen "atone" rate 1 [sig, cutOff]

oe22 :: SoundMap.InstrumentSigExp Stereo
oe22 _noteDur noteVel notePit =
   let ping' = ping noteVel notePit
       left  = rec (\x -> ping' + lowPass  AR  500 (Orchestra.delay 0.311 x))
       right = rec (\x -> ping' + highPass AR 1000 (Orchestra.delay 0.271 x))
   in  Stereo left right

o21 = (hdr, [instrAssoc1p0 oe22])

tut21 = example "tut21" score9 o21

oe23 :: SoundMap.InstrumentSigExp Mono
oe23 _noteDur noteVel notePit =
   let ping' = ping noteVel notePit
       rev = rec (\x -> ping' +
                    0.7 * (lowPass  AR  500 (Orchestra.delay 0.311 x)
                         + highPass AR 1000 (Orchestra.delay 0.271 x)))
   in  Mono rev

o22 = (hdr, [instrAssoc1p0 oe23])
```

```
tut22 = example "tut22" score9 o22
```

This completes our discussion of sound synthesis and Csound. For more information, please consult the CSound manual or check out

http://mitpress.mit.edu/e-books/csound/frontpage.html

The function `applyOutFunc` applies sound expression function to the expressions which represent the parameter fields from 6 on. These are the fields where the additional instrument parameters are put by `CSound.Score.statementToWords`.

```
test :: Output out => (Name, Score.T, TutOrchestra out) -> IO ExitCode
test = play csoundDir

toOrchestra :: Output out => TutOrchestra out -> Orchestra.T out
toOrchestra (hd, instrs) =
   Orchestra.Cons hd (SoundMap.instrumentTableToInstrBlocks instrs)

play :: Output out =>
   FilePath -> (Name, Score.T, TutOrchestra out) -> IO ExitCode
play dir (name, s, o') =
   let scorename = name ++ ".sco"
       orchname  = name ++ ".orc"
--     wavename  = name ++ ".wav"
       o = toOrchestra o'
--     (Orchestra.Cons (rate, _) _) = o
   in  do writeFile (dir++"/"++scorename) (Score.toString s)
          writeFile (dir++"/"++orchname)  (Orchestra.toString o)
{-
          system ("cd "++dir++" ; csound -d -W -o "
                    ++ wavename ++ " " ++ orchname ++ " " ++ scorename
                    ++ " ; play " ++ wavename)
-}
          system ("cd "++dir++" ; csound -d -A -o stdout -s "
                    ++ orchname ++ " " ++ scorename
                    ++ " | play -t aiff -")
{-
          system ("cd "++dir++" ; csound -d -o stdout -s "
                    ++ orchname ++ " " ++ scorename
                    ++ " | play -r " ++ show rate ++ " -t sw -")
-}
{-
          system ("cd "++dir++" ; csound -d -o dac "  -- /dev/dsp makes some chaotic noise
                    ++ orchname ++ " " ++ scorename)
-}
{-
          system (dir ++ "/csound.exe -W -o " ++ wavename
                    ++ " " ++ orchname ++ " " ++ scorename)
-}
```

Here are some bonus instruments for your pleasure and enjoyment. The first ten instruments are lifted from

http://wings.buffalo.edu/academic/department/AandL/music/pub/accci/01/01_01_1b.txt.html

The tutorial explains how to add echo/reverb and other effects to the instruments if you need to. This instrument sounds like an electric piano and is really simple – `pianoEnv` sets the amplitude envelope, and the sound waveform is just a series of 10 harmonics. To make the sound brighter, increase the weight of the upper harmonics.

```
piano, reedy, flute
   :: (Name, Score.T, TutOrchestra Mono)

pianoOrc, reedyOrc, fluteOrc
   :: TutOrchestra Mono

pianoScore, reedyScore, fluteScore :: TutOrchestra out -> Score.T
pianoEnv, reedyEnv, fluteEnv,
  pianoWave, reedyWave, fluteWave :: Score.Statement
pianoEnvTN, reedyEnvTN, fluteEnvTN,
  pianoWaveTN, reedyWaveTN, fluteWaveTN :: Score.Table
pianoEnvTable, reedyEnvTable, fluteEnvTable,
  pianoWaveTable, reedyWaveTable, fluteWaveTable :: SigExp

pianoEnvTN  = 10; pianoEnvTable  = tableNumber pianoEnvTN
pianoWaveTN = 11; pianoWaveTable = tableNumber pianoWaveTN

pianoEnv    = Score.Table pianoEnvTN 0 1024 True (lineSeg1 0 [(20, 0.99),
                                    (380, 0.4), (400, 0.2), (224, 0)])
pianoWave   = Score.Table pianoWaveTN  0 1024 True (compSine1 [0.158, 0.316,
                     1.0, 1.0, 0.282, 0.112, 0.063, 0.079, 0.126, 0.071])

pianoScore orc = pianoEnv : pianoWave : scored orc attrToInstr1p0 tune1

pianoOE :: SoundMap.InstrumentSigExp Mono
pianoOE noteDur noteVel notePit =
   let ampEnv = osc CR pianoEnvTable (dbToAmp noteVel) (1/noteDur)
       signal = osc AR pianoWaveTable ampEnv (pchToHz notePit)
   in  Mono signal

pianoOrc = (hdr, [instrAssoc1p0 pianoOE])

piano = example "piano" pianoScore pianoOrc
```

Here is another instrument with a reedy sound to it

```
reedyEnvTN  = 12; reedyEnvTable  = tableNumber reedyEnvTN
reedyWaveTN = 13; reedyWaveTable = tableNumber reedyWaveTN

reedyEnv    = Score.Table reedyEnvTN   0 1024 True (lineSeg1 0 [(172, 1.0),
     (170, 0.8), (170, 0.6), (170, 0.7), (170, 0.6), (172,0)])
reedyWave   = Score.Table reedyWaveTN  0 1024 True (compSine1 [0.4, 0.3,
                     0.35, 0.5, 0.1, 0.2, 0.15, 0.0, 0.02, 0.05, 0.03])

reedyScore orc = reedyEnv : reedyWave : scored orc attrToInstr1p0 tune1

reedyOE :: SoundMap.InstrumentSigExp Mono
```

```
reedyOE noteDur noteVel notePit =
   let ampEnv = osc CR reedyEnvTable (dbToAmp noteVel) (1/noteDur)
       signal = osc AR reedyWaveTable ampEnv (pchToHz notePit)
   in  Mono signal

reedyOrc = (hdr, [instrAssoc1p0 reedyOE])

reedy = example "reedy" reedyScore reedyOrc
```

We can use a little trick to make it sound like several reeds playing by adding three signals that are slightly out of tune:

```
reedy2OE :: SoundMap.InstrumentSigExp Stereo
reedy2OE noteDur noteVel notePit =
   let ampEnv = osc CR reedyEnvTable (dbToAmp noteVel) (1/noteDur)
       freq   = pchToHz notePit
       reedyOsc = osc AR reedyWaveTable
       a1     = reedyOsc ampEnv freq
       a2     = reedyOsc (ampEnv * 0.44) (freq + (0.023 * freq))
       a3     = reedyOsc (ampEnv * 0.26) (freq + (0.019 * freq))
       left   = (a1 * 0.5) + (a2 * 0.35) + (a3 * 0.65)
       right  = (a1 * 0.5) + (a2 * 0.65) + (a3 * 0.35)
   in  Stereo left right

reedy2Orc :: TutOrchestra Stereo
reedy2Orc = (hdr, [instrAssoc1p0 reedy2OE])

reedy2 :: (Name, Score.T, TutOrchestra Stereo)
reedy2 = example "reedy2" reedyScore reedy2Orc
```

This instrument tries to emulate a flute sound by introducing random variations to the amplitude envelope. The score file passes in two parameters – the first one is the depth of the random tremolo in percent of total amplitude. The tremolo is implemented using the `randomI` function, which generates a signal that interpolates between 2 random numbers over a certain number of samples that is specified by the second parameter.

```
fluteTune :: TutMelody Pair

fluteTune = Music.line
               (map ($ TutAttr 1.6 (30, 40))
                  [c 1 hn, e 1 hn, g 1 hn, c 2 hn,
                   a 1 hn, c 2 qn, a 1 qn, g 1 dhn]
                ++ [qnr])


fluteEnvTN  = 14; fluteEnvTable  = tableNumber fluteEnvTN
fluteWaveTN = 15; fluteWaveTable = tableNumber fluteWaveTN

fluteEnv    = Score.Table fluteEnvTN   0 1024 True (lineSeg1 0 [(100, 0.8),
                        (200, 0.9), (100, 0.7), (300, 0.2), (324, 0.0)])
fluteWave   = Score.Table fluteWaveTN  0 1024 True (compSine1 [1.0, 0.4,
                                                  0.2, 0.1, 0.1, 0.05])

fluteScore orc = fluteEnv : fluteWave : scored orc attrToInstr1p2 fluteTune
```

```
fluteOE :: SigExp -> SigExp -> SoundMap.InstrumentSigExp Mono
fluteOE depth numSam noteDur noteVel notePit =
   let vol    = dbToAmp noteVel
       rand   = randomI AR numSam (vol/100 * depth)
       ampEnv = oscI AR fluteEnvTable
                     (rand + vol) (1 / noteDur)
       signal = oscI AR fluteWaveTable
                     ampEnv (pchToHz notePit)
   in  Mono signal

fluteOrc = (hdr, [instrAssoc1p2 fluteOE])

flute = example "flute" fluteScore fluteOrc
```

Dirty hacks are going on here in order to pass the Phoneme values through all functions.

```
voice' :: SigExp -> SigExp -> SigExp -> SigExp ->
             SigExp -> SigExp -> SigExp -> SigExp -> SigExp
voice' vibWave wave gain vibAmp vibFreq amp freq phoneme =
   sigGen "voice" AR 1
      [amp, freq, phoneme, gain, vibFreq, vibAmp, wave, vibWave]

data Phoneme =
      Eee | Ihh | Ehh | Aaa |
      Ahh | Aww | Ohh | Uhh |
      Uuu | Ooo | Rrr | Lll |
      Mmm | Nnn | Nng | Ngg |
      Fff | Sss | Thh | Shh |
      Xxx | Hee | Hoo | Hah |
      Bbb | Ddd | Jjj | Ggg |
      Vvv | Zzz | Thz | Zhh
   deriving (Show, Eq, Ord, Enum)

voiceTune :: TutMelody Pair
voiceTune = Music.line
               (map (\(n,ph) ->
                         n (TutAttr 1 (fromIntegral (fromEnum ph), 2)))
                  [(c 1 hn, Aaa), (e 1 hn, Ehh), (g 1 hn, Ohh), (c 2 hn, Ehh),
                   (a 1 hn, Eee), (c 2 qn, Aww), (a 1 qn, Aww), (g 1 dhn, Aaa)]
                ++ [qnr])


voiceVibWaveTN,    voiceWaveTN    :: Score.Table
voiceVibWaveTable, voiceWaveTable :: SigExp
voiceVibWaveTN = 14; voiceVibWaveTable = tableNumber voiceVibWaveTN
voiceWaveTN    = 15; voiceWaveTable    = tableNumber voiceWaveTN

voiceWave, voiceVibWave :: Score.Statement
voiceWave    = Score.Table voiceWaveTN    0 1024 True
   (let width = 50
    in  lineSeg1 0 [(width, 1), (width, 0), (1024-2*width, 0)])
voiceVibWave = Score.Table voiceVibWaveTN 0 1024 True (compSine1 [1.0, 0.4])

voiceScore :: TutOrchestra out -> Score.T
voiceScore orc =
```

```
   voiceVibWave : voiceWave : scored orc attrToInstr1p2 voiceTune

voiceOE :: SigExp -> SigExp -> SoundMap.InstrumentSigExp Mono
voiceOE phoneme gain _noteDur noteVel notePit =
   let vol    = dbToAmp noteVel
       signal = voice' voiceVibWaveTable voiceWaveTable
                   gain (3/100) 5 vol (pchToHz notePit) phoneme
   in  Mono signal

voiceOrc :: TutOrchestra Mono
voiceOrc = (hdr, [instrAssoc1p2 voiceOE])

voice :: (Name, Score.T, TutOrchestra Mono)
voice = example "voice" voiceScore voiceOrc
```

4.4 MML

```
module Haskore.Interface.MML where

import qualified Haskore.Basic.Pitch    as Pitch
import qualified Haskore.Music          as Music
import qualified Haskore.Melody         as Melody
import           Haskore.Basic.Duration((%+))

import Control.Monad.Trans.State (State, state, evalState, )
```

I found some music notated in a language called MML. The description consists of strings.

- `l`n determines the duration of subsequent notes: `l1` - whole note, `l2` - half note, `l4` - quarter note and so on.
- > switch to the octave above
- < switch to the octave below
- Lower case letter `a` - `g` play the note of the corresponding pitch class.
- # (sharp) or − (flat) may follow a note name in order to increase or decrease, respectively, the pitch of the note by a semitone.
- An additional figure for the note duration may follow.
- `p` is pause.

See module `Kantate147` for an example.

```
type Accum = (Music.Dur, Pitch.Octave)

barToMusic :: String -> Accum -> ([Melody.T ()], Accum)
barToMusic []     accum      = ([], accum)
barToMusic (c:cs) (dur, oct) =
```

```
   let charToDur dc = 1 %+ read (dc:[])
       prependAtom atom adur (ms, newAccum) =
          (atom adur : ms, newAccum)
       procNote ndur pitch c0s =
          let mkNote c1s = prependAtom (flip (Melody.note (oct, pitch)) ())
                                       ndur (barToMusic c1s (dur, oct))
          in  case c0s of
                '#':c1s -> procNote ndur (succ pitch) c1s
                '-':c1s -> procNote ndur (pred pitch) c1s
                c1 :c1s -> if '0'<=c1 && c1<='9'
                           then procNote (charToDur c1) pitch c1s
                           else mkNote c0s
                []      -> mkNote c0s
   in  case c of
         'c' -> procNote dur Pitch.C cs
         'd' -> procNote dur Pitch.D cs
         'e' -> procNote dur Pitch.E cs
         'f' -> procNote dur Pitch.F cs
         'g' -> procNote dur Pitch.G cs
         'a' -> procNote dur Pitch.A cs
         'b' -> procNote dur Pitch.B cs
         'p' -> let (c1:c1s) = cs
                in  prependAtom Music.rest (charToDur c1)
                                (barToMusic c1s (dur, oct))
         '<' -> barToMusic cs (dur, oct-1)
         '>' -> barToMusic cs (dur, oct+1)
         'l' -> let (c1:c1s) = cs
                in  barToMusic c1s (charToDur c1, oct)
         _   -> error ("unexpected character '"++[c]++"' in Haskore.Interface.MML description")

toMusicState :: String -> State Accum [Melody.T ()]
toMusicState s = state (barToMusic s)

toMusic :: Pitch.Octave -> String -> Melody.T ()
toMusic oct s = Music.line (evalState (toMusicState s) (0, oct))
```

5 Processing and Analysis

5.1 Optimization

This module provides functions that simplify the structure of a `Music.T` according to the rules proven in Section 3.2.1

```
module Haskore.Process.Optimization where

import qualified Medium.Controlled.List as CtrlMediumList
import qualified Medium.Controlled      as CtrlMedium
import qualified Haskore.Music as Music
import Medium.Controlled.List (serial, parallel, )
import Data.List.HT (partitionMaybe, )
import Data.Maybe.HT (toMaybe, )
import Data.Maybe (catMaybes, fromMaybe, )
```

`Music.T` objects that come out of `ReadMidi.toMusic` almost always contain redundancies, like rests of zero duration and redundant instrument specifications. The function `Optimization.all` reduces the redundancy to make a `Music.T` file less cluttered and more efficient to use.

```
all, rest, composition, duration, tempo, transpose, volume ::
   Music.T note -> Music.T note
all = tempo . transpose . volume . singleton . composition . rest
```

Remove rests of zero duration.

```
rest = Music.mapList
   (,)
   (flip const)
   (filter (not . isZeroRest))
   (filter (not . isZeroRest))

isZeroRest :: Music.T note -> Bool
isZeroRest =
   Music.switchList
      (\d at -> d==0 && maybe True (const False) at)
      (const (const False))
      (const False)
      (const False)
```

Remove empty parallel and serial compositions and controllers of empty music.

```
composition = fromMaybe (Music.rest 0) . Music.foldList
   (\d -> Just . Music.atom d)
   (fmap . Music.control)
   ((\ms -> toMaybe (not (null ms)) (serial   ms)) . catMaybes)
   ((\ms -> toMaybe (not (null ms)) (parallel ms)) . catMaybes)
```

Remove any atom of zero duration. This is not really an optimization but a hack to get rid of MIDI NoteOn and NoteOff events at the same time point.

```
duration = fromMaybe (Music.rest 0) . Music.foldList
   (\d -> toMaybe (d /= 0) . Music.atom d)
   (fmap . Music.control)
   (Just . serial   . catMaybes)
   (Just . parallel . catMaybes)
```

The control structures for tempo, transposition and change of instruments can be handled very similar using the following routines. The function `mergeControl'` checks if nested controllers are of the same kind. If they are then they are merged into one. The function would be much simpler if it would be implemented for specific constructors, but we want to stay independent from the particular data structure, which is already quite complex.

```
mergeControl' ::
      (Music.Control -> Maybe a)
   -> (a -> Music.T note -> Music.T note)
   -> (a -> a -> a)
   -> Music.T note
   -> Music.T note
```

```
mergeControl' extract control merge =
   let fcSub c m = fmap (flip (,) m) (extract c)
       fc' c0 m0 x0 =
                maybe (Music.control c0 m0)
                      (\(x1,m1) -> control (merge x0 x1) m1)
                      (Music.switchList (const (const Nothing))
                          fcSub (const Nothing) (const Nothing) m0)
       fc c m = maybe (Music.control c m)
                      (fc' c m)
                      (extract c)
   in  Music.foldList
          Music.atom fc Music.line Music.chord
```

The following function collects neighboured controllers into groups, extracts controllers of a specific type and prepends a controller to the list of neighboured controllers, which has the total effect of the extracted controllers. This change of ordering is always possible because in the current set of controllers two neighboured controllers of different type commutes. E.g. it is `transpose n . changeTempo r == changeTempo r . transpose n` and thus the following simplification `transpose 1 . changeTempo 2 . transpose 3 == transpose 4 . changeTempo 2` is possible.

```
mergeControl, mergeControlCompact ::
      (Music.Control -> Maybe a)
   -> (a -> Music.T note -> Music.T note)
   -> (a -> a -> a)
   -> Music.T note
   -> Music.T note
mergeControlCompact extract control merge =
   let collectControl =
          Music.switchList
             (\d n -> ([], Music.atom d n))
             (\c m -> let cm = collectControl m
                      in  (c : fst cm, snd cm))
             ((,) [] . Music.line  . map recourse)
             ((,) [] . Music.chord . map recourse)
       recourse m =
          let cm = collectControl m
              (xs, cs') = partitionMaybe extract (fst cm)
              x  = foldl1 merge xs
              collectedCtrl = if null xs then id else control x
          in  collectedCtrl (foldr id (snd cm) (map Music.control cs'))
   in  recourse

-- more intuitive implementation
mergeControl extract control merge =
--   flattenControllers .
--   CtrlMediumList.mapControl
   CtrlMedium.foldList
      CtrlMediumList.prim
      CtrlMediumList.serial
      CtrlMediumList.parallel
      (\cs cm ->
          let (xs, cs') = partitionMaybe extract cs
              collectedCtrl =
                 if null xs then id else control (foldl1 merge xs)
```

```
          in  collectedCtrl (foldr id cm (map Music.control cs'))) .
   cumulateControllers

cumulateControllers ::
      CtrlMediumList.T control a
   -> CtrlMediumList.T [control] a
cumulateControllers =
   CtrlMedium.foldList
      CtrlMediumList.prim
      CtrlMediumList.serial
      CtrlMediumList.parallel
      (\c m ->
         let cm = CtrlMedium.control [c] m
         in  CtrlMedium.switchList
                (const cm)
                (const cm)
                (const cm)
                (\cs m' -> CtrlMedium.control (c:cs) m')
                m)

flattenControllers ::
      CtrlMediumList.T [control] a
   -> CtrlMediumList.T control a
flattenControllers =
   CtrlMedium.foldList
      CtrlMediumList.prim
      CtrlMediumList.serial
      CtrlMediumList.parallel
      (flip (foldr id) . map CtrlMedium.control)
```

The function `removeNeutral` removes controllers that have no effect.

```
removeNeutral :: (Music.Control -> Bool) -> Music.T note -> Music.T note
removeNeutral isNeutral =
   let fc c m = if isNeutral c
                then m
                else Music.control c m
   in  Music.foldList Music.atom fc Music.line Music.chord
```

Remove redundant `Tempo`s.

```
tempo =
   let maybeTempo (Music.Tempo t) = Just t
       maybeTempo _               = Nothing
   in  removeNeutral (== Music.Tempo 1) .
          mergeControl maybeTempo Music.changeTempo (*)
```

Remove redundant `Transpose`s.

```
transpose =
   let maybeTranspose (Music.Transpose t) = Just t
       maybeTranspose _                   = Nothing
   in  removeNeutral (== Music.Transpose 0) .
          mergeControl maybeTranspose Music.transpose (+)
```

Change repeated Volume Note Attributes to Phrase Attributes.

```
volume =
   let maybeLoudness (Music.Phrase (Music.Dyn (Music.Loudness t))) = Just t
       maybeLoudness _ = Nothing
   in  removeNeutral (== Music.Phrase (Music.Dyn (Music.Loudness 1))) .
          mergeControl maybeLoudness Music.loudness1 (*)
```

Eliminate `Serial` and `Parallel` composition if they contain only one member. This can be done very general for `CtrlMedium.T`. We have also a version which works on `Music.T`. Since the medium data type supports controllers there is no longer a real difference between these two functions.

```
singletonMedium ::
   CtrlMediumList.T control a -> CtrlMediumList.T control a
singletonMedium =
   CtrlMedium.foldList CtrlMediumList.prim
      (\ms -> case ms of {[x] -> x; _ -> serial   ms})
      (\ms -> case ms of {[x] -> x; _ -> parallel ms})
      (CtrlMedium.control)

singleton :: Music.T note -> Music.T note
singleton =
   Music.foldList Music.atom Music.control
      (\ms -> case ms of {[x] -> x; _ -> Music.line  ms})
      (\ms -> case ms of {[x] -> x; _ -> Music.chord ms})
```

5.2 Structure Analysis

This module contains a function which builds a hierarchical music object from a serial one. This is achieved by searching for long common infixes. A common infix is replaced by a single object at each occurence.

This module proofs the sophistication of the separation between general arrangement of some objects as provided by the module `Medium` and the special needs of music provided by the module `Music`. It's possible to formulate these algorithms without the knowledge of Music and we can insert the type `Tag` to distinguish between media primitives and macro calls. The only drawback is that it is not possible to descend into controlled sub-structures, like Tempo and Trans.

```
module Medium.Controlled.ContextFreeGrammar
   (T, Tag(..), TagMedium, fromMedium, toMedium) where

import qualified Medium.Controlled.List as CtrlMediumList
import qualified Medium.Controlled      as CtrlMedium
import Medium.Plain.ContextFreeGrammar
   (Tag(..), joinTag, replaceInfix,
    whileM, smallestCycle, maximumCommonInfixMulti)
import Medium (prim, serial1, parallel1)

import Data.Maybe (fromJust)
import qualified Haskore.General.Map as Map

import Control.Monad.Trans.State (state, execState)
```

Condense all common infixes down to length 'thres'. The infixes are replaced by some marks using the constructor Left. They can be considered as macros or as non-terminals in a grammar. The normal primitives are preserved with constructor Right. We end up with a context-free grammar of the media.

```
type TagMedium key control prim = CtrlMediumList.T control (Tag key prim)

type T key control prim = [(key, TagMedium key control prim)]

fromMedium :: (Ord key, Ord control, Ord prim) =>
   [key] -> Int -> CtrlMediumList.T control prim -> T key control prim
fromMedium (key:keys) thres m =
   let action = whileM (>= thres) (map (state . condense) keys)
       -- action = sequence (take 1 (map (state . condense) keys))
   in  reverse $ execState action [(key, fmap Prim m)]
fromMedium _ _ _ =
   error ("No key given."++
      " Please provide an infinite or at least huge number of macro names.")
```

The inverse of `fromMedium`: Expand all macros. Cyclic macro references shouldn't be a problem if it is possible to resolve the dependencies. We manage the grammar in the dictionary `dict`. Now a naive way for expanding the macros is to recourse into each macro call manually using lookups to `dict`. This would imply that we need new memory for each expansion of the same macro. We have chosen a different approach: We map `dict` to a new dictionary `dict'` which contains the expanded versions of each Medium. For expansion we don't use repeated lookups to `dict` but we use only one lookup to `dict'` – which contains the fully expanded version of the considered Medium. This method is rather the same as if you write Haskell values that invokes each other.

The function `expand` computes the expansion for each key and the function `toMedium` computes the expansion of the first macro. Thus `toMedium` quite inverts `fromMedium`.

```
toMedium :: (Show key, Ord key, Ord prim) =>
   T key control prim -> CtrlMediumList.T control prim
toMedium = snd . head . expand

expand :: (Show key, Ord key, Ord prim) =>
   T key control prim -> [(key, CtrlMediumList.T control prim)]
expand grammar =
   let notFound key = error ("The non-terminal '" ++ show key ++ "' is unknown.")
       dict  = Map.fromList grammar
       dict' = Map.map (CtrlMedium.foldList expandSub serial1 parallel1
                           CtrlMedium.control) dict
       expandSub (Prim p) = prim p
       expandSub (Call key) =
          Map.findWithDefault dict' (notFound key) key
       expandSub (CallMulti n key) =
          serial1 (replicate n (Map.findWithDefault dict' (notFound key) key))
   in  map (fromJust . Map.lookup (Map.mapWithKey (,) dict') . fst) grammar
```

Find the longest common infix over all parts of the music and replace it in all of them.

```
condense :: (Ord key, Ord control, Ord prim) =>
      key
   -> T key control prim
```

```
   -> (Int, T key control prim)
condense key x =
   let getSerials = CtrlMedium.switchList
          (const [])
          (\xs -> xs : concatMap getSerials xs)
          (\xs ->      concatMap getSerials xs)
          (const getSerials)
       infx = smallestCycle (maximumCommonInfixMulti length
                  (concatMap (getSerials . snd) x))
       absorbSingleton _ [m] = m
       absorbSingleton collect ms = collect ms
       replaceRec = CtrlMedium.foldList prim
          (absorbSingleton serial1 . map joinTag . replaceInfix key infx)
          (absorbSingleton parallel1)
          (CtrlMedium.control)
   in  (length infx, (key, serial1 infx) : map (\(k, ms) -> (k, replaceRec ms)) x)
```

5.3 Markov Chains

Markov chains are now available in a package called `markov-chain`.

5.4 Pretty printing Music

This module aims at formatting (pretty printing) of musical objects with Haskell syntax. This is particularly useful for converting algorithmically generated music into Haskell code that can be edited and furtherly developed.

```
module Haskore.Process.Format where

import qualified Language.Haskell.Pretty as Pretty
import qualified Language.Haskell.Syntax as Syntax
import qualified Language.Haskell.Parser as Parser

import qualified Haskore.Basic.Duration  as Duration
import qualified Haskore.Music           as Music
import qualified Haskore.Melody          as Melody
import qualified Haskore.Melody.Standard as StdMelody
import qualified Medium.Controlled as CtrlMedium

import Medium.Controlled.ContextFreeGrammar as Grammar
import qualified Haskore.General.Map as Map
import qualified Data.Ratio as Ratio
import qualified Data.Char  as Char
import Data.List(intersperse)
```

Format a grammar as computed with the module `Medium.Controlled.ContextFreeGrammar`.

```
prettyGrammarMedium :: (Show prim, Show control) =>
   Grammar.T String control prim -> String
prettyGrammarMedium = prettyGrammar controlGen prim
```

```
prettyGrammarMelody ::
   Grammar.T String Music.Control (Music.Primitive StdMelody.Note) -> String
prettyGrammarMelody = prettyGrammar control primMelody

prettyGrammar ::
   (Int -> control -> (Int -> ShowS) -> ShowS) ->
   (Int -> prim -> ShowS) ->
   Grammar.T String control prim -> String
prettyGrammar controlSyntax primSyntax g =
   let text = unlines (map (flip id "" . bind controlSyntax primSyntax) g)
       Parser.ParseOk (Syntax.HsModule _ _ _ _ code) =
          Parser.parseModule text
   in  unlines (map Pretty.prettyPrint code) -- show code
```

Format a `Medium` object that contains references to other medium objects.

```
bind ::
   (Int -> control -> (Int -> ShowS) -> ShowS) ->
   (Int -> prim -> ShowS) ->
   (String, Grammar.TagMedium String control prim) -> ShowS
bind controlSyntax primSyntax (key, ms) =
   showString key . showString " = " . tagMedium 0 controlSyntax primSyntax ms

tagMedium ::
   Int ->
   (Int -> control -> (Int -> ShowS) -> ShowS) ->
   (Int -> prim -> ShowS) ->
   Grammar.TagMedium String control prim -> ShowS
tagMedium prec controlSyntax primSyntax m =
   let primSyntax' _     (Grammar.Call s) = showString s
       primSyntax' prec' (Grammar.CallMulti n s) =
          enclose prec' 0
             (showString "serial $ replicate " . showsPrec 10 n .
              showString " " . showString s)
       primSyntax' prec' (Grammar.Prim p) = primSyntax prec' p
   in  CtrlMedium.foldList
          (flip primSyntax')
          (listFunc "serial")
          (listFunc "parallel")
          (flip . flip controlSyntax)
          m prec

list :: [Int -> ShowS] -> ShowS
list = foldr (.) (showString "]") . (showString "[" :) .
          intersperse (showString ",") . map (flip id 0)

listFunc :: String -> [Int -> ShowS] -> Int -> ShowS
listFunc func ps prec =
   enclose prec 10 (showString func . showString " " . list ps)

prim :: (Show p) => Int -> p -> ShowS
prim prec p = enclose prec 10 (showString "prim " . showsPrec 10 p)

dummySrcLoc :: Syntax.SrcLoc
dummySrcLoc = Syntax.SrcLoc {Syntax.srcFilename = "",
                             Syntax.srcLine = 0,
```

```
                                Syntax.srcColumn = 0}
```

Of course we also want to format plain music, that is music without tags.

```
prettyMelody :: StdMelody.T -> String
prettyMelody m = prettyExp (melody 0 m "")

prettyExp :: String -> String
prettyExp text =
   let Parser.ParseOk (Syntax.HsModule _ _ _ _
          [Syntax.HsPatBind _ _ (Syntax.HsUnGuardedRhs code) _]) =
             Parser.parseModule ("dummy = "++text)
   in  Pretty.prettyPrint code
```

Now we go to define functions that handle the particular primitives of music. Note that `Control` information and `NoteAttribute`s are printed as atoms.

```
melody :: Int -> StdMelody.T -> ShowS
melody prec m =
   Music.foldList
      (flip . flip atom)
      (flip . flip control)
      (listFunc "line")
      (listFunc "chord")
      m prec

primMelody :: Int -> Music.Primitive StdMelody.Note -> ShowS
primMelody prec (Music.Atom d at) = atom prec d at

atom :: Show attr =>
   Int -> Duration.T -> Music.Atom (Melody.Note attr) -> ShowS
atom prec d = maybe (rest prec d) (note prec d)

note :: Show attr =>
   Int -> Duration.T -> Melody.Note attr -> ShowS
note prec d (Melody.Note nas (o,pc)) =
   enclose prec 10 (showString (map Char.toLower (show pc)) .
      showString " " . showsPrec 10 o .
      showString " " . durSyntax id "n" d .
      showString " " . showsPrec 10 nas)

rest :: Int -> Duration.T -> ShowS
rest prec d =
   durSyntax (\dStr -> enclose prec 10 (showString "rest " . dStr)) "nr" d

controlGen :: (Show control) => Int -> control -> (Int -> ShowS) -> ShowS
controlGen prec c m =
   enclose prec 10
     (showString "control " . showsPrec 10 c .
      showString " " . m 10)

control :: Int -> Music.Control -> (Int -> ShowS) -> ShowS
control prec c m =
   let controlSyntax name arg =
          enclose prec 10
```

```
            (showString name . showString " " . arg . showString " " . m 10)
  in  case c of
        Music.Tempo d     -> controlSyntax "changeTempo" (showDur   10 d)
        Music.Transpose p -> controlSyntax "transpose"   (showsPrec 10 p)
        Music.Player p    -> controlSyntax "setPlayer"   (showsPrec 10 p)
        Music.Phrase p    -> controlSyntax "phrase"      (showsPrec 10 p)
```

Note that the call to `show` can't be moved from the `controlSyntax` calls in `control` to `controlSyntax` because that provokes a compiler problem, namely

```
Mismatched contexts
When matching the contexts of the signatures for
  controlSyntax :: forall a.
                   (Show a) =>
                   String -> a -> StdMelody.T -> Language.Haskell.Syntax.HsExp
  control :: Music.Primitive -> Language.Haskell.Syntax.HsExp
The signature contexts in a mutually recursive group should all be identical
When generalising the type(s) for controlSyntax, control
```

```
durSyntax :: (ShowS -> ShowS) -> String -> Duration.T -> ShowS
durSyntax showRatio suffix d =
   maybe
      (showRatio (showDur 10 d))
      (\s -> showString (s++suffix))
      (Map.lookup Duration.nameDictionary d)

showDur :: Int -> Duration.T -> ShowS
showDur prec =
   (\d -> enclose prec 7
        (shows (Ratio.numerator d) .
         showString "%+" .
         shows (Ratio.denominator d))) .
   Duration.toRatio
```

Enclose an expression in parentheses if the inner operator has at most the precedence of the outer operator.

```
enclose :: Int -> Int -> ShowS -> ShowS
enclose outerPrec innerPrec = showParen (outerPrec >= innerPrec)
```

6 Related and Future Research

Many proposals have been put forth for programming languages targeted for computer music composition [Dan89, Sch83, Col84, AK92, DFV92, HS92, CR84, OFLB94], so many in fact that it would be difficult to describe them all here. None of them (perhaps surprisingly) are based on a *pure* functional language, with one exception: the recent work done by Orlarey et al. at GRAME [OFLB94], which uses a pure lambda calculus approach to music description, and bears some resemblance to our effort. There are some other related approaches based on variants of Lisp, most notably Dannenberg's *Fugue* language [DFV92], in which operators similar to ours can be found but where the emphasis is more on instrument synthesis rather

than note-oriented composition. Fugue also highlights the utility of lazy evaluation in certain contexts, but extra effort is needed to make this work in Lisp, whereas in a non-strict language such as Haskell it essentially comes "for free". Other efforts based on Lisp utilize Lisp primarily as a convenient vehicle for "embedded language design," and the applicative nature of Lisp is not exploited well (for example, in Common Music the user will find a large number of macros which are difficult if not impossible to use in a functional style).

We are not aware of any computer music language that has been shown to exhibit the kinds of algebraic properties that we have demonstrated for Haskore. Indeed, none of the languages that we have investigated make a useful distinction between music and performance, a property that we find especially attractive about the Haskore design. On the other hand, Balaban describes an abstract notion (apparently not yet a programming language) of "music structure," and provides various operators that look similar to ours [Bal92]. In addition, she describes an operation called *flatten* that resembles our literal interpretation `perform`. It would be interesting to translate her ideas into Haskell; the match would likely be good.

Perhaps surprisingly, the work that we find most closely related to ours is not about music at all: it is Henderson's *functional geometry*, a functional language approach to generating computer graphics [Hen82]. There we find a structure that is in spirit very similar to ours: most importantly, a clear distinction between object *description* and *interpretation* (which in this paper we have been calling musical objects and their performance). A similar structure can be found in Arya's *functional animation* work [Ary94].

There are many interesting avenues to pursue with this research. On the theoretical side, we need a deeper investigation of the algebraic structure of music, and would like to express certain modern theories of music in Haskore. The possibility of expressing other scale types instead of the thus far unstated assumption of standard equal temperament scales is another area of investigation. On the practical side, the potential of a graphical interface to Haskore is appealing. We are also interested in extending the methodology to sound synthesis. Our primary goal currently, however, is to continue using Haskore as a vehicle for interesting algorithmic composition (for example, see [HB95]).

A Helper modules

A.1 Convenient Functions for Getting Started With Haskore and MIDI

```
module Haskore.Interface.MIDI.Render where

import qualified Haskore.Interface.MIDI.Write         as WriteMidi
import qualified Haskore.Interface.MIDI.InstrumentMap as InstrMap
import qualified Sound.MIDI.General          as GeneralMidi

import qualified Sound.MIDI.File.Save        as SaveMidi
import qualified Sound.MIDI.File             as MidiFile
import qualified Sound.MIDI.Message.Channel  as ChannelMsg

import qualified Haskore.Music.GeneralMIDI   as MidiMusic
import qualified Haskore.Music.Rhythmic      as RhyMusic
import qualified Haskore.Music               as Music
import qualified Haskore.Melody              as Melody
import qualified Haskore.Performance.Context as Context
import qualified Haskore.Performance.Fancy   as FancyPerformance

import qualified Numeric.NonNegative.Class   as NonNeg
import qualified Numeric.NonNegative.Wrapper as NonNegW

import System.Cmd (rawSystem, )
import System.Exit (ExitCode, )
```

Given a `Player.Map`, `Context.T`, `InstrMap.T`, and file name, we can write a `MidiMusic.T` value into a midi file:

```
fileFromRhythmicMusic ::
   (Ord instr, Ord drum, NonNeg.C time, RealFrac time, RealFrac dyn) =>
   FilePath ->
      (InstrMap.ChannelProgramPitchTable drum,
       InstrMap.ChannelProgramTable instr,
       Context.T time dyn (RhyMusic.Note drum instr),
       RhyMusic.T drum instr) -> IO ()
fileFromRhythmicMusic fn m =
   SaveMidi.toFile fn (WriteMidi.fromRhythmicMusic m)
```

A.1.1 Test routines

Using the defaults above, from a `MidiMusic.T` object, we can:

1. generate a `Performance.T` using `Haskore.Performance.Default.fancyFromMusic`
2. generate a `MidiFile.T` data structure

```
midi :: MidiMusic.T -> MidiFile.T
midi =
   WriteMidi.fromRhythmicPerformance [] InstrMap.defltGM .
```

```
   FancyPerformance.floatFromMusic

generalMidi :: MidiMusic.T -> MidiFile.T
generalMidi =
   WriteMidi.fromGMPerformanceAuto .
   FancyPerformance.floatFromMusic

generalMidiDeflt :: MidiMusic.T -> MidiFile.T
generalMidiDeflt =
   WriteMidi.fromGMPerformance (InstrMap.lookup InstrMap.defltCMap) .
   FancyPerformance.floatFromMusic

mixedMidi :: MidiMusic.T -> MidiFile.T
mixedMidi =
   WriteMidi.fromRhythmicPerformanceMixed [] InstrMap.defltGM .
   FancyPerformance.floatFromMusic

mixedGeneralMidi :: MidiMusic.T -> MidiFile.T
mixedGeneralMidi =
   WriteMidi.fromGMPerformanceMixedAuto .
   FancyPerformance.floatFromMusic
```

3. generate a MIDI file

```
fileFromGeneralMIDIMusic :: FilePath -> MidiMusic.T -> IO ()
fileFromGeneralMIDIMusic filename = SaveMidi.toFile filename . generalMidi
```

4. generate and play a MIDI file on Windows 95, Windows NT, or Linux

```
fileName :: FilePath
fileName = "test.mid"

play :: String -> [String] -> MidiMusic.T -> IO ExitCode
play cmd opts m =
   do fileFromGeneralMIDIMusic fileName m
      rawSystem cmd (opts ++ [fileName])

playWin95, playWinNT,
   playLinux, playAlsa, playTimidity, playTimidityJack :: MidiMusic.T -> IO ExitCode
playWin95        = play "mplayer" []
playWinNT        = play "mplay32" []
playLinux        = play "playmidi" ["-rf"]
playAlsa         = play "pmidi" ["-p 128:0"]
playTimidity     = play "timidity" ["-B8,9"]
playTimidityJack = play "timidity" ["-Oj"]
```

Alternatively, just run `fileFromGeneralMIDIMusic "test.mid" m` manually, and then invoke the midi player on your system using `playTest`, defined below for NT:

```
playTest :: IO ExitCode
playTest =
   rawSystem "mplay32" [fileName]
```

A.1.2 Some General Midi test functions

Use these functions with caution.

A General Midi user patch map; i.e. one that maps GM instrument names to themselves, using a channel that is the patch number modulo 16. This is for use ONLY in the code that follows, o/w channel duplication is possible, which will screw things up in general.

```
gmUpm :: InstrMap.ChannelProgramTable MidiMusic.Instr
gmUpm =
   zipWith
      (\instr chan ->
         (instr, (chan, GeneralMidi.instrumentToProgram instr)))
      GeneralMidi.instruments
      (cycle $ map ChannelMsg.toChannel [0..15])
```

Something to play each "instrument group" of 8 GM instruments; this function will play a C major arpeggio on each instrument.

```
gmTest :: Int -> IO ()
gmTest i =
   let gMM = take 8 (drop (i*8) GeneralMidi.instruments)
       mu  = Music.line (map simple gMM)
       simple instr = MidiMusic.fromMelodyNullAttr instr Melody.cMajArp
   in  fileFromRhythmicMusic fileName
          ([], gmUpm, FancyPerformance.context ::
                   Context.T NonNegW.Float Float MidiMusic.Note, mu)
```

A.2 Utility functions

```
module Haskore.General.Utility where

import Control.Monad.Trans.State (state, runState, )
import System.Random (RandomGen, randomR, randomRs, mkStdGen, )
import Data.List.HT (segmentBefore, partition, )
import Data.List (foldl', )
import Data.Ratio ((%), denominator, numerator, Ratio, )
import Data.Maybe (fromMaybe, listToMaybe, )
import Data.Char (ord, chr, )
import Data.Word (Word8, )
import qualified Haskore.General.Map as Map
```

`splitBy` takes a boolean test and a list; it divides up the list and turns it into a *list of sub-lists*; each sub-list consists of

1. one element for which the test is true (or the first element in the list), and
2. all elements after that element for which the test is false.

For example, `splitBy (>10) [27, 0, 2, 1, 15, 3, 42, 4]` yields `[ [27,0,2,1], [15,3], [42,4] ]`.

```
splitBy :: (a -> Bool) -> [a] -> [[a]]
splitBy p = dropWhile null . segmentBefore p
```

`segmentBefore` will have at most one empty list at the beginning, which is dropped by `dropWhile`.

It should have signature segmentBefore :: (a -> Bool) -> [a] -> ([a], [(a, [a])]) or even better segmentBefore :: (a -> Bool) -> [a] -> AlternatingListUniform.T a [a] and could be implemented using Uniform.fromEitherList

A variant of `foldr` and `foldr1` which works only for non-empty lists and initializes the accumulator depending on the last element of the list.

```
foldrf :: (a -> b -> b) -> (a -> b) -> [a] -> b
foldrf f g =
   let aux []     = error "foldrf: list must be non-empty"
       aux (x:[]) = g x
       aux (x:xs) = f x (aux xs)
   in  aux
```

Randomly permutate a list. For this purpose we generate a random `Bool` value for each item of the list which specifies in what sublist it is inserted. Both sublists are then concatenated hereafter. By repeating this procedure several times the list should be somehow randomly ordered.

Some notes about perfect shuffling from Oleg: http://okmij.org/ftp/Haskell/misc.html#perfect-shuffle

```
shuffle :: RandomGen g => [a] -> g -> ([a],g)
shuffle x g0 =
   let (choices,g1) =
          runState (mapM (const (state (randomR (False,True)))) x) g0
       xc = zip x choices
   in  (map fst (uncurry (++) (partition snd xc)), g1)
```

`flattenTuples2` flattens a list of pairs into a list. Similarly, `flattenTuples3` flattens a list of 3-tuples into a list, and so on.

```
flattenTuples2 :: [(a,a)]     -> [a]
flattenTuples3 :: [(a,a,a)]   -> [a]
flattenTuples4 :: [(a,a,a,a)] -> [a]

flattenTuples2 = concatMap (\(x,y)     -> [x,y])
flattenTuples3 = concatMap (\(x,y,z)   -> [x,y,z])
flattenTuples4 = concatMap (\(x,y,z,w) -> [x,y,z,w])
```

Choose the first element from a list, and return the default value, if the list is empty.

```
headWithDefault :: a -> [a] -> a
headWithDefault deflt = fromMaybe deflt . listToMaybe
```

Implementation with the partial function `head`, which is a bad thing.

```
headWithDefault deflt xs = head (xs ++ [deflt])
```

Compare

```
let (x,y) = splitInit [0..] in (last x, y)
```

and

```
let as = [0..]; (x,y) = (init as, last as) in (last x, y)
```

Variants of `zip` and `zip3` which check that all argument lists have the same length.

```
zipWithMatch :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWithMatch f (x:xs) (y:ys) = f x y : zipWithMatch f xs ys
zipWithMatch _ [] [] = []
zipWithMatch _ _ _ = error "zipWithMatch: lengths of lists differ"

zipWithMatch3 :: (a -> b -> c -> d) -> [a] -> [b] -> [c] -> [d]
zipWithMatch3 f (x:xs) (y:ys) (z:zs) = f x y z : zipWithMatch3 f xs ys zs
zipWithMatch3 _ [] [] [] = []
zipWithMatch3 _ _ _ _ = error "zipWithMatch3: lengths of lists differ"
```

This is a variant of `maximum` which returns at least zero, i.e. always a non-negative number. This is necessary for determining the length of a parallel music composition where the empty list has zero duration.

```
maximum0 :: (Ord a, Num a) => [a] -> a
maximum0 = foldl' max 0
```

Convert a mapping (i.e. list of pairs) to a function, and use this for a translation function, which translates every character in a by replacing it by looking it up in l2 and replacing it with the according character in l2.

```
translate :: (Ord a) => [ a ] -> [ a ] -> [ a ] -> [ a ]
translate l1 l2 a =
   if length l1 == length l2
   then let table = Map.fromList (zip l1 l2)
        in  map (\x -> Map.findWithDefault table x x) a
   else error "translate: lists must have equal lengths"
```

A random list of integers between 0 and n.

```
randList :: Int -> [ Int ]
randList n = randomRs (0, n) (mkStdGen 0)
```

Is one rational divisible by another one (i.e., is it a integer multiple of it)?

```
divisible :: Integral a => Ratio a -> Ratio a -> Bool
divisible r1 r2 =
   0 == mod (numerator r1 * denominator r2)
            (numerator r2 * denominator r1)
```

Do the division.

```
divide :: Integral a => Ratio a -> Ratio a -> a
divide r1 r2 =
   let (q, r) = divideModulus r1 r2
   in  if r == 0
```

```
       then q
       else error "Utility.divide: rationals are indivisible"

modulus :: Integral a => Ratio a -> Ratio a -> Ratio a
modulus r1 r2 = snd (divideModulus r1 r2)

divideModulus :: Integral a => Ratio a -> Ratio a -> (a, Ratio a)
divideModulus r1 r2 =
   let (q, r) = divMod (numerator r1 * denominator r2)
                       (numerator r2 * denominator r1)
   in  (q, r % (denominator r1 * denominator r2))
```

Also the GCD can be generalized to ratios:

```
gcdDur :: Integral a => Ratio a -> Ratio a -> Ratio a
gcdDur x1 x2 =
   let a = numerator x1
       b = denominator x1
       c = numerator x2
       d = denominator x2
   in  gcd a c % lcm b d
```

```
type ByteList = [Word8]

stringCharFromByte :: ByteList -> String
stringCharFromByte = map (chr . fromIntegral)

stringByteFromChar :: String -> ByteList
stringByteFromChar = map (fromIntegral . ord)
```

B Examples

B.1 Haskore in Action

```
module Haskore.Example.Miscellaneous where

import           Haskore.Composition.Trill as Trill
import           Haskore.Composition.Drum  as Drum

import qualified Haskore.Music           as Music
import           Haskore.Music (rest, delay, (/=:))
import           Haskore.Music.GeneralMIDI as MidiMusic
import qualified Haskore.Music.Rhythmic  as RhyMusic
import qualified Haskore.Melody          as Melody
import           Haskore.Melody.Standard as StdMelody
import qualified Haskore.Performance.Context as Context

import qualified Haskore.Interface.MIDI.InstrumentMap as InstrMap
import qualified Haskore.Interface.MIDI.Write        as WriteMidi
import qualified Haskore.Interface.MIDI.Read         as ReadMidi
import qualified Haskore.Interface.MIDI.Render       as Render
```

```
import qualified Sound.MIDI.File.Save    as SaveMidi
import qualified Sound.MIDI.File.Load    as LoadMidi
import qualified Sound.MIDI.File         as MidiFile
import qualified Sound.MIDI.General      as GeneralMidi

import qualified Haskore.Example.SelfSim        as SelfSim
import qualified Haskore.Example.ChildSong6     as ChildSong6
import qualified Haskore.Example.Ssf            as Ssf

import           Haskore.Basic.Duration ((%+))
import qualified Numeric.NonNegative.Wrapper as NonNeg

import Data.Tuple.HT (fst3, snd3, thd3, )


t0, t1, t2, t3, t4, t5,
 t10s, t12, t12a, t13, t13a, t13b, t13c, t13d, t13e,
 t14, t14b, t14c, t14d, cs6, ssf0 :: MidiFile.T

piano, vibes, flute :: GeneralMidi.Instrument
piano = GeneralMidi.AcousticGrandPiano
vibes = GeneralMidi.Vibraphone
flute = GeneralMidi.Flute
```

Simple examples of Haskore in action. Note that this module also imports modules ChildSong6, SelfSim, and Ssf.

From the tutorial, try things such as pr12, cMajArp, cMajChd, etc. and try applying inversions, retrogrades, etc. on the same examples. Also try `ChildSong.song`. For example:

```
t0 = Render.generalMidiDeflt ChildSong6.song
```

C Major scale for use in examples below:

```
cms', cms :: Melody.T ()
cms' = line (map (\n -> n en ())
          [c 0, d 0, e 0, f 0, g 0, a 0, b 0, c 1])
cms = changeTempo 2 cms'

drumScale :: MidiMusic.T
drumScale =
   line (map (\n -> Drum.toMusicDefaultAttr (toEnum (n+13)) sn)
             [0,2,4,5,7,9,11,12])
```

Test of various articulations and dynamics:

```
t1 = Render.generalMidi
       (staccato (sn/10) drumScale +:+
                         drumScale +:+
        legato   (sn/10) drumScale    )
```

```
temp, mu2 :: MidiMusic.T
temp = MidiMusic.fromMelodyNullAttr piano (crescendo 4.0 (c 0 en ()))

mu2 = MidiMusic.fromMelodyNullAttr vibes
       (diminuendo 0.75 cms +:+
        crescendo 0.75 (loudness1 0.25 cms))
t2 = Render.generalMidiDeflt mu2

t3 = Render.generalMidiDeflt (MidiMusic.fromMelodyNullAttr flute
       (accelerando 0.3 cms +:+
        ritardando  0.6 cms    ))
```

A function to recursively apply transformations `f'` (to elements in a sequence) and `g'` (to accumulated phrases):

```
rep :: (Music.T note -> Music.T note)
    -> (Music.T note -> Music.T note)
    -> Int -> Music.T note -> Music.T note
rep _  _  0 _ = rest 0
rep f' g' n m = m =:= g' (rep f' g' (n-1) (f' m))
```

An example using "rep" three times, recursively, to create a "cascade" of sounds.

```
run, cascade, cascades :: Melody.T ()
run       = rep (transpose 5) (delay tn) 8 (c 0 tn ())
cascade   = rep (transpose 4) (delay en) 8 run
cascades  = rep  id           (delay sn) 2 cascade

t4' :: Melody.T () -> MidiFile.T
t4' x     = Render.generalMidiDeflt (MidiMusic.fromMelodyNullAttr piano x)
t4        = Render.generalMidiDeflt (MidiMusic.fromMelodyNullAttr piano
              (cascades +:+ Music.reverse cascades))
```

What happens if we simply reverse the `f` and `g` arguments?

```
run', cascade', cascades' :: Melody.T ()
run'      = rep (delay tn) (transpose 5) 4 (c 0 tn ())
cascade'  = rep (delay en) (transpose 4) 6 run'
cascades' = rep (delay sn)  id           2 cascade'
t5        = Render.generalMidiDeflt (MidiMusic.fromMelodyNullAttr piano cascades')
```

Example from the SelfSim module.

```
t10s   = Render.generalMidiDeflt (rep (delay SelfSim.durss) (transpose 4) 2 SelfSim.ss)
```

Example from the ChildSong6 module.

```
cs6 = Render.generalMidiDeflt ChildSong6.song
```

Example from the Ssf (Stars and Stripes Forever) module.

```
ssf0 = Render.generalMidiDeflt Ssf.song
```

Midi percussion test. Plays all "notes" in a range. (Requires adding an instrument for percussion to the `InstrMap`.)

```
drums :: GeneralMidi.Drum -> GeneralMidi.Drum -> MidiMusic.T
drums dr0 dr1 =
   line (map (\drm -> Drum.toMusicDefaultAttr drm sn) [dr0..dr1])

t11 :: GeneralMidi.Drum -> GeneralMidi.Drum -> MidiFile.T
t11 dr0 dr1 = Render.generalMidiDeflt (drums dr0 dr1)
```

Test of `Music.take` and shorten.

```
t12 = Render.generalMidiDeflt (Music.take 4 ChildSong6.song)
t12a =
   Render.generalMidiDeflt
      (MidiMusic.fromMelodyNullAttr piano cms /=: ChildSong6.song)
```

Tests of the trill functions.

```
t13note :: MidiMusic.T
t13note = MidiMusic.fromMelodyNullAttr piano (c 1 qn ())
t13 =  Render.generalMidiDeflt (trill   1 sn t13note)
t13a = Render.generalMidiDeflt (trill'  2 dqn t13note)
t13b = Render.generalMidiDeflt (trillN  1 5 t13note)
t13c = Render.generalMidiDeflt (trillN' 3 7 t13note)
t13d = Render.generalMidiDeflt (roll tn t13note)
t13e = Render.generalMidiDeflt (changeTempo (2/3) (transpose 2 (trillN' 2 7 t13note)))
```

Tests of drum.

```
t14 = Render.generalMidiDeflt (Drum.toMusicDefaultAttr AcousticSnare qn)
```

A "funk groove"

```
t14b = let p1 = Drum.toMusicDefaultAttr LowTom        qn
           p2 = Drum.toMusicDefaultAttr AcousticSnare en
       in Render.generalMidiDeflt (changeTempo 3 (Music.replicate 4
                 (line [p1, qnr, p2,  qnr, p2,
                        p1, p1,  qnr, p2,  enr]
                  =:= roll en (Drum.toMusicDefaultAttr ClosedHiHat 2))))
```

A "jazz groove"

```
t14c = let p1 = Drum.toMusicDefaultAttr CrashCymbal2  qn
           p2 = Drum.toMusicDefaultAttr AcousticSnare en
           p3 = Drum.toMusicDefaultAttr LowTom        qn
       in Render.generalMidiDeflt (changeTempo 3 (Music.replicate 8
```

```
                ((p1 +:+ changeTempo (3%+2) (p2 +:+ enr +:+ p2))
                 =:= (p3 +:+ qnr)) ))

t14d = let p1 = Drum.toMusicDefaultAttr LowTom        en
           p2 = Drum.toMusicDefaultAttr AcousticSnare hn
       in Render.generalMidiDeflt(line [roll tn p1,
                        p1,
                        p1,
                        rest en,
                        roll tn p1,
                        p1,
                        p1,
                        rest qn,
                        roll tn p2,
                        p1,
                        p1]  )
```

Tests of the MIDI interface. `MidiMusic.T` into a MIDI file.

```
tab :: MidiMusic.T -> IO ()
tab m = SaveMidi.toFile "test.mid" (Render.generalMidiDeflt m)
```

`MidiMusic.T` to a MidiFile datatype and back to Music.

```
type StdContext =
   Context.T NonNeg.Float Float (RhyMusic.Note MidiMusic.Drum MidiMusic.Instr)
-- type StdContext = Pf.Context NonNeg.Float Float MidiMusic.Note -- rejected by Hugs

type MidiArrange =
   (InstrMap.ChannelTable MidiMusic.Instr, StdContext, MidiMusic.T)

tad :: MidiMusic.T -> MidiArrange
tad = ReadMidi.toGMMusic . Render.generalMidiDeflt
```

A MIDI file to a MidiFile datatype and back to a MIDI file.

```
tcb, tc, tcd, tcdab :: FilePath -> IO ()
tcb file = LoadMidi.fromFile file >>= SaveMidi.toFile "test.mid"
```

MIDI file to MidiFile datatype.

```
tc file = LoadMidi.fromFile file >>= print
```

MIDI file to `MidiMusic.T`, a `InstrMap`, and a `Context`.

```
tcd file = do
             x <- fmap ReadMidi.toGMMusic
                       (LoadMidi.fromFile file)
             print $ fst3 (x::MidiArrange)
             print $ snd3 x
             print $ thd3 x
```

A MIDI file to `MidiMusic.T` and back to a MIDI file.

```
tcdab file =
   LoadMidi.fromFile file >>=
      (SaveMidi.toFile "test.mid" . WriteMidi.fromGMMusic .
         (id::MidiArrange -> MidiArrange) . ReadMidi.toGMMusic)
```

B.2 Children's Song No. 6

This is a partial encoding of Chick Corea's "Children's Song No. 6".

```
module Haskore.Example.ChildSong6 where

import           Haskore.Melody.Standard   as Melody
import           Haskore.Music.GeneralMIDI as MidiMusic
import qualified Haskore.Music             as Music
```

note updaters for mappings

```
fd :: t -> (t -> NoteAttributes -> m) -> m
fd dur n = n dur v

vel :: (NoteAttributes -> m) -> m
vel  n   = n   v

v :: NoteAttributes
v        = Melody.na

lmap :: (a -> Melody.T) -> [a] -> Melody.T
lmap func l = line (map func l)


bassLine, mainVoice :: Melody.T
song :: MidiMusic.T
```

Baseline:

```
b1, b2, b3 :: Melody.T
b1 = lmap (fd dqn) [b  3, fs 4, g  4, fs 4]
b2 = lmap (fd dqn) [b  3, es 4, fs 4, es 4]
b3 = lmap (fd dqn) [as 3, fs 4, g  4, fs 4]

bassLine =
   Music.loudness1 (10/13)
      (line [Music.replicate 3 b1, Music.replicate 2 b2,
             Music.replicate 4 b3, Music.replicate 5 b1])
```

Main Voice:

```
v1, v1a, v1b :: Melody.T
v1  = v1a +:+ v1b
v1a = lmap (fd en) [a 5, e 5, d 5, fs 5, cs 5, b 4, e 5, b 4]
v1b = lmap vel     [cs 5 tn, d 5 (qn-tn), cs 5 en, b 4 en]
```

```
v2, v2a, v2b, v2c, v2d, v2e, v2f :: Melody.T
v2  = line [v2a, v2b, v2c, v2d, v2e, v2f]
v2a = lmap vel [cs 5 (dhn+dhn), d 5 dhn,
                f 5 hn, gs 5 qn, fs 5 (hn+en), g 5 en]
v2b = lmap (fd en) [fs 5, e 5, cs 5, as 4] +:+ a 4 dqn v +:+
      lmap (fd en) [as 4, cs 5, fs 5, e 5, fs 5, g 5, as 5]
v2c = lmap vel [cs 6 (hn+en), d 6 en, cs 6 en, e 5 en] +:+ enr +:+
      lmap vel [as 5 en, a 5 en, g 5 en, d 5 qn, c 5 en, cs 5 en]
v2d = lmap (fd en) [fs 5, cs 5, e 5, cs 5, a 4, as 4, d 5, e 5, fs 5] +:+
      lmap vel [fs 5 tn, e 5 (qn-tn), d 5 en, e 5 tn, d 5 (qn-tn),
                cs 5 en, d 5 tn, cs 5 (qn-tn), b 4 (en+hn)]
v2e = lmap vel [cs 5 en, b 4 en, fs 5 en, a 5 en, b 5 (hn+qn), a 5 en,
                fs 5 en, e 5 qn, d 5 en, fs 5 en, e 5 hn, d 5 hn, fs 5 qn]
v2f = changeTempo (3/2) (lmap vel [cs 5 en, d 5 en, cs 5 en]) +:+ b 4 (3*dhn+hn) v

mainVoice = Music.replicate 3 v1 +:+ v2
```

Putting it all together:

```
song = MidiMusic.fromStdMelody MidiMusic.AcousticGrandPiano
          (transpose (-48) (changeTempo 3
             (bassLine =:= mainVoice)))
```

B.3 Self-Similar (Fractal) Music.T

```
module Haskore.Example.SelfSim where

import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Melody as Melody
import qualified Haskore.Music  as Music
import           Haskore.Music.GeneralMIDI as MidiMusic
import qualified Haskore.Interface.MIDI.Render as Render
import qualified Sound.MIDI.File   as MidiFile
```

An example of self-similar, or fractal, music.

```
data Cluster = Cl SNote [Cluster]  -- this is called a Rose tree
type Pat     = [SNote]
type SNote   = [(Pitch.Absolute,Dur)]    -- i.e. a chord

sim :: Pat -> [Cluster]
sim pat = map mkCluster pat
    where mkCluster notes = Cl notes (map (mkCluster . addmult notes) pat)


addmult :: (Num a, Num b) => [(a, b)] -> [(a, b)] -> [(a, b)]
addmult pds iss = zipWith addmult' pds iss
                  where addmult' (p,d) (i,s) = (p+i,d*s)

simFringe :: (Num a, Eq a) => a -> Pat -> [SNote]
simFringe n pat = fringe n (Cl [(0,0)] (sim pat))
```

```
fringe :: (Num a, Eq a) => a -> Cluster -> [SNote]
fringe 0 (Cl n _)   = [n]
fringe m (Cl _ cls) = concatMap (fringe (m-1)) cls

-- this just converts the result to Haskore:
simToHask :: [[(Pitch.Absolute, Music.Dur)]] -> Melody.T ()
simToHask s = let mkNote (p,d) = Melody.note (Pitch.fromInt p) d ()
              in line (map (chord . map mkNote) s)

-- and here are some examples of it being applied:

sim4 :: Int -> Melody.T ()
sim1, sim2, sim12, sim3, sim4s :: Int -> MidiMusic.T
t6, t7, t8, t9, t10 :: MidiFile.T

sim1 n = MidiMusic.fromMelodyNullAttr MidiMusic.AcousticBass
           (transpose (-12)
              (changeTempo 4 (simToHask (simFringe n pat1))))
t6 = Render.generalMidiDeflt (sim1 4)

sim2 n = MidiMusic.fromMelodyNullAttr MidiMusic.AcousticGrandPiano
           (transpose 5
              (changeTempo 4 (simToHask (simFringe n pat2))))
t7 = Render.generalMidiDeflt (sim2 4)

sim12 n = sim1 n =:= sim2 n
t8 = Render.generalMidiDeflt (sim12 4)

sim3 n = MidiMusic.fromMelodyNullAttr MidiMusic.Vibraphone
           (transpose 0
              (changeTempo 4 (simToHask (simFringe n pat3))))
t9 = Render.generalMidiDeflt (sim3 3)

sim4 n  = (transpose 12
              (changeTempo 2 (simToHask (simFringe n pat4'))))

sim4s n = let s = sim4 n
              l1 = MidiMusic.fromMelodyNullAttr MidiMusic.Flute s
              l2 = MidiMusic.fromMelodyNullAttr MidiMusic.AcousticBass
                      (transpose (-36) (Music.reverse s))
          in  l1 =:= l2

ss :: MidiMusic.T
ss     = sim4s 3
durss :: Music.Dur
durss  = Music.dur ss

t10    = Render.generalMidiDeflt ss

pat1, pat2, pat3, pat4, pat4' :: [SNote]
pat1 = [[(0,1.0)],[(4,0.5)],[(7,1.0)],[(5,0.5)]]
pat2 = [[(0,0.5)],[(4,1.0)],[(7,0.5)],[(5,1.0)]]
pat3 = [[(2,0.6)],[(5,1.3)],[(0,1.0)],[(7,0.9)]]
pat4' = [[(3,0.5)],[(4,0.25)],[(0,0.25)],[(6,1.0)]]
pat4 = [[(3,0.5),(8,0.5),(22,0.5)],[(4,0.25),(7,0.25),(21,0.25)],
```

```
        [(0,0.25),(5,0.25),(15,0.25)],[(6,1.0),(9,1.0),(19,1.0)]]
```

B.4 Guitar

In this section we want to develop a simulation of a guitar. This clearly demonstrates the power of our music-by-programming approach. After writing some routines for doing the mechanical stuff we can describe the music concisely as a sequence of chords.

```
module Haskore.Example.Guitar where

import qualified Haskore.Basic.Pitch as Pitch
import           Haskore.Basic.Pitch (Class(..))
import qualified Haskore.Basic.Duration as Dur
-- import           Haskore.Melody.Standard   as StdMelody
import           Haskore.Music.GeneralMIDI as MidiMusic
import           Haskore.Music.Rhythmic    as RhyMusic
import qualified Haskore.Melody            as Melody
import qualified Haskore.Music             as Music

import qualified Data.List as List
```

On a guitar a chord is not played as an immediate sequence of the constituting notes, but the order and the number of occurences of each tone is adapted to the guitar and the possibilities of the player. We want to automatically design a sequence of tones that represents a given chord. Our approach is simple: For every string we choose the lowest possible note which occurs in the chord. This way we may miss notes of the chord, but we have a good approximation. If a chord consists of more than six notes, we have to ignore some notes definitely.

For given pitches of all guitar strings and the pitch classes of a chord, `mapChordToString` compute the tones that are played on each string of the guitar.

```
mapChordToString :: [Pitch.T] -> [Pitch.Class] -> [Pitch.T]
mapChordToString strs chrd =
   map (choosePitchForString chrd) strs

choosePitchForString :: [Pitch.Class] -> Pitch.T -> Pitch.T
choosePitchForString chrd str@(_,pc) =
   let diff x = mod (Pitch.classToInt x - Pitch.classToInt pc) 12
       smallestDiff = minimum (map diff chrd)
   in  Pitch.transpose smallestDiff str

stringPitches :: [Pitch.T]
stringPitches =
   reverse [(-2,E), (-2,A), (-1,D), (-1,G), (-1,B), (0,E)]
```

Once we obtain the tones that are played on a guitar we want to arrange them into a guitar like melody. We distinguish between up strokes and down strokes, which are often played alternatingly. According to the stroke direction, the low notes are played slightly before the high ones and vice versa. We define the respective delays for each string. Since both direction are perceived differently, we have to prefetch the down strokes a bit.

```
data Direction =
     Up
   | Down

delayTime :: Dur
delayTime = en/15

dirDelays :: Direction -> [Dur.Offset]
dirDelays dir =
   map (Dur.toRatio delayTime *)
      (case dir of
         Up   -> [0..5]
         Down -> [2,1..(-3)])
```

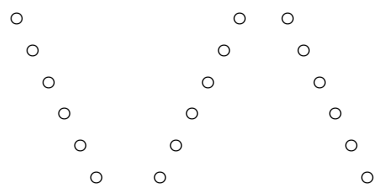
Here is the only creative part: The essential description of the guitar music.

```
type UpDownPattern = [(Dur, Direction)]

udp, udpInter, udpLast :: UpDownPattern
udp      = [(qn,Up), (en,Down), (qn,Up), (en,Down), (qn,   Up)]
udpInter = [(qn,Up), (en,Down), (qn,Up), (en,Down), (en,Up), (en,Down)]
udpLast  = [(qn,Up), (en,Down), (qn,Up), (en,Down), (qn+wn,Up)]

chords :: [([Pitch.Class], UpDownPattern)]
chords =
   [([C,E,G],    udp),
    ([C,E,G,Bf], udp),
    ([F,A,C],    udp),
    ([F,Af,C],   udpInter),
    ([C,E,G],    udp),
    ([G,B,D],    udp),
    ([C,F,G],    udp),
    ([C,E,G],    udpLast)]
```

The next step is to arrange the notes corresponding to the chords.

```
type DelayedNote = (Dur.Offset, (Dur, Maybe Pitch.T))

chordToPattern :: [Pitch.Class] -> UpDownPattern -> [[DelayedNote]]
chordToPattern chrd =
   map (\(dur,ord) ->
      zipWith
         (\delay p -> (delay, (dur, Just p)))
         (dirDelays ord)
         (mapChordToString stringPitches chrd))

guitarEvents :: [[DelayedNote]]
guitarEvents =
   concatMap (uncurry chordToPattern) chords
```

We want to simulate the guitar by a parallel composition of six strings. The sound of each string finishes when the next sound on the string is played. Thus we have to compute the time each string oscillates. Finally we want to obtain this pattern of events:



```
delayNotes :: [DelayedNote] -> [Melody.T ()]
delayNotes m =
   let zero = (0, (0, Nothing))
   in  zipWith
          (\(d0, (dur, at)) (d1, _) ->
               Music.atom (Dur.add (d1-d0) dur)
                  (fmap (Melody.Note ()) at))
          (zero : m) (m ++ [zero])

stringMelodies :: [Melody.T ()]
stringMelodies =
   map (line . delayNotes) (List.transpose guitarEvents)

parallelSong :: [instr] -> RhyMusic.T drum instr
parallelSong instrs =
   changeTempo 2 (chord (zipWith RhyMusic.fromMelodyNullAttr
                                 instrs stringMelodies))

parallelSongMIDI :: MidiMusic.T
parallelSongMIDI =
   transpose 12 (parallelSong (repeat MidiMusic.ElectricGuitarClean))
```

Unfortunately the Guitar music appears to be slightly longer than it is on the note sheet. To workaround that we use notes of very short duration but very long legato. For simplicity this simulation is not as precise as the one above. We don't prefetch the down strokes and we do not exactly care for the correct length of the string sounds. The resulting MIDI files does still not sound satisfyingly because notes of equal pitch overlap, which is not properly supported by MIDI.

```
<----------------->
                <--------------->
```

The end of the first note terminates the second one, which is not intended. Of course, you can play the MidiMusic using other back ends.

```
chordWithLegatoPattern ::
   [RhyMusic.T drum instr] -> UpDownPattern -> RhyMusic.T drum instr
chordWithLegatoPattern tones pattern =
   let beat (dur, dir) =
          legato dur
             (line (case dir of {Up -> tones; Down -> reverse tones}) +:+
              Music.rest (dur - delayTime * List.genericLength tones))
   in  line (map beat pattern)
```

```
legatoSong :: [instr] -> RhyMusic.T drum instr
legatoSong instrs =
   changeTempo 2 (line (map
      (uncurry
         (chordWithLegatoPattern .
          zipWith RhyMusic.fromMelodyNullAttr instrs .
          map (Music.atom delayTime . Just . Melody.Note ()) .
          mapChordToString stringPitches))
      chords))

legatoSongMIDI :: MidiMusic.T
legatoSongMIDI =
   transpose 12 (legatoSong (repeat MidiMusic.ElectricGuitarClean))
```

C Design discussion

This section presents the advantages and disadvantages of several design decisions that has been made.

Principal type `T` Analogously to Modula-3 we use the following naming scheme: A module has the name of the principal type and the type itself has the name `T`. If there is only one constructor for that type its name is `Cons`. If the main object of a module is a type class, its name is `C`. A function in a module don't need a prefix related to the principal type. Many functions can be considered as conversion functions. They should be named `TargetType.fromSourceType` or `SourceType.toTargetType`. If there is a choice, the first form is prefered. This does better fit to the order of functions and their arguments. Compare `a = A.fromB b` and `a = B.toA b`.

A programmer using such a module is encouraged to import it with qualified identifiers. This way the programmer may abbreviate the module name to its convenience.

`Music.T` The data structure should be hidden. The user should use `changeTempo` and similar functions instead of the constructors `Tempo` etc. This way the definition of a `Music.T` stays independent from the actual data structure `Music.T`. Then `changeTempo` can be implemented silently using a constructor or using a mapping function.

`Medium.T` The idea of extracting the structure of animation movies and music into an abstract data structure is taken from Paul Hudak's paper "An Algebraic Theory of Polymorphic Temporal Media".

The temporial media data structure `Medium.T` is used here as the basis type for Haskore's Music.

Binary composition vs. List composition There are two natural representations for temporal media. We have implemented both of them:

1. `Medium.Plain.Binary` uses binary constructors `:+:`, `:=:`
2. `Medium.Plain.List` uses List constructors `Serial`, `Parallel`

Both of these modules provide the functions `foldBinFlat` and `foldListFlat` which apply binary functions or list functions, respectively, to `Medium.T`. Import your prefered module to `Medium`.

Each of these data structures has its advantages:

`Medium.Binary.T`

- There is only one way to represent a zero object, which must be a single media primitive (`Prim`).
- You need only a few constructors for serial and parallel compositions.

`Medium.List.T`

- Zero objects can be represented without a particalur zero primitives.
- You can represent two different zero objects, an empty parallelism and an empty serialism. Both can be interpreted as limits of compositions of decreasing size.
- You can store music with an internal structure which is lost in a performance. E.g. a serial composition of serial compositions will sound identical to a flattened serial composition, but the separation might contain additional information.

In my (Henning's) opinion `Music.T` is for representing musical ideas and `Performance.T` is for representing the sound of a song. Thus it is ok and even useful if there are several ways to represent the same sound impression (`Performance.T`) in different ways (`Music.T`), just like it is possible to write very different LaTeX code which results in the same page graphics. The same style of text may have different meanings which can be seen only in the LaTeX source code. Analogously music can be structured more detailed than one can hear.

Algebraic structure The type `Medium.T` almost forms an algebraic ring where `=:=` is like a sum (commutative) and `+:+` is like a product (non-commutative). Unfortunately `Medium.T` is not really a ring: There are no inverse elements with respect to addition (`=:=`). Further `=:=` is not distributive with respect to `+:+` because `x` is different from `x =:= x`. There is also a problem if the durations of the parallel music objects differ. I.e. if `dur y /= dur z` then `x +:+ (y =:= z)` is different from `(x +:+ y) =:= (x +:+ z)` even if `x == x =:= x` holds. So it is probably better not to make `Medium.T` an instance of a `Ring` type class. (In Prelude 98 the class `Num` is quite a `Ring` type class.)

Relative times in `Performance.T` Absolute times for events disallow infinite streams of music. The time information becomes more and more inaccurate and finally there is an overflow or no change in time. Relative times make synchronization difficult, especially many small time differences are critical. But since the `Music.T` is inherently based on time differences one cannot get rid of sum rounding errors. The problem can only be weakened by more precise floating point formats.

Type variable for time and dynamics in `Performance.T` In the original design of Haskore `Float` was the only fractional type used for time and volume measures in `Performance.T`. This is good with respect to efficiency. But rounding errors make it almost impossible to test literal equivalence (Section 3.2.1) between different music expressions. In order to match both applications I introduced type variables `time` and `dyn` which is now floating all around. It also needs some explicit type hints in some cases where the performance is only an interim step. In future `Music.T` itself might get a `time` type parameter. We should certainly declare types for every-day use such as `CommonMusic.T` which instantiates `Music.T` with `Double` or so.

Unification of Rests and Notes Since rests and notes share the property of the duration, the constructor `Music.Atom` is used which handles the duration and the particalur music primitive, namely Rest and Note. All functions concerning duration (`dur`, `cut`) don't need to interpret the musical primitive.

Pitch With the definition `Pitch = (Octave, PitchClass)` (swapped order with respect to original Haskore) the order on `Pitch` equals the order on pitches. Functions like `o0`, `o1`, `o2` etc. may support this order for short style functional note definitions. It should be e.g. `o0 g == g 0`. Alternatively one can put this into a duration function like `qn'`, `en'`, etc. Then it must hold e.g. `qn' 0 g == g 0 qn`

The problem is that the range of notes of the enumeration `PitchClass` overlaps with notes from neighbouring octaves. Overlapping `PitchClasses`, e.g. `(0,Bs) < (1,Cf)` although `absPitch (0,Bs) > absPitch (1,Cf)`

The musical naming of notes is a bit unlogical. The range is not from A to G but from C to B. Further on there are two octaves with note names without indices (e.g. A and a). Both octaves are candidates for a "zero" octave. We define that octave 0 is the one which contains a.

Absolute pitch Find a definition for the absolute pitch that will be commonly used for MIDI, CSound, and Signal output.

Yamaha-SY35 manual says:

- Note $00 - (-2,C)
- Note $7F - (8,G)

But which A is 440 Hz?

By playing around with the Multi key range I found out that the keyboard ranges from (1,C) to (6,C) (in MIDI terms). The frequencies of the instruments played at the same note are not equal. :-(Many of them have (3,A) (MIDI) = 440 Hz, but some are an octave below, some are an octave above. In CSound it was (8,A) = 440 Hz in original Haskore. Very confusing.

Volume vs. Velocity MIDI distinguishes Volume and Velocity. Volume is related to the physical amplitude, i.e. if we want to change the Volume of a sound we simply amplify the sound by a constant factor. In contrast to that Velocity means the speed with which a key is pressed or released. This is most oftenly

interpreted as the force with which an instrument is played. This distinction is very sensible and is reflected in `Music.T`. Velocity is inherently related to the beginning and the end of a note, whereas the Volume can be changed everywhere. All phrases related to dynamics are mapped to velocities and not to volumes, since one cannot change the volume of natural instruments without changing the force to play them (and thus changing their timbre). The control of Volume is to be added later, together with controllers like pitch bender, frequency modulation and so on.

Global instrument setting vs. note attribute In the original version of Haskore, there was an `Instr` constructor that set the instrument used in the enclosed piece of music. I found that changing an instrument by surrounding a piece of music with a special constructor is not very natural. On which parts of the piece it has an effect or if it has an effect at all depends on `Instr` statements within the piece of music. To assert that instruments are set once and only once and that setting an instrument has an effect, we distinguish between (instrument-less) melodies and music (with instrument information) now. In a melody we store only notes and rests, in a music we store an instrument for any note. Even more since the instrument is stored for each note this can be interpreted as an instrument event, where some instruments support note pitches and others not (sound effects) or other attributes (velocity).

PhraseFun The original Haskore version used `PhraseFuns` of the type `Music.T -> (Performance.T, Dur)`. This way it was a bit cumbersome to combine different phrases. In principle all `PhraseFuns` could be of type `(Performance.T, Dur) -> (Performance.T, Dur)` This would be a more clean design but lacks some efficiency because e.g. the Loudness can be controlled by changing the default velocity of the performance context. This is much more efficient (even more if Loudness phrases are cascaded) than modifying a performance afterwards. Now the performance is no longer generated as-is, but it is enclosed in a state monad, that manages the `Performance.Context`. The `PhraseFuns` are now of type `Performance.PState -> Performance.PState` which is both clean and efficient.

Phrase The original version of Haskore used a list of `PhraseAttributes` for the `Phrase` constructor. Now it allows only one attribute in order to make the order of application transparent to the user.

Type of `Music.Dur` Durations are represented as rational numbers; specifically, as ratios of two Haskell `Integer` values. Previous versions of Haskore used floating-point numbers, but rational numbers are more precise and allow quick-checking of music composition properties.

References

[AK92] D.P. Anderson and R. Kuivila. Formula: A programming language for expressive computer music. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[Ary94] K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.

[Bal92] M. Balaban. Music structures: Interleaving the temporal and hierarchical aspects of music. In M. Balaban, K. Ebcioglu, and O. Laske, editors, *Understanding Music With AI*, pages 110–139. AAAI Press, 1992.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.

[Col84] D. Collinge. Moxie: A languge for computer music performance. In *Proc. Int'l Computer Music Conference*, pages 217–220. Computer Music Association, 1984.

[CR84] P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programmming*, pages 85–95. ACM, 1984.

[Dan89] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[DFV92] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[For73] A. Forte. *The Structure of Atonal Music*. Yale University Press, New Haven, CT, 1973.

[HB95] P. Hudak and J. Berger. A model of performance, interaction, and improvisation. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1995.

[Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programmming*. ACM, 1982.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMGW96] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996. available via `ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore/hmn-lhs.ps`.

[HS92] G. Haus and A. Sametti. Scoresynth: A system for the synthesis of music scores based on petri nets and a music algebra. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[IMA90] Midi 1.0 detailed specification: Document version 4.1.1, February 1990.

[JB91] D. Jaffe and L. Boynton. An overview of the sound and music kits for the NeXT computer. In S.T. Pope, editor, *The Well-Tempered Object*, pages 107–118. MIT Press, 1991.

[OFLB94] O. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1994.

[Sch83] B. Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983.

[Ver86] B. Vercoe. Csound: A manual for the audio processing system and supporting programs. Technical report, MIT Media Lab, 1986.

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the definition; numbers in roman refer to the pages where the entry is used.