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 and 2005 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* [**?**] before proceeding. I also assume some familiarity with *equational reasoning*; an excellent introductory text on this is [**?**].

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.

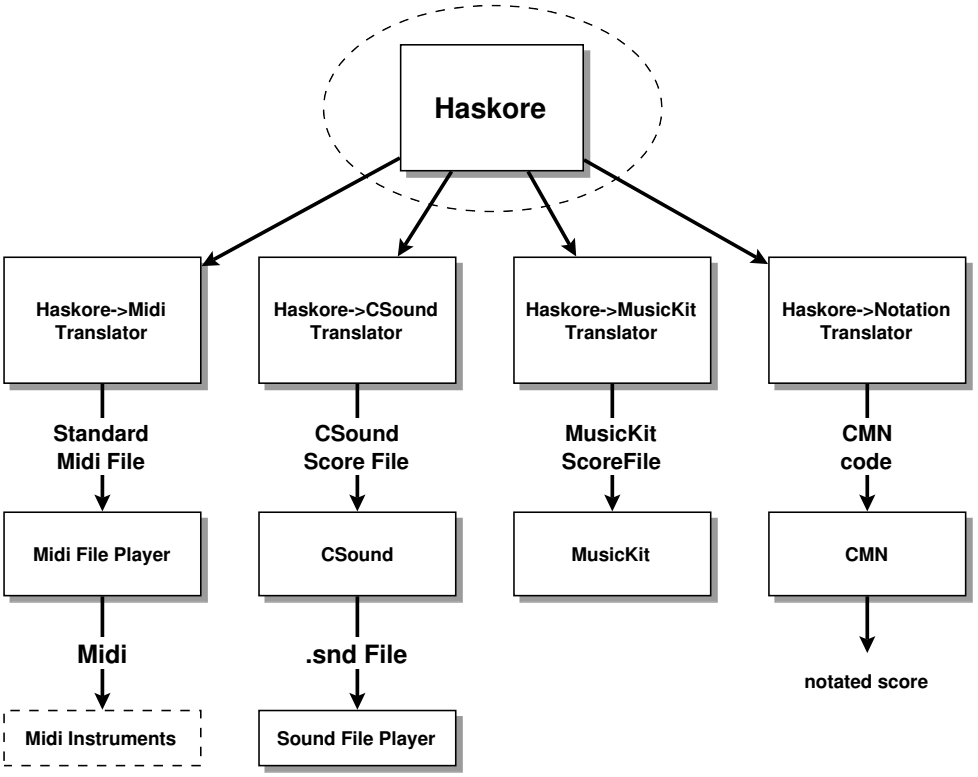


Figure 1: Overall System Diagram

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 [**?**], NeXT MusicKit score files [**?**] [1], and CSound score files [**?**] [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,
                module Music,
                module Performance,
```

[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 Player,
                module Haskore.Interface.MIDI.Write,
                module Haskore.Interface.MIDI.Read,
                module Haskore.Interface.MIDI.Save,
                module Haskore.Interface.MIDI.Load,
                module Haskore.Interface.MIDI.Render)
        where

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

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 [**?**]; it is the "Haskore in practice" version alluded to in Section 6.1 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 Composing Music

3.1 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
```

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.

```
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
     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?
```

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`.

3.2 Music

```
module Haskore.Music where

import Haskore.General.Utility (pairMap)
import qualified Data.List as List
import Data.Ratio ((%))

import qualified Media.Temporal
import qualified Media
import qualified Media.List
import Media (prim, serial, parallel)
import qualified Haskore.Basic.Pitch as Pitch
import Haskore.Basic.Pitch hiding (T)
```

Melodies consist essentially of the musical atoms notes and rests.

```
data Atom = Rest                           -- a rest
          | Note Pitch.T [NoteAttribute] -- a note
                                           -- a percussion?
    deriving (Show, Eq, Ord)
```

Here a `Note` is its pitch paired with its duration (in number of whole notes), along with a list of `NoteAttributes` (defined later). A `Rest` also has a duration, but of course no pitch or other attributes.

Note that durations are represented as rational numbers; specifically, as ratios of two Haskore `Int` values. Previous versions of Haskore used floating-point numbers, but rational numbers are more precise (as long as the `Int` values do not exceed the maximum allowable).

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

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

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

data Primitive =
         Atom Dur Atom              -- atomic object
       | Control Control T          -- control a sub-structure
    deriving (Show, Eq, Ord)

type T = Media.List.T Primitive

type Dur        = Media.Temporal.Dur
type DurRatio   = Dur
type IName      = String
type PlayerName = String

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

mkControl :: (a -> Control) -> (a -> T -> T)
mkControl ctrl x m = control (ctrl x) m
changeTempo :: DurRatio -> T -> T
changeTempo = mkControl Tempo
transpose :: Pitch.Relative -> T -> T
transpose = mkControl Transpose
setInstrument :: IName -> T -> T
setInstrument = mkControl Instrument
setPlayer :: PlayerName -> T -> T
setPlayer = mkControl Player
phrase :: PhraseAttribute -> T -> T
phrase = mkControl Phrase

infixr 7 +:+  {- like multiplication -}
infixr 6 =:=  {- like addition -}
-- make them visible for importers of Music
(+:+), (=:=) :: T -> T -> T
(+:+) = (Media.+:+)
(=:=) = (Media.=:=)
```

```
note :: Pitch.T -> Dur -> [NoteAttribute] -> T
note p d' nas = prim (Atom d' (Note p nas))

note' :: Pitch.Class -> Pitch.Octave ->
           Dur -> [NoteAttribute] -> T
note' = flip (curry note)

cf,c,cs,df,d,ds,ef,e,es,ff,f,fs,gf,g,gs,af,a,as,bf,b,bs ::
   Pitch.Octave -> Dur -> [NoteAttribute] -> T

cf = note' Cf;  c = note' C;  cs = note' Cs
df = note' Df;  d = note' D;  ds = note' Ds
ef = note' Ef;  e = note' E;  es = note' Es
ff = note' Ff;  f = note' F;  fs = note' Fs
gf = note' Gf;  g = note' G;  gs = note' Gs
af = note' Af;  a = note' A;  as = note' As
bf = note' Bf;  b = note' B;  bs = note' Bs
```

Figure 3: Convenient note construction functions.

- `serial ms` is the sequential composition of the elements of the list `ms`; e.g. `Serial [m1, m2]` means that `m1` and `m2` are played in sequence. (cf. Section C)
- `parallel ms` is the parallel composition of the elements of the list `ms`; e.g. `Parallel [m1, m2]` means that `m1` and `m2` are played simultaneously.
- `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).
- `setInstrument iname m` declares that `m` is to be performed using instrument `iname`.
- `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)

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.

3.3 Convenient Auxiliary Functions

For convenience, let's create simple names for familiar notes (Figure 3), durations, and rests (Figure 4). Despite the large number of them, these names are sufficiently "unusual" that name clashes are unlikely.

```
rest :: Dur -> T
rest d' = prim (Atom d' Rest)

dotted, doubleDotted :: Dur -> Dur
dotted       = ((3%2) *)
doubleDotted = ((7%4) *)

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

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

bn   = 2     ; bnr   = rest bn     -- brevis rest
wn   = 1     ; wnr   = rest wn     -- whole note rest
hn   = 1% 2  ; hnr   = rest hn     -- half note rest
qn   = 1% 4  ; qnr   = rest qn     -- quarter note rest
en   = 1% 8  ; enr   = rest en     -- eight note rest
sn   = 1%16  ; snr   = rest sn     -- sixteenth note rest
tn   = 1%32  ; tnr   = rest tn     -- thirty-second note rest
sfn  = 1%64  ; sfnr  = rest sfn    -- sixty-fourth note rest

dwn  = dotted wn ; dwnr  = rest dwn    -- dotted whole note rest
dhn  = dotted hn ; dhnr  = rest dhn    -- dotted half note rest
dqn  = dotted qn ; dqnr  = rest dqn    -- dotted quarter note rest
den  = dotted en ; denr  = rest den    -- dotted eighth note rest
dsn  = dotted sn ; dsnr  = rest dsn    -- dotted sixteenth note rest
dtn  = dotted tn ; dtnr  = rest dtn    -- dotted thirty-second note rest

ddhn = doubleDotted hn ; ddhnr = rest ddhn   -- double-dotted half note rest
ddqn = doubleDotted qn ; ddqnr = rest ddqn   -- double-dotted quarter note rest
dden = doubleDotted en ; ddenr = rest dden   -- double-dotted eighth note rest
```

Figure 4: Convenient duration and rest definitions.

3.4 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] -> T
line  = serial
chord = parallel
```

From the notes in the C major triad in register 4, I can now construct a C major arpeggio and chord as well:

```
cMaj :: [T]
cMaj = [ n 4 qn [] | n <- [c,e,g] ]  -- octave 4, quarter notes

cMajArp, cMajChd :: T
cMajArp = line  cMaj
cMajChd = chord cMaj
```

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 -> T
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 -> T
repeat m = serial (List.repeat m)
```

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.

Inversion and Retrograde. The notions of inversion, retrograde, retrograde inversion, etc. used in 12-tone theory are also easily captured in Haskore. First let's define a transformation from a line created by `line` to a list:

```
invertAtom :: Pitch.T -> Atom -> Atom
invertAtom r (Note p nas) =
   Note (Pitch.fromInt (2 * Pitch.toInt r - Pitch.toInt p)) nas
invertAtom _ (Rest)       = Rest

retro, invert, retroInvert, invertRetro :: [(d,Atom)] -> [(d,Atom)]
```

```
retro    = List.reverse
invert l = let h@((_, Note r _) : _) = l
               inv (d', at) = (d', invertAtom r at)
           in  map inv h
retroInvert = retro  . invert
invertRetro = invert . retro
```

3 Exercise. *Show that* "`retro . retro`", "`invert . invert`", *and* "`retroInvert . invertRetro`" *are the identity on values created by* `line`.

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

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

instance Media.Temporal.Class Primitive where
   dur (Atom d' _)           = d'
   dur (Control (Tempo t) m) = dur m / t
   dur (Control  _        m) = dur m
   none d' = Atom d' Rest
```

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 -> T
reverse = mapList
   (curry id)
   (curry id)
   List.reverse
   (\ms -> let durs = map dur ms
               dmax = maximum durs
           in  zipWith (delay . (dmax -)) durs ms)
```

Truncating Parallel Composition Note that the duration of `m1 =:= m2` is the maximum of the durations of
codem1 and `m2` (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 -> T
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] -> [T]
takeLine newDur = snd . takeLine' newDur

take' :: Dur -> T -> (Dur, T)
take' newDur m =
   if newDur == 0
   then (0, rest 0)
   else foldListFlat
      (\oldDur at -> let takenDur = min oldDur newDur
                     in (takenDur, atom takenDur at))
      (\ctrl m' -> case ctrl of
          Tempo t -> pairMap ((/t), changeTempo t)
                             (take' (newDur * t) m')
          _       -> pairMap (id,   control ctrl)
                             (take' newDur m'))
      (\ms -> pairMap (id,line) (takeLine' newDur ms))
      (\ms -> pairMap (maximum,chord) (unzip (map (take' newDur) ms)))
      m

takeLine' :: Dur -> [T] -> (Dur, [T])
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 -> T -> T
```

```
m1 /=: m2 = Haskore.Music.take (min (dur m1) (dur m2)) (m1 =:= m2)
```

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

4 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.

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

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

drop' :: Dur -> T -> (Dur, T)
drop' remDur m =
   if remDur == 0
   then (0, m)
   else foldListFlat
      (\oldDur _ -> let newDur = min oldDur remDur
                    in (newDur, rest (oldDur-newDur)))
      (\ctrl m' -> case ctrl of
          Tempo t -> pairMap ((/t), changeTempo t)
                             (drop' (remDur * t) m')
          _       -> pairMap (id, control ctrl)
                             (drop' remDur m'))
      (\ms -> pairMap (id,line) (dropLine' remDur ms))
      (\ms -> pairMap (maximum,chord) (unzip (map (drop' remDur) ms)))
      m

dropLine' :: Dur -> [T] -> (Dur, [T])
dropLine' 0 m  = (0, m)
dropLine' remDur [] = (remDur, [])
dropLine' remDur (m:ms) =
   let (dropped, m') = drop' remDur m
   in  if dropped < remDur
       then pairMap ((dropped+), id) (dropLine' (remDur - dropped) ms)
       else (remDur, if m' == rest 0 then ms else m' : ms)
```

In `pairMap` we use `fst` and `snd`, in order to make sure that it also works if one of the arguments is an infinite list.

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

```
foldBinFlat :: (Dur -> Atom -> b)
   -> (Control -> T -> b)
   -> (T -> T -> b)
   -> (T -> T -> b)
   -> b -> T -> b
foldBinFlat fa fc = Media.foldBinFlat (foldPrim fa fc)

foldListFlat :: (Dur -> Atom -> b)
   -> (Control -> T -> b)
   -> ([T] -> b)
   -> ([T] -> b)
   -> T -> b
foldListFlat fa fc = Media.foldListFlat (foldPrim fa fc)

foldPrim :: (Dur -> Atom -> b) -> (Control -> T -> b) -> Primitive -> b
foldPrim fa fc pr =
   case pr of
      Atom    d'   at -> fa d' at
      Control ctrl m  -> fc ctrl m

foldList :: (Dur -> Atom -> b)
   -> (Control -> b -> b)
   -> ([b] -> b)
   -> ([b] -> b)
   -> T -> b
foldList fa fc fser fpar = Media.foldList
   (\pr -> case pr of
      Atom    d'   at -> fa d' at
      Control ctrl m  -> fc ctrl (foldList fa fc fser fpar m))
   fser fpar


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

mapListFlat fa fc = Media.mapListFlat
   (\pr -> case pr of
      Atom    d'   at -> uncurry Atom (fa d' at)
      Control ctrl m  -> uncurry Control (fc ctrl m))
```

```
mapList fa fc fser fpar = Media.mapList
   (\pr -> case pr of
      Atom    d'   at -> uncurry Atom (fa d' at)
      Control ctrl m  -> uncurry Control (fc ctrl (mapList fa fc fser fpar m)))
   fser fpar
```

3.5 Trills

```
module Haskore.Basic.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 -> Music.T
```

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 -> Music.T
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 -> Music.T
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 -> Music.T
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 -> Music.T
rolln :: Integer   -> Music.T -> Music.T

roll  d      = trill  0 d
rolln nTimes = trilln 0 nTimes
```

3.6 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 conveneient 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.

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 5, 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. So all we need is a way to convert these percussion sound names into a `Music.T` object; i.e. a `Note`:

```
toMusic :: T -> Dur -> [NoteAttribute] -> Music.T
toMusic ds =
   note (Pitch.fromInt (fromEnum ds + 35 - MidiFile.zeroKey))
```

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

```
funkGroove :: Music.T
funkGroove =
    let p1 = toMusic LowTom        qn []
        p2 = toMusic AcousticSnare en []
    in changeTempo 3 (setInstrument "Drums" (Music.take 8 (Music.repeat
         ( (Music.line [p1, qnr, p2, qnr, p2,
                        p1, p1, qnr, p2, enr])
           =:= roll en (toMusic ClosedHiHat 2 []) )
       )))
```

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

```
module Haskore.Basic.Drum
   (T(..), toMusic, lineToMusic, elementToMusic, funkGroove) where

import Data.Ix(Ix)

import Haskore.Basic.Trill
import Haskore.Music
             (NoteAttribute, Dur, qn, en, qnr, enr, (=:=),
              changeTempo, setInstrument,
              note, rest, line)

import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Music as Music
import qualified Haskore.Interface.MIDI.File         as MidiFile

data T =
       AcousticBassDrum  -- Midi Key 35
     | BassDrum1         -- Midi Key 36
     | SideStick         -- ...
     | AcousticSnare | HandClap      | ElectricSnare | LowFloorTom
     | ClosedHiHat   | HighFloorTom  | PedalHiHat    | LowTom
     | OpenHiHat     | LowMidTom     | HiMidTom      | CrashCymbal1
     | HighTom       | RideCymbal1   | ChineseCymbal | RideBell
     | Tambourine    | SplashCymbal  | Cowbell       | CrashCymbal2
     | Vibraslap     | RideCymbal2   | HiBongo       | LowBongo
     | MuteHiConga   | OpenHiConga   | LowConga      | HighTimbale
     | LowTimbale    | HighAgogo     | LowAgogo      | Cabasa
     | Maracas       | ShortWhistle  | LongWhistle   | ShortGuiro
     | LongGuiro     | Claves        | HiWoodBlock   | LowWoodBlock
     | MuteCuica     | OpenCuica     | MuteTriangle
     | OpenTriangle      -- Midi Key 82
   deriving (Show,Eq,Ord,Ix,Enum)
```

Figure 5: General Midi Percussion Names

```
data Element =
     N             Dur [NoteAttribute] -- note
   | R             Dur                 -- rest
   | Roll  Dur     Dur [NoteAttribute] -- roll w/duration
   | Rolln Integer Dur [NoteAttribute] -- roll w/number of strokes

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

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

3.7 Phrasing and Articulation

Recall that the `Note` constructor contained a field of `NoteAttribute`s. These are values that are attached to notes for the purpose of notation or musical interpretation. Likewise, the `Phrase` constructor permits one to annotate an entire musical object with a `PhraseAttribute`. These two attribute datatypes cover a wide range of attributions found in common practice notation, and are shown in Figure 6. 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 5.

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

```
accent, crescendo, diminuendo, loudness,
  ritardando, accelerando,
  staccato, legato :: Float -> T -> T

accent     = phrase . Dyn . Accent
crescendo  = phrase . Dyn . Crescendo
diminuendo = phrase . Dyn . Diminuendo
loudness   = phrase . Dyn . Loudness

ritardando  = phrase . Tmp . Ritardando
accelerando = phrase . Tmp . Accelerando

staccato = phrase . Art . Staccato
legato   = phrase . Art . Legato
```

```
data NoteAttribute = Velocity Float -- intensity of playing between 0 and 1
                   | Fingering Int
                   | Dynamics String
                   | PFields [Float]
     deriving (Eq, Ord, Show)

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

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

data StdLoudness = PPP | PP | P | MP | SF | MF | NF | FF | FFF
     deriving (Eq, Ord, Show, Enum)

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

data Articulation = Staccato Float | Legato Float | Slurred Float
                  | 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)

data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
              | TremoloHead | SlashHead | ArtHarmonic | NoHead
     deriving (Eq, Ord, Show)
```

Figure 6: Note and Phrase Attributes.

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, we would expect `Legato 1.2` to create more of a legato feel than `Legato 1.1`. The following constants represent default values for some of the parameterized attributes:

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

defltLegato    = legato 1.1
defltStaccato  = staccato 0.5
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 6 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).

5 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.

3.8 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
```

3.9 Representing 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 "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:

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

```
module Haskore.Basic.Chord where

import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Basic.Interval as I
import qualified Haskore.Music as Music

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 ch = aux ch
   where aux (n1:n2:ns) = (n2-n1) : aux (n2:ns)
         aux [n]        = [head ch - n]
         aux _          = error "pitchToInterval: Chord must be non-empty."

intervalToPitch :: T -> T
intervalToPitch ch = 0 : aux 0 ch
   where aux _ [_]    = []
         aux p (n:ns) = n' : aux n' ns  where n' = p+n
         aux _ _      = error "intervalToPitch: Chord must be non-empty."
```

6 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]`).

We want to close the section with 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]


makeChord :: [Pitch.Relative] -> Music.T -> [Music.T]
makeChord int m = map (flip Music.transpose m) int

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

major = makeChord majorInt
minor = makeChord minorInt

majorSeventh      = makeChord majorSeventhInt
minorSeventh      = makeChord minorSeventhInt
dominantSeventh   = makeChord dominantSeventhInt
minorMajorSeventh = makeChord minorMajorSeventhInt
```

```
sustainedFourth = makeChord sustainedFourthInt
```

3.10 Tempo

```
module Haskore.Basic.Tempo where

import           Data.Ratio((%))
import qualified Data.List as List
import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Music as Music
import Haskore.Music(changeTempo, qn, en, sn, line, (+:+), (=:=))
```

Set tempo. To make it easier to initialize the duration element of a `PerformanceContext.T` (see Section 4), 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 * fromRational dur)
```

Additionally we define some common tempos and some range of interpretation as in Figure 7. 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 8. They can be expressed quite naturally in Haskore as follows (note the use of the `where` clause in `pr2` to capture recurring phrases):

```
pr1, pr2 :: Pitch.T -> Music.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)
```

```
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 7: Common names for tempo.

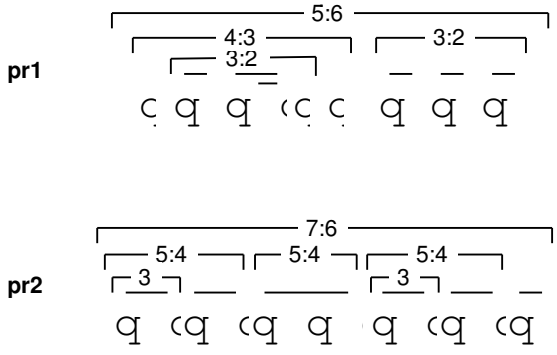


Figure 8: Nested Polyrhythms

```
      (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 -> Music.T
mkLn n p d = line (take n (List.repeat (Music.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 :: Music.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 -> Music.T
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.

4 Interpretation and Performance

```
module Haskore.Music.Performance where

import Haskore.Music
          (IName, PlayerName, NoteAttribute, PhraseAttribute)
import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Music as Music
import Data.List(unfoldr)
```

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 = [Event]

data Event = Event {eTime :: Time, eInst :: IName,    ePitch  :: Pitch.Absolute,
                    eDur  :: Dur,  eVel  :: Velocity, pFields :: [Float]}
     deriving (Eq,Ord,Show)

type Time      = Float
```

```
type Dur       = Float
type Volume    = Float
type Velocity  = Float -- between 0 and 2, default is 1
```

An event is the lowest of our music representations not yet committed to Midi, CSound, or the MusicKit. An event `Event {eTime = s, eInst = i, ePitch = p, eDur = d, eVel = v}` captures the fact that `s` seconds after the preceding event (cf. Section C) instrument `i` starts sounding with pitch `p` and velocity `v` for a duration `d` (where now duration is measured in seconds, rather than beats).

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 *players* to use; that is, we need a mapping from the `PlayerName`s in an abstract musical object to the actual players to be used. (We don't yet need a mapping from abstract `IName`s to instruments, since this is handled in the translation from a performance into, say, Midi, such as defined in Section 6.1.)

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

```
fromMusic :: PlayerMap -> Context -> Music.T -> T

type PlayerMap    = PlayerName -> PlayerT
data Context =
   Context {cTime :: Time, cPlayer :: PlayerT, cInstrument :: IName,
            cDur  :: Dur,  cKey    :: Key,     cVelocity   :: Velocity}
      deriving Show

type UpdateContext a = (a -> a) -> Context -> Context

updateTime :: UpdateContext Time
updateTime f c = c {cTime = f (cTime c)}
updatePlayer :: UpdateContext PlayerT
updatePlayer f c = c {cPlayer = f (cPlayer c)}
updateInstrument :: UpdateContext IName
updateInstrument f c = c {cInstrument = f (cInstrument c)}
updateDur :: UpdateContext Dur
updateDur f c = c {cDur = f (cDur c)}
updateKey :: UpdateContext Key
updateKey f c = c {cKey = f (cKey c)}
updateVelocity :: UpdateContext Velocity
updateVelocity f c = c {cVelocity = f (cVelocity c)}

type Key = Pitch.Absolute

romMusic pmap c@Context {cTime = t, cPlayer = pl, cDur = dt, cKey = k} m =
case m of
   Note p d nas    -> playNote pl c p d nas
   Rest d          -> []
```

```
   m1 :+: m2       -> fromMusic pmap c m1 ++
                      fromMusic pmap (c {cTime = t + dur m1 * dt}) m2
   m1 :=: m2       -> merge (fromMusic pmap c m1) (fromMusic pmap c m2)
   Tempo  a   m    -> fromMusic pmap (c {cDur = dt / fromRational a}) m
   Transpose  p  m -> fromMusic pmap (c {cKey = k + p}) m
   Instrument nm m -> fromMusic pmap (c {cInst = nm}) m
   Player nm  m    -> fromMusic pmap (c {cPlayer = pmap nm}) m
   Phrase pas m    -> interpPhrase pl pmap c pas m
```

Some things to note:

1. 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 `dt` appropriately and updating the `Dur` field of the context.

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

2. 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 `interpPhrase`. Precisely how these two functions work is described in Section 5.

3. The `Dur` component of the context is the duration, in seconds, of one whole note. See Section 3.10 for assisting functions.

4. 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 3.4) 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 9.

5. In contrast, the sub-sequences derived from the arguments to `Parallel` are merged into a time-ordered stream. The definition of `merge` is given below.

```
merge, mergeFirst :: (T, Dur) -> (T, Dur) -> (T, Dur)

{- merge two performances provided that e0 is earlier than e1 -}
mergeFirst (e0:es0, ld0) (evs1, ld1) =
   let (es, ld) = merge (es0, ld0)
          (case evs1 of
             e1:es1 -> (e1{eTime = eTime e1 - eTime e0} : es1, ld1)
             []     -> ([], max 0 (ld1 - eTime e0)))
   in  (e0 : es, ld)
mergeFirst _ _ = error "mergeFirst should never fail this way."
```

```
fromMusic pmap c m = events (fromMusic' pmap c m)

{- lastDur is 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 one remains playing. -}

data T' = C' {events :: T,
              duration, lastDur :: Dur}
   deriving (Show, Eq)

fromMusic' :: PlayerMap -> Context -> Music.T -> T'

fromMusic' pmap c@Context {cTime = t, cPlayer = pl, cDur = dt, cKey = k} =
   Music.foldBinFlat
      (\d at ->
         let noteDur = fromRational d * dt
         in case at of
               Music.Note p nas -> C' (playNote pl c p d nas) noteDur noteDur
               Music.Rest       -> C' [] noteDur (t + noteDur))
      (\ctrl m ->
         case ctrl of
            Music.Tempo      a  -> fromMusic' pmap (c {cDur = dt / fromRational a})
            Music.Transpose  p  -> fromMusic' pmap (c {cKey = k + p}) m
            Music.Instrument nm -> fromMusic' pmap (c {cInstrument = nm}) m
            Music.Player     nm -> fromMusic' pmap (c {cPlayer = pmap nm}) m
            Music.Phrase     pa -> interpPhrase pl pmap c pa m)
      {- I didn't succeed with a formulation using fold or similar functions.
         foldl works in principle but fails on infinite streams. -}
      (\m0 m1 -> let C' ev0 d0 ld0 = fromMusic' pmap  c                m0
                     C' ev1 d1 ld1 = fromMusic' pmap (c {cTime = ld0}) m1
                 in  C' (ev0 ++ ev1) (d0 + d1) ld1)
      (\m0 m1 -> let C' ev0 d0 ld0 = fromMusic' pmap c m0
                     C' ev1 d1 ld1 = fromMusic' pmap c m1
                     (ev, ld) = merge (ev0, ld0) (ev1, ld1)
                 in  C' ev (max d0 d1) ld)
      (C' [] 0 t)
```

Figure 9: The "real" `fromMusic` function.

```
merge pf0@(e0:_, _) pf1@(e1:_, _) =
   if e0 <= e1 then mergeFirst pf0 pf1
               else mergeFirst pf1 pf0
merge ([], ld0) ([], ld1) = ([], max ld0 ld1)
merge pf0@([], _) pf1 = mergeFirst pf1 pf0
merge pf0 pf1@([], _) = mergeFirst pf0 pf1
```

Note that `merge` compares entire events rather than just start times. This is to ensure that it is commutative, a desirable condition for some of the proofs used in Section 4.1. It is also necessary to assert a unique representation of the performance independent of the structure of the `Music.T`.

The function `partition` is somehow the inverse to `merge`. It is similar to `List.partition`. We could use the List function if the event times were absolute, because then the events need not to be altered on splits. But absolute time points can't be used for infinite music thus we take the burden of adapting the time differences when an event is removed from the performance list and put to the list of events of a particular instrument. `t0` is the time gone since the last event in the first partition, `t1` is the time gone since the last event in the second partition.

Note, that we must use `fst esp` and `snd esp` in the definition of `ins0` and `ins1`. If we declare `ins0 (es0,es1) =` … then the run-time system would wait until it is sure that `partition` returns a pair and not Bottom.

```
partition :: (Event -> Bool) -> Time -> Time ->
                 [Event] -> ([Event], [Event])
partition _ _  _  [] = ([], [])
partition p t0 t1 (e:es) =
   let t0' = t0 + eTime e
       t1' = t1 + eTime e
       ins0 esp = (e {eTime = t0'} : fst esp, snd esp)
       ins1 esp = (fst esp, e {eTime = t1'} : snd esp)
   in  if p e
       then ins0 (partition p 0 t1' es)
       else ins1 (partition p t0' 0 es)
```

Since we need it later for MIDI generation, we will also define a slicing into equivalence classes of events.

```
slice :: Eq a => (Event -> a) -> T -> [(a, T)]
slice f perf =
  let splitByHeadInst [] = Nothing
      splitByHeadInst pf =
         let i = f (head pf)
             (pf0, pf1) = partition ((i==) . f) 0 0 pf
         in  Just ((i,pf0), pf1)
  in  unfoldr splitByHeadInst perf

usedInstruments :: T -> [Music.IName]
usedInstruments = map fst . slice eInst
```

4.1 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:

```
(m1 :+: m2) :+: (m3 :+: m4)
  m1 :+: m2 :+: m3 :+: m4
```

Thus we define a notion of equivalence:

7 Definition. Two musical objects `m1` and `m2` are *equivalent*, written `m1` ≡ `m2`, if and only if:

```
(∀imap,c)  Pf.fromMusic imap c m1 = Pf.fromMusic imap c m2
```

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 7.1) 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:

8 Axiom. *For any* `r1`, `r2`, *and* `m`*:*

```
Tempo r1 (Tempo r2 m) ≡ Tempo (r1*r2) 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 (Tempo r1 (Tempo r2 m))
= Pf.fromMusic (dt / r1) (Tempo r2 m)          -- unfolding Pf.fromMusic
= Pf.fromMusic ((dt / r1) / r2) m              -- unfolding Pf.fromMusic
= Pf.fromMusic (dt / (r1 * r2)) m              -- simple arithmetic
= Pf.fromMusic dt (Tempo (r1*r2) 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)`):

9 Axiom. *For any* `r`, `m1`, *and* `m2`*:*

```
Tempo r (m1 :+: m2) ≡ Tempo r m1 :+: Tempo r m2
```

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

Proof.

```
Pf.fromMusic (t,dt) (Tempo r (m1 :+: m2))
= Pf.fromMusic (t,dt/r) (m1 :+: m2)           -- unfolding Pf.fromMusic
= Pf.fromMusic (t,dt/r) m1 ++
          Pf.fromMusic (t',dt/r) m2           -- unfolding Pf.fromMusic
= Pf.fromMusic (t,dt) (Tempo r m1) ++
          Pf.fromMusic (t',dt) (Tempo r m2)   -- folding Pf.fromMusic
              where t'  = t + dur m1 * dt/r
= Pf.fromMusic (t,dt) (Tempo r m1) ++
          Pf.fromMusic (t'',dt) (Tempo r m2)  -- folding dur
              where t'' = t + dur (Tempo r m1) * dt
= Pf.fromMusic (t,dt)
               (Tempo r m1 :+: Tempo r m2)    -- folding Pf.fromMusic
```

□

An even simpler axiom is given by:

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

```
Tempo 1 m ≡ m
```

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

Proof.

```
Pf.fromMusic (t,dt) (Tempo 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:

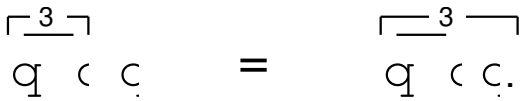


Figure 10: Equivalent Phrases

11 Theorem. *For any* `r`, `m1`, *and* `m2`*:*

```
Tempo r m1 :+: m2 ≡ Tempo r (m1 :+: Tempo (recip r) m2)
```

Proof.

```
Tempo r (m1 :+: Tempo (recip r) m2)
= Tempo r m1 :+: Tempo r (Tempo (recip r) m2)     -- by Axiom 1
= Tempo r m1 :+: Tempo (r * recip r) m2           -- by Axiom 2
= Tempo r m1 :+: Tempo 1 m2                       -- simple arithmetic
= Tempo r m1 :+: m2                               -- by Axiom 3
```

□

For example, this fact justifies the equivalence of the two phrases shown in Figure 10.

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).

12 Axiom. `Tempo` *is* multiplicative *and* `Transpose` *is* additive*. That is, for any* `r1`, `r2`, `p1`, `p2`, *and* `m`*:*

```
Tempo r1 (Tempo r2 m) ≡ Tempo (r1*r2) m
Trans p1 (Trans p2 m) ≡ Trans (p1+p2) m
```

13 Axiom. *Function composition is* commutative *with respect to both tempo scaling and transposition. That is, for any* `r1`, `r2`, `p1` *and* `p2`*:*

```
Tempo r1 . Tempo r2 ≡ Tempo r2 . Tempo r1
Trans p1 . Trans p2 ≡ Trans p2 . Trans p1
Tempo r1 . Trans p1 ≡ Trans p1 . Tempo r1
```

14 Axiom. *Tempo scaling and transposition are* distributive *over both sequential and parallel composition. That is, for any* `r`, `p`, `m1`, *and* `m2`*:*

```
Tempo r (m1 :+: m2) ≡ Tempo r m1 :+: Tempo r m2
Tempo r (m1 :=: m2) ≡ Tempo r m1 :=: Tempo r m2
Trans p (m1 :+: m2) ≡ Trans p m1 :+: Trans p m2
Trans p (m1 :=: m2) ≡ Trans p m1 :=: Trans p m2
```

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
```

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

```
m0 :=: m1 ≡ m1 :=: m0
```

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

```
   Tempo r (Rest 0) ≡ Rest 0
   Trans p (Rest 0) ≡ Rest 0
m :+: Rest 0 ≡ m ≡ Rest 0 :+: m
m :=: Rest 0 ≡ m ≡ Rest 0 :=: m
```

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

5 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 3.7) 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 PlayerT = PlayerC { name         :: PlayerName,
                         playNote     :: NoteFun,
                         interpPhrase :: PhraseFun,
                         notatePlayer :: NotateFun }

instance Show PlayerT where
   show p = "Player.c " ++ name p

type NoteFun   =
     Context -> Pitch.T -> Music.Dur -> [NoteAttribute] -> T
```

```
type PhraseFun =
     PlayerMap -> Context -> PhraseAttribute -> Music.T -> T'
type NotateFun = ()
```

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

6 Interfaces to other musical software

6.1 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 "Basic Midi". The most important aspect of Basic Midi is that Channel 10 (9 in Haskore's 0-based numbering) is dedicated to *percussion*. A future release of Haskore should make these distinctions more concrete.

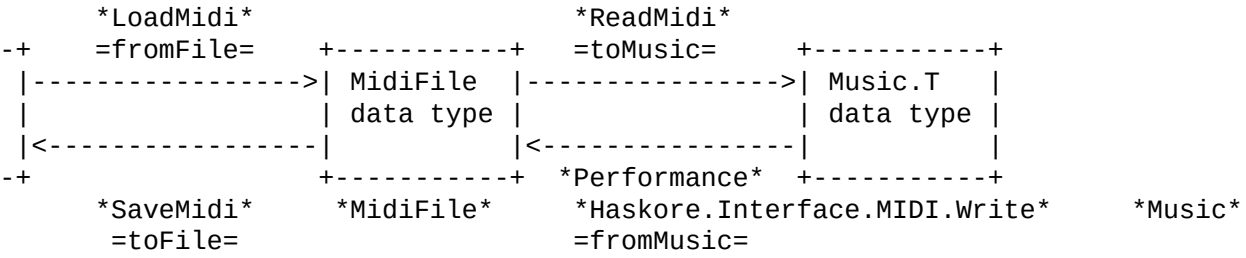
Haskore provides a way to specify a Midi channel number and General Midi instrument selection for each `IName` in a Haskore composition. It also provides a means to generate a Standard Midi File, which can then be played using any conventional Midi software. Finally, it provides a way for existing Midi files to be read and converted into a `Music.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
                 (fromPerformance, fromPerformanceGM,
```

```
                  fromPerformanceMixed, fromPerformanceMixedGM,
                  fromMusic, fromMusicGM,
                  fromMusicMixed, fromMusicMixedGM,
                  volumeHaskoreToMIDI, volumeMIDIToHaskore)
      where

import Haskore.General.Utility(roundDiff)
import Data.FiniteMap(lookupFM, addToFM, emptyFM)
import Control.Monad.State(evalState,mapState,zipWithM)
import qualified Control.Monad.State as State

import Haskore.Music.Performance(Event, eTime, ePitch, eDur, eVel, eInst)
import qualified Haskore.Interface.MIDI.File         as MidiFile
import qualified Haskore.Interface.MIDI.UserPatchMap as UserPatchMap

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

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 key modules (*) and functions (=):



Given a `UserPatchMap.T` (Section 6.1.2), a performance is converted to a datatype representing a Standard Midi File of type 1 (tracks played simultaneously) using the `fromPerformance` function. In contrast to that, the function `fromPerformanceGM` creates a new `UserPatchMap.T` by matching the instrument names with General Midi names and mapping the instruments to channels one by one.

The "`Mixed`" functions create files of MIDI type 0 that is there is one track containing the whole music. This is the only mode which can be used 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.

```
fromPerformance, fromPerformanceMixed ::
   UserPatchMap.T -> Performance.T -> MidiFile.T
```

```
fromPerformanceGM, fromPerformanceMixedGM ::
   Performance.T -> MidiFile.T

fromPerformance = fromPerformanceBase . const

fromPerformanceGM = fromPerformanceBase (UserPatchMap.fromInstruments . map fst)

fromPerformanceBase makePMap pf =
  let splitList   = Performance.slice eInst pf
      pMap        = makePMap splitList
      lookupIName = UserPatchMap.lookup pMap
  in  MidiFile.C MidiFile.Parallel (MidiFile.Ticks division)
         (zipWith performToMEvs
                  (map (const . lookupIName . fst) splitList)
                  (map snd splitList))


fromPerformanceBase, fromPerformanceMixedBase ::
   ([(Music.IName, Performance.T)] -> UserPatchMap.T)
      -> Performance.T -> MidiFile.T

fromPerformanceMixed =
   fromPerformanceMixedBase . const

fromPerformanceMixedGM =
   fromPerformanceMixedBase (UserPatchMap.fromInstruments . map fst)

fromPerformanceMixedBase makePMap pf =
  let pMap = makePMap (Performance.slice eInst pf)
  in  MidiFile.C MidiFile.Mixed (MidiFile.Ticks division)
         [performToMEvs (UserPatchMap.lookup pMap) pf]
```

The more comfortable function `fromMusic` turns a `Music.T` immediately into a `MidiFile.T`. Thus it needs also a `Context` and a `UserPatchMap.T`. The signature is chosen so that it can be used as an inverse to `ReadMidi.toMusic`. The function `fromMusicGM` is similar but doesn't need a `UserPatchMap.T` because it creates one from the set of instruments actually used in the `Music.T`.

```
fromMusic, fromMusicMixed ::
   (UserPatchMap.T, Context.T, Music.T) -> MidiFile.T

fromMusicGM, fromMusicMixedGM ::
   (Context.T, Music.T) -> MidiFile.T

fromMusic      (upm,c,m) = fromMusicBase (fromPerformance upm)      c m
fromMusicGM        (c,m) = fromMusicBase  fromPerformanceGM         c m
fromMusicMixed (upm,c,m) = fromMusicBase (fromPerformanceMixed upm) c m
```

```
fromMusicMixedGM   (c,m) = fromMusicBase  fromPerformanceMixedGM    c m

fromMusicBase :: (Performance.T -> MidiFile.T) ->
   Context.T -> Music.T -> MidiFile.T
fromMusicBase p c m = p (Performance.fromMusic Player.fancyMap c m)
```

A table of General Midi assignments called `GeneralMidi.map` is imported from `GeneralMidi` in Section 6.1.7. The Midi file datatype itself is imported from the module `MidiFile`, functions for writing it to files are found in the module `SaveMidi`, and functions for reading MIDI files come from the modules `LoadMidi` and `ReadMidi`. All these modules are described later in this section.

6.1.1 The Gory Details

Some preliminaries, otherwise known as constants:

```
division :: Int
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 `performToMEvs`, which converts a `Performance.T` into a stream of `MEvents`.

As said before, we can't use absolute times, but when converting the relative performance event times (`Float`) to the relative midi event times (`Int`) we have to prevent accumulated rounding errors. We avoid this problem with a stateful conversion which remembers each rounding error we make. This rounding error is used to correct the next rounding. Given the relative time and duration of a note the function `roundTime` creates `State` which computes both the rounded time and the rounded duration. Both of them are corrected by previous rounding errors. The term `roundDiff t` computes the rounding state for the time rounding only. The function `mapState` is used to apply the rounding error to the duration conversion, while preserving the state of the rounding error for subsequent conversions.

We manage a `FiniteMap` 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.

```
type PatchMap = Music.IName -> (MidiFile.Channel, MidiFile.Program)

performToMEvs :: PatchMap -> Performance.T -> [MidiFile.Event]
performToMEvs pMap pf =
  let setTempo       = (0, MidiFile.MetaEvent (MidiFile.SetTempo MidiFile.defltST))

      times          = map (toDelta . eTime) pf
      durs           = map (toDelta . eDur)  pf
      roundTime d t  = mapState (\(t',frac) -> ((t', round (frac+d)),frac))
```

```
                                (roundDiff t)
      midiTimeDurs   = evalState (zipWithM roundTime durs times) 0

      updateChannelMap (midiChan, progNum) cm =
         if Just progNum == lookupFM cm midiChan
         then ((midiChan, Nothing), cm)
         else ((midiChan, Just (MidiFile.MidiEvent midiChan
                                  (MidiFile.ProgChange progNum))),
               addToFM cm midiChan progNum)
      midiChannels =
         evalState (mapM (State.State .
            updateChannelMap . pMap . eInst) pf) emptyFM

      midiEventPairs = map mkMEvents pf

      loop [] [] [] = []
      loop ((t,d):tds) ((me0,me1):meps) ((chan,progChange):chans) =
         let mec0 = MidiFile.MidiEvent chan me0
             mec1 = MidiFile.MidiEvent chan me1
         in  maybe ((t, mec0) :)
                (\pcME -> ((t, pcME) :) . ((0, mec0) :))
                progChange
                (insertMEvent (d, mec1) (loop tds meps chans))
      loop _ _ _ = error "Number of durations and number of events will always mato
  in  setTempo : loop midiTimeDurs midiEventPairs midiChannels
```

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 `MkMEvents` turns a Haskore `Event` into two `MEvents`, a `NoteOn` and a `NoteOff`.

```
mkMEvents :: Performance.Event -> (MidiFile.MidiEvent, MidiFile.MidiEvent)
mkMEvents (Performance.Event {ePitch = p, eVel = v}) =
   let v' = round (max 0 $ min 127 $ v*64)
       -- adjust frequency with respect to default piano
       midiP = p + MidiFile.zeroKey
       checkedP =
          if midiP<0
          then error ("pitch " ++ show midiP ++ " must not be negative")
          else if midiP>127
               then error ("pitch " ++ show midiP ++ " must be below 128")
               else midiP
   in  (MidiFile.NoteOn checkedP v', MidiFile.NoteOff checkedP v')

toDelta :: (Num a) => a -> a
toDelta t = t * 2 * fromIntegral division
--              ^ compensate defltDurT
```

The final critical function is `insertMEvent`, which inserts an `MidiFile.Event` into an already time-ordered sequence of `MEvents`. It is used to insert a `NoteOff` event into a list of `NoteOn` and `NoteOff` events.

It looks a bit cumbersome to insert every single `NoteOff` into the remaining list of events. 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 ...

```
insertMEvent :: MidiFile.Event -> [MidiFile.Event] -> [MidiFile.Event]
insertMEvent mev0 [] = [mev0]
insertMEvent mev0@(t0, me0) (mev1@(t1, me1) : mevs) =
   if mev0 <= mev1
   then mev0 : (t1-t0, me1) : mevs
   else mev1 : insertMEvent (t0-t1, me0) 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 (max 0 $ min 127 $ 64 + 16 * logBase 2 v)

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

6.1.2 User patch map

```
module Haskore.Interface.MIDI.UserPatchMap where

import Data.Char(toLower)
import Data.List(find)

import qualified Haskore.Music as Music
import qualified Haskore.Interface.MIDI.File    as MidiFile
import qualified Haskore.Interface.MIDI.General as GeneralMidi
```

A `UserPatchMap.T` is a user-supplied table for mapping instrument names (`INames`) 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 T = [(Music.IName, (MidiFile.Channel, MidiFile.Program))]
```

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

```
repair :: [Music.IName] -> T -> T
repair insts pMap =
```

```
   if allValid pMap insts
       then pMap
       else fromInstruments insts

allValid :: T -> [Music.IName] -> Bool
allValid upm = all (\x -> any (partialMatch x . fst) upm)
```

If a Haskore user only uses General Midi instrument names as `INames`, we can define a function that automatically creates a `UserPatchMap.T` 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).

```
fromInstruments :: [Music.IName] -> T
fromInstruments = fromInstruments' 0
  where fromInstruments' _ []     = []
        fromInstruments' n (i:is) =
           if n>=15 then
              error "Too many instruments; not enough MIDI channels."
           else
              if map toLower i `elem` percList
              then (i, (9, 0))
                       : fromInstruments' n is
              else (i, (chanList !! n,
                           Haskore.Interface.MIDI.UserPatchMap.lookup
                              GeneralMidi.map i))
                       : fromInstruments' (n+1) is
        percList = ["percussion", "perc", "drums"]
        chanList = [0..8] ++ [10..15]  -- 10th channel (#9) is for percussion
```

The following functions lookup `INames` in `UserPatchMap.Ts` to recover channel and program change numbers. Note that the function that does string matching ignores case, and allows substring matches. 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 :: String -> String -> Bool
partialMatch "piano" "Acoustic Grand Piano" = True
partialMatch s1 s2 =
  let s1' = map toLower s1
      s2' = map toLower s2
      len = min (length s1) (length s2)
  in take len s1' == take len s2'

lookup :: [(String, a)] -> String -> a
lookup  ys x =
   maybe (error ("UserPatchMap.lookup: Instrument " ++ x ++ " unknown"))
         snd (find (partialMatch x . fst) ys)
```

A default `UserPatchMap.T`. Note: the PC sound card I'm using is limited to 9 instruments.

```
deflt :: T
deflt =
   map (\(iName, gmName, chan) ->
         (iName, (chan, Haskore.Interface.MIDI.UserPatchMap.lookup
                     GeneralMidi.map gmName)))
   [("piano","Acoustic Grand Piano",1),
    ("vibes","Vibraphone",2),
    ("bass","Acoustic Bass",3),
    ("flute","Flute",4),
    ("sax","Tenor Sax",5),
    ("guitar","Acoustic Guitar (steel)",6),
    ("violin","Viola",7),
    ("violins","String Ensemble 1",8),
    ("drums","Acoustic Grand Piano",9)]
      -- the GM name for drums is unimportant, only channel 9
```

6.1.3 Midi-File Datatypes

```
module Haskore.Interface.MIDI.File(
        T(..), Division(..), Track, Type(..), Event, Event'(..), ElapsedTime,
        Pitch, ControlNum, PBRange, Program, Pressure,
        Channel, ControlVal, Velocity,
        MidiEvent(..),
        Tempo, SMPTEHours, SMPTEMins, SMPTESecs, SMPTEFrames, SMPTEBits,
        MetaEvent(..),
        Key(..), Mode(..),
       defltST, defltDurT, zeroKey, empty,
       showLines, changeVelocity, getTracks, resampleTime,
       sortEvents, progChangeBeforeSetTempo
        ) where

import Data.Ix(Ix)
import Data.List(sort,groupBy)
```

```
data T = C Type Division [Track] deriving (Show, Eq)

data Type     = Mixed | Parallel | Serial
     deriving (Show,Eq,Enum)
data Division = Ticks Int | SMPTE Int Int
     deriving (Show,Eq)

type Track       = [Event]
type Event       = (ElapsedTime, Event')
type ElapsedTime = Int
```

```
data Event' = MidiEvent Channel MidiEvent
            | MetaEvent MetaEvent
            | SysExStart String         -- F0
            | SysExCont  String         -- F7
     deriving (Show,Eq,Ord)
```

```
type Pitch       = Int
type Velocity    = Int
type ControlNum  = Int
type PBRange     = Int
type Program     = Int
type Pressure    = Int
type Channel     = Int
type ControlVal  = Int

data MidiEvent = NoteOff    Pitch Velocity
               | NoteOn     Pitch Velocity
               | PolyAfter  Pitch Pressure
               | ProgChange Program
               | Control    ControlNum ControlVal
               | PitchBend  PBRange
               | MonoAfter  Pressure
     deriving (Show, Eq, Ord)
```

```
type Tempo       = Int
type SMPTEHours  = Int
type SMPTEMins   = Int
type SMPTESecs   = Int
type SMPTEFrames = Int
type SMPTEBits   = Int

data MetaEvent = SequenceNum Int
               | TextEvent String
               | Copyright String
               | TrackName String
               | InstrName String
               | Lyric String
               | Marker String
               | CuePoint String
```

```
               | MIDIPrefix Channel
               | EndOfTrack
               | SetTempo Tempo
               | SMPTEOffset SMPTEHours SMPTEMins SMPTESecs SMPTEFrames SMPTEBits
               | TimeSig Int Int Int Int
               | KeySig Key Mode
               | SequencerSpecific [Int]
               | Unknown String
    deriving (Show, Eq, Ord)
```

The following enumerated type lists all the keys in order of their key signatures from flats to sharps. (Cf = 7 flats, Gf = 6 flats ... F = 1 flat, C = 0 flats/sharps, G = 1 sharp, ... Cs = 7 sharps.) Useful for transposition.

```
data Key = KeyCf | KeyGf | KeyDf | KeyAf | KeyEf | KeyBf | KeyF
         | KeyC  | KeyG  | KeyD  | KeyA  | KeyE  | KeyB  | KeyFs | KeyCs
             deriving (Eq, Ord, Ix, Enum, Show)
```

The Key Signature specifies a mode, either major or minor.

```
data Mode = Major | Minor
            deriving (Show, Eq, Ord, Enum)
```

Default duration of a whole note, in seconds; and the default SetTempo value, in microseconds per quarter note. Both express the default of 120 beats per minute.

```
defltDurT :: Int
defltDurT = 2
defltST :: Int
defltST = div 1000000 defltDurT
```

A MIDI problem is that one cannot uniquely map a MIDI key to a frequency. The frequency depends on the instrument. I don't know if the deviations are defined for General MIDI. If this applies one could add transposition information to the use patch map. For now I have chosen a value that leads to the right frequency for some piano sound in my setup.

```
zeroKey :: Pitch
zeroKey = 48
```

An empty MIDI file.

```
empty :: T
empty = C Mixed (Ticks 0) [[]]
```

Some routines for debugging of Midi data Show the `File.T` with one event per line, suited for comparing MidiFiles with 'diff'. Can this be replaced by `LoadMidi.showMidiFile`?

```
showLines :: T -> String
showLines (C mfType division tracks) =
   let showTrack track =
          "    (\n" ++
          unlines (map (\event -> "      " ++ show event ++ " :") track) ++
          "    []) :\n"
   in  "MidiFile.C " ++ show mfType ++ " (" ++ show division ++ ") (\n" ++
       concatMap showTrack tracks ++
       "  [])"
```

A hack that changes the velocities by a rational factor.

```
changeVelocity :: Double -> T -> T
changeVelocity r (C mfType division tracks) =
   let multVel vel = round (r * fromIntegral vel)
       procMidiEvent (NoteOn  pitch vel) = NoteOn  pitch (multVel vel)
       procMidiEvent (NoteOff pitch vel) = NoteOff pitch (multVel vel)
       procMidiEvent me = me
       procEvent (time, MidiEvent chan ev) =
          (time, MidiEvent chan (procMidiEvent ev))
       procEvent ev = ev
   in  C mfType division (map (map procEvent) tracks)
```

Changing the time base.

```
resampleTime :: Double -> T -> T
resampleTime r (C mfType division tracks) =
   let divTime time = round (fromIntegral time / r)
       procEvent (0, MetaEvent (SetTempo t)) =
          (0, MetaEvent (SetTempo (round (fromIntegral t * r))))
       procEvent (_, MetaEvent (SetTempo _)) =
          error "SetTempo can be handled only at time 0"
       procEvent (time, ev) = (divTime time, ev)
   in  C mfType division (map (map procEvent) tracks)

getTracks :: T -> [Track]
getTracks (C _ _ trks) = trks
```

Sort MIDI note events lexicographically. This is to make MIDI files unique and robust against changes in the computation. In principle Performance.merge should handle this but due to rounding errors in Float the order of note events still depends on some internal issues. The sample rate of MIDI events should be coarse enough to assert unique results.

```
sortEvents :: T -> T
sortEvents (C mfType division tracks) =
   let isNote (NoteOn  _ _) = True
       isNote (NoteOff _ _) = True
```

```
      isNote _             = False
      coincideNote (_, MidiEvent _ x0) (t1, MidiEvent _ x1) =
         t1 == 0 && isNote x0 && isNote x1
      coincideNote _ _ = False
      sortTime mes =
         let (me':mes') = sort (map snd mes)
         in  (fst (head mes), me') : map (\me->(0,me)) mes'
      sortTrack = concatMap sortTime . groupBy coincideNote
   in  C mfType division (map sortTrack tracks)
```

Old versions of module `WriteMidi` wrote `ProgramChange` and `SetTempo` once at the beginning of a file in that order. The current version supports multiple `ProgramChange`s in a track and thus a `ProgramChange` is set immediately before a note. Because of this a `ProgramChange` is now always after a `SetTempo`. For checking equivalence with old MIDI files we can switch this back.

```
progChangeBeforeSetTempo :: T -> T
progChangeBeforeSetTempo (C mfType division tracks) =
   let sortTrack ((t0, st@(MetaEvent (SetTempo _))) :
                  (t1, pc@(MidiEvent _ (ProgChange _))) :
                  (t2, me2) : mes) =
         (t0, pc) : (0, st) : (t1+t2, me2) : mes
      sortTrack ((t0, st@(MetaEvent (SetTempo _))) :
                  (_,  pc@(MidiEvent _ (ProgChange _))) : mes) =
         (t0, pc) : (0, st) : mes
      sortTrack mes = mes
   in  C mfType division (map sortTrack tracks)
```

6.1.4 Saving MIDI Files

The functions in this module allow `MidiFile.T`s to be made into Standard MIDI files (*.mid) that can be read and played by music programs such as Cakewalk.

```
module Haskore.Interface.MIDI.Save (toFile, toStream, toOpenStream) where

import Data.Char(chr)
import Data.Ix
import Control.Monad.Writer (Writer, tell, execWriter)
import IOExtensions (writeBinaryFile)

import           Haskore.Interface.MIDI.File
import qualified Haskore.Interface.MIDI.File        as MidiFile
import qualified Haskore.General.Bit as Bit
```

The function `SaveMidi.toFile` is the main function for writing `MidiFile` values to an actual file; its first argument is the filename:

```
toFile :: FilePath -> MidiFile.T -> IO ()
toFile fn mf = writeBinaryFile fn (toStream mf)
```

19 Exercise. *Take as many examples as you like from the previous sections, create one or more* `UserPatchMap`*s, write the examples to a file, and play them using a conventional Midi player.*

Section A defines some functions which should make the above exercise easier. Sections B.1, B.2, and B.3 contain more extensive examples.

Midi files are first converted to a monadic string computation using the function `outMF`, and then "executed" using `runM :: MidiWriter a -> String`.

```
toStream, toOpenStream :: MidiFile.T -> String
toStream     = execWriter . outMF outChunk
toOpenStream = execWriter . outMF outOpenChunk

outMF :: OutChunk -> MidiFile.T -> MidiWriter ()
outMF outChk (MidiFile.C mft divisn trks) =
  do
    outChunk "MThd" (do
                       out 2 (fromEnum mft)    -- format (type 0, 1 or 2)
                       out 2 (length trks)     -- number of tracks to come
                       outputDivision divisn)  -- time unit
    mapM_ (outputTrack outChk) trks

outputDivision :: Division -> MidiWriter ()
outputDivision (Ticks nticks)      = out 2 nticks
outputDivision (SMPTE mode nticks) = do
                                       out 1 (256-mode)
                                       out 1 nticks

outputTrack :: OutChunk -> Track -> MidiWriter ()
outputTrack outChk trk =
   outChk "MTrk" (mapM_ outputEvent (trk ++ [(0, MetaEvent EndOfTrack)]))
```

The following functions encode various `MidiFile.T` elements into the raw data of a standard MIDI file.

```
outputEvent :: MidiFile.Event -> MidiWriter ()
outputEvent (dt, e) =
   do outVar dt
      case e of
         (MidiEvent ch mevent) -> outputMidiEvent ch mevent
         (MetaEvent mevent)    -> outputMetaEvent mevent
         _                     -> error ("don't know, how to write a "++show e++".'

outputMidiEvent :: MidiFile.Channel -> MidiEvent -> MidiWriter ()
```

```
outputMidiEvent c e =
   let outC = outChan c
   in  case e of
         (NoteOff    p v)   -> outC  8 [p,v]
         (NoteOn     p v)   -> outC  9 [p,v]
         (PolyAfter  p pr)  -> outC 10 [p,pr]
         (Control    cn cv) -> outC 11 [cn,cv]
         (ProgChange pn)    -> outC 12 [pn]
         (MonoAfter  pr)    -> outC 13 [pr]
         (PitchBend  pb)    -> outC 14 [lo,hi] -- little-endian!!
                                 where (hi,lo) = Bit.splitAt 8 pb

-- output a channel event
outChan :: MidiFile.Channel -> Int -> [Int] -> MidiWriter ()
outChan chan code bytes = do
                           out 1 (16*code+chan)
                           mapM_ (out 1) bytes


outMeta    :: Int -> [Int] -> MidiWriter ()
outMeta code bytes = do
                      out 1 255
                      out 1 code
                      outVar (length bytes)
                      outList bytes

outMetaStr :: Int -> String -> MidiWriter ()
outMetaStr code bytes = do
                         out 1 255
                         out 1 code
                         outVar (length bytes)
                         outStr bytes

-- As with outChunk, there are other ways to do this - but
-- it's not obvious which is best or if performance is a big issue.
outMetaMW :: Int -> MidiWriter a -> MidiWriter a
outMetaMW code m = do
                    out 1 255
                    out 1 code
                    outVar (mLength m)
                    m

outputMetaEvent :: MetaEvent -> MidiWriter ()
outputMetaEvent (SequenceNum num) = outMetaMW   0 (out 2 num)
outputMetaEvent (TextEvent s)     = outMetaStr  1 s
```

```
outputMetaEvent (Copyright s)     = outMetaStr  2 s
outputMetaEvent (TrackName s)     = outMetaStr  3 s
outputMetaEvent (InstrName s)     = outMetaStr  4 s
outputMetaEvent (Lyric s)         = outMetaStr  5 s
outputMetaEvent (Marker s)        = outMetaStr  6 s
outputMetaEvent (CuePoint s)      = outMetaStr  7 s
outputMetaEvent (MIDIPrefix c)    = outMeta    32 [c]
outputMetaEvent EndOfTrack        = outMeta    47 []

outputMetaEvent (SetTempo tp)     = outMetaMW  81 (out 3 tp)
outputMetaEvent (SMPTEOffset hr mn se fr ff)
                                  = outMeta    84 [hr,mn,se,fr,ff]
outputMetaEvent (TimeSig n d c b) = outMeta    88 [n,d,c,b]
outputMetaEvent (KeySig sf mi)    = outMeta    89 [sf', fromEnum mi]
                                      where k = index (KeyCf,KeyCs) sf - 7
                                            sf' = if (k >= 0)
                                                  then k
                                                  else 255+k
outputMetaEvent (SequencerSpecific codes)
                                  = outMeta    127 codes
outputMetaEvent (Unknown s)       = outMetaStr 21 s
```

The midiwriter accumulates a String. For all the usual reasons, the String is represented by ShowS.

```
type MidiWriter a = Writer [Char] a

mLength  :: MidiWriter a -> Int
mLength m = length (execWriter m)

out :: Int -> Int -> MidiWriter ()
out a x = tell (map (chr . fromIntegral) (Bit.someBytes a x))

outStr :: String -> MidiWriter ()
outStr cs = tell cs

outList :: [Int] -> MidiWriter ()
outList xs = tell (map chr xs)
```

Numbers of variable size are represented by sequences of 7-bit blocks tagged (in the top bit) with a bit indicating: (1) that more data follows; or (0) that this is the last block.

```
outVar :: Int -> MidiWriter ()
outVar n = do
             outVarAux leftover
             out 1 data7
           where (leftover, data7) = Bit.splitAt 7 n
                 outVarAux 0 = return ()
```

```
                outVarAux x = do
                                outVarAux leftover'
                                out 1 (128+data7')  --make signal bit 1
                              where (leftover',data7') = Bit.splitAt 7 x

outTag :: String -> MidiWriter ()
outTag tag@(_:_:_:_:[]) = outStr tag
outTag tag =
   error ("SaveMidi.outChunk: Chunk name " ++ tag ++
          " does not consist of 4 characters.")

-- Note: here I've chosen to compute the track twice
-- rather than store it.  Other options are worth exploring.

type OutChunk = String -> MidiWriter () -> MidiWriter ()

outChunk, outOpenChunk :: OutChunk

outChunk tag m =
   do
     outTag tag
     out 4 (mLength m)
     m

{- Does the MIDI standard allow chunks without a length specification?
   This is essential for infinite music and music that is created on the fly. -}
outOpenChunk tag m =
   do
     outTag tag
     out 4 (-1)
     m
```

6.1.5 Loading MIDI Files

The Haskore.Interface.MIDI.Load module loads and parses a MIDI File; it can convert it into a MidiFile data type object or simply print out the contents of the file.

```
module Haskore.Interface.MIDI.Load (fromStream, fromFile, showFile)
 where

import           Haskore.Interface.MIDI.File
import qualified Haskore.Interface.MIDI.File as MidiFile

import IOExtensions (readBinaryFile)
import qualified Haskore.General.Bit as Bit
```

```
import Data.Bits (testBit, (.|.))
import Data.Word (Word8, Word32)
import Data.Char (ord)
import Data.Maybe (fromJust, mapMaybe)
import Haskore.General.Utility (unlinesS, rightS, concatS)
import Control.Monad (MonadPlus, mzero, mplus)
```

The main load function.

```
fromFile :: FilePath -> IO MidiFile.T
fromFile filename =
   fmap fromStream (readBinaryFile filename)

fromStream :: String -> MidiFile.T
fromStream contents =
    case runP parse (contents, (AtBeginning,0),-1) of
      Just (mf,"",_,_) -> mf
      Just (_ ,_ ,_,_) -> error "Garbage left over." -- return mf
      Nothing          -> error "Error reading midi file: unfamiliar format or file
```

A MIDI file is made of “chunks”, each of which is either a “header chunk” or a “track chunk”. To be correct, it must consist of one header chunk followed by any number of track chunks, but for robustness’s sake we ignore any non-header chunks that come before a header chunk. The header tells us the number of tracks to come, which is passed to `getTracks`.

```
parse :: MidiReader MidiFile.T
parse =
   do
      chunk <- getChunk
      case chunk of
        Header (format, nTracks, division) ->
          do
            chunks <- sequence (replicate nTracks getChunk)
            return (MidiFile.C format division (map removeEndOfTrack
                       (mapMaybe trackFromChunk chunks)))
        _                                -> parse
```

Check if a chunk contains a track. Like `parse`, if a chunk is not a track chunk, it is just ignored.

```
trackFromChunk :: Chunk -> Maybe Track
trackFromChunk (Track t) = Just t
trackFromChunk  _        = Nothing
```

There are two ways to mark the end of the track: The end of the event list and the meta event `EndOfTrack`. Thus the end marker is redundant and we remove a `EndOfTrack` at the end of the track and complain about all `EndOfTrack`s within the event list.

```
removeEndOfTrack :: Track -> Track
```

```
removeEndOfTrack [] = error "Track does not end with EndOfTrack"
removeEndOfTrack ((_, MetaEvent EndOfTrack):[]) = []
removeEndOfTrack ((_, MetaEvent EndOfTrack):_) =
   error "EndOfTrack inside a track"
removeEndOfTrack (e:es) = e : removeEndOfTrack es
```

Parse a chunk, whether a header chunk, a track chunk, or otherwise. A chunk consists of a four-byte type code (a header is “MThd”; a track is “MTrk”), four bytes for the size of the coming data, and the data itself.

```
getChunk :: MidiReader Chunk
getChunk = do
               ty                    <- getN 4
               size                  <- get4
               setSize size
               case ty of
                 "MThd"    -> fmap Header getHeader
                 "MTrk"    -> fmap Track  getTrack
                 _         -> do
                                getN size   --  g <- getN size
                                return AlienChunk

data Chunk = Header (MidiFile.Type, Int, Division)
           | Track Track
           | AlienChunk
  deriving Eq
```

Parse a Header Chunk. A header consists of a format (0, 1, or 2), the number of track chunks to come, and the smallest time division to be used in reading the rest of the file.

```
getHeader :: MidiReader (MidiFile.Type, Int, Division)
getHeader =
        do
           format   <- get2
           nTracks  <- get2
           division <- getDivision
           return (toEnum format, nTracks, division)
```

The division is implemented thus: the most significant bit is 0 if it’s in ticks per quarter note; 1 if it’s an SMPTE value.

```
getDivision :: MidiReader Division
getDivision = do
               x <- get1
               y <- get1
               if x < 128
                 then return (Ticks (x*256+y))
                 else return (SMPTE (256-x) y)
```

A track is a series of events. Parse a track, stopping when the size is zero.

```
getTrack :: MidiReader [MidiFile.Event]
getTrack =
       do
          size <- readSize
          case size of
            0 -> return []
            _ -> do
                   e  <- getFancyEvent
                   es <- getTrack
                   return (e:es)
```

Each event is preceded by the delta time: the time in ticks between the last event and the current event. Parse a time and an event, ignoring System Exclusive messages.

```
getFancyEvent :: MidiReader MidiFile.Event
getFancyEvent =
   do
      time <- getVar
      e    <- getEvent
      return (time, e)
```

Parse an event. Note that in the case of a regular Midi Event, the tag is the status, and we read the first byte of data before we call `midiEvent`. In the case of a MidiEvent with running status, we find out the status from the parser (it's been nice enough to keep track of it for us), and the tag that we've already gotten is the first byte of data.

```
getEvent :: MidiReader MidiFile.Event'
getEvent =
       do
          tag <- get1
          case tag of
            240       -> do
                         size     <- getVar
                           contents <- getN size
                           return (SysExStart contents)
            247       -> do
                         size     <- getVar
                           contents <- getN size
                           return (SysExCont contents)
              255       -> do
                           code <- get1
                           size <- getVar
                         e    <- getMetaEvent code size
                           return (MetaEvent e)
              x | x>127 -> do
```

```
                        firstData <- get1
                        getMidiEvent (decodeStatus tag) firstData
            _         -> do                         -- running status
                        s <- readME
                        getMidiEvent s tag
```

Simpler version of `getFancyTrack`, used in the Show functions.

```
getPlainTrack :: MidiReader [MidiFile.Event]
getPlainTrack = oneOrMore getFancyEvent

data WhichMidiEvent = AtBeginning
                    | ItsaNoteOff
                    | ItsaNoteOn
                    | ItsaPolyAfter
                    | ItsaControl
                    | ItsaProgChange
                    | ItsaMonoAfter
                    | ItsaPitchBend
       deriving Show
type Status = (WhichMidiEvent, Int)
```

Find out the status (MidiEvent type and channel) given a byte of data.

```
decodeStatus :: Int -> Status
decodeStatus tag = (w, channel)
  where w = case code of
               08 -> ItsaNoteOff
               09 -> ItsaNoteOn
               10 -> ItsaPolyAfter
               11 -> ItsaControl
               12 -> ItsaProgChange
               13 -> ItsaMonoAfter
               14 -> ItsaPitchBend
               _  -> error "invalid MidiEvent code"
        (code, channel) = Bit.splitAt 4 tag
```

Parse a MIDI Event. Note that since getting the first byte is a little complex (there are issues with running status), it has already been handled for us by `event`.

```
getMidiEvent :: Status -> Int -> MidiReader MidiFile.Event'
getMidiEvent s@(wME, channel) firstData =
   let getME =
         case wME of
           ItsaNoteOff    -> fmap (NoteOff   firstData) get1
           ItsaNoteOn     -> fmap (NoteOn    firstData) get1
{-
```

```
            ItsaNoteOn     -> do v <- get1
                                 case v of
                                   0 -> return (NoteOff firstData 0)
                                   _ -> return (NoteOn  firstData v)
-}
            ItsaPolyAfter  -> fmap (PolyAfter firstData) get1
            ItsaControl    -> fmap (Control   firstData) get1
            ItsaPitchBend  -> fmap (\msb -> PitchBend (firstData+256*msb)) get1
            ItsaProgChange -> return (ProgChange firstData)
            ItsaMonoAfter  -> return (MonoAfter  firstData)
            AtBeginning    -> error "AtBeginning"
   in  do setME s
          fmap (MidiEvent channel) getME
```

Parse a MetaEvent.

```
getMetaEvent :: Int -> Int -> MidiReader MetaEvent
getMetaEvent code size =
   case code of
      000 -> fmap SequenceNum (get2)
      001 -> fmap TextEvent   (getN size)
      002 -> fmap Copyright   (getN size)
      003 -> fmap TrackName   (getN size)
      004 -> fmap InstrName   (getN size)
      005 -> fmap Lyric       (getN size)
      006 -> fmap Marker      (getN size)
      007 -> fmap CuePoint    (getN size)

      032 -> fmap MIDIPrefix   get1
      047 -> return EndOfTrack
      081 -> fmap SetTempo     get3

      084 -> do {hrs    <- get1 ; mins <- get1 ; secs <- get1;
                 frames <- get1 ; bits <- get1 ;
                 return (SMPTEOffset hrs mins secs frames bits)}

      088 -> do
                n <- get1
                d <- get1
                c <- get1
                b <- get1
                return (TimeSig n d c b)

      089 -> do
                sf <- get1
                mi <- get1
```

```
                return (KeySig (toKeyName sf) (toEnum mi))

      127 -> fmap (SequencerSpecific . map fromEnum) (getN size)

      _   -> fmap Unknown (getN size)

toKeyName :: Int -> Key
toKeyName sf = toEnum ((sf+7) `mod` 15)
```

`getCh` gets a single character (a byte) from the input.

```
getCh :: MidiReader Char
getCh = do {sub1Size; tokenP myHead}
          where  myHead ([]    ,_ ,_ ) = Nothing
                 myHead ((c:cs),st,sz) = Just (c,cs,st,sz)
```

`getN n` returns n characters (bytes) from the input.

```
getN :: Int -> MidiReader String
getN 0 = return []
getN n = do
           a <- getCh
           b <- getN (n-1)
           return (a:b)
```

`get1`, `get2`, `get3`, and `get4` take 1-, 2-, 3-, or 4-byte numbers from the input (respectively), convert the base-256 data into a single number, and return.

```
getByte :: MidiReader Word8
getByte = fmap fromIntegral get1

get1 :: MidiReader Int
get1 = fmap ord getCh

get2 :: MidiReader Int
get2 = do
         x1 <- get1
         x2 <- get1
         return (Bit.fromBytes [x1,x2])

get3 :: MidiReader Int
get3 = do
         x1 <- get1
         x2 <- get1
         x3 <- get1
         return (Bit.fromBytes [x1,x2,x3])
```

```
get4 :: MidiReader Int
get4 = do
         x1 <- get1
         x2 <- get1
         x3 <- get1
         x4 <- get1
         return (Bit.fromBytes [x1,x2,x3,x4])
```

Variable-length quantities are used often in MIDI notation. They are represented in the following way. Each byte (containing 8 bits) uses the 7 least significant bits to store information. The most significant bit is used to signal whether or not more information is coming. If it's 1, another byte is coming. If it's 0, that byte is the last one. `getVar` gets a variable-length quantity from the input.

```
getVar :: MidiReader Int
getVar =
  let getVarAux n =
        do
           digit <- getByte
           let digitExt = fromIntegral digit :: Word32
            in if flip testBit 7 digit            -- if it's the last byte
               then getVarAux (Bit.shiftL 7 n .|. Bit.trunc 7 digitExt)
               else return (fromIntegral (Bit.shiftL 7 n .|. digitExt))
  in  getVarAux 0
```

Functions to show the decoded contents of a Midi file in an easy-to-read format.

```
showFile :: String -> IO ()
showFile file = readBinaryFile file >>= (putStr . showChunks)

showChunks :: String -> String
showChunks mf = showMR getChunks (unlinesS . map pp) (mf, (AtBeginning,0),-1) ""
 where
  pp :: (String, String, Status, Int) -> ShowS
  pp ("MThd",contents,st,sz) =
    showString "Header: " .
    showMR getHeader shows (contents,st,sz)
  pp ("MTrk",contents,st,sz) =
    showString "Track:\n" .
    showMR getPlainTrack (unlinesS . map showTrackEvent) (contents,st,sz)
  pp (ty,contents,_,_) =
    showString "Chunk: " .
    showString ty .
    showString " " .
    shows (map fromEnum contents) .
    showString "\n"

showTrackEvent :: MidiFile.Event -> ShowS
```

```
showTrackEvent (t,e) =
  rightS 10 (shows t) . showString " : " . showEvent e

showEvent :: MidiFile.Event' -> ShowS
showEvent (MidiEvent ch e) =
  showString "MidiEvent " . shows ch . showString " " . shows e
showEvent (MetaEvent e) =
  showString "MetaEvent " . shows e
showEvent (SysExStart s) =
  showString "SysExStart " . concatS (map (shows.fromEnum) s)
showEvent (SysExCont s) =
  showString "SysExCont "  . concatS (map (shows.fromEnum) s)

showMR :: MidiReader a -> (a->ShowS) -> (String, Status, Int) -> ShowS
showMR m pp (s,st,sz) =
  case runP m (s,st,sz) of
  Nothing           -> showString "Parse failed: " . shows (map fromEnum s)
  Just (a,[]  ,_,_) -> pp a
  Just (a,junk,_,_) -> pp a . showString "Junk: " . shows (map fromEnum junk)
```

These two functions, the `plainChunk` and `getChunks` parsers, do not combine directly into a single master parser. Rather, they should be used to chop parts of a midi file up into chunks of bytes which can be outputted separately.

Chop a Midi file into chunks returning:

- list of "chunk-type"-contents-running status triples; and
- leftover slop (should be empty in correctly formatted file)

```
getChunks :: MidiReader [(String, String, Status, Int)]
getChunks = zeroOrMore getPlainChunk

getPlainChunk :: MidiReader (String, String, Status, Int)
getPlainChunk =
            do
                ty       <- getN 4       -- chunk type: header or track
               size     <- get4         -- size of what's next
               contents <- getN size    -- what's next
               status   <- readME       -- running status
               return (ty, contents, status, -1) -- Don't worry about size
```

The following parser monad parses a Midi File. As it parses, it keeps track of these things:

- `(w,c)` a.k.a. `st` Running status. In MIDI, a shortcut is used for long strings of similar MIDI events: if a stream of consecutive events all have the same type and channel, the type and channel can be ommitted for all but the first event. To implement this "feature", the parser must keep track of the type and channel of the most recent Midi Event.

- `sz` The size, in bytes, of what's left to parse, so that it knows when it's done.

```
type MidiReader a = Parser String WhichMidiEvent Int Int a

data Parser s w c sz a = P ((s,(w,c),sz) -> Maybe (a,s,(w,c),sz))

unP :: Parser s w c sz a -> ((s,(w,c),sz) -> Maybe (a,s,(w,c),sz))
unP (P a) = a

-- Access to state
tokenP  :: ((s,(w,c),sz) -> Maybe (a,s,(w,c),sz)) -> Parser s w c sz a
runP    :: Parser s w c sz a -> (s,(w,c),sz) -> Maybe (a,s,(w,c),sz)

tokenP get   = P $ get
runP m (s,st,sz) = (unP m) (s,st,sz)

instance Monad (Parser s w c sz) where
  m >>= k  = P $ \ (s,st,sz) -> do
                                    (a,s',st',sz') <- unP m (s,st,sz)
                                    unP (k a) (s',st',sz')
  m >>  k  = P $ \ (s,st,sz) -> do
                                    (_,s',st',sz') <- unP m (s,st,sz)
                                    unP k     (s',st',sz')
  return a = P $ \ (s,st,sz) -> return (a,s,st,sz)

instance Functor (Parser s w c sz) where
  fmap f m = P $ \ (s,st,sz) -> do   (a,s',st',sz') <- unP m (s,st,sz)
                                    return (f a,s',st',sz')
--  fmap f m = do { a <- m; return (f a)}
--  fmap f m = m >>= (\ a -> return (f a))

setME  :: Status -> MidiReader ()
setME st' = P $ \ (s,_,sz) -> return ((),s,st',sz)

readME :: MidiReader Status
readME  = P $ \ (s,st,sz) -> return (st,s,st,sz)

setSize :: Int -> MidiReader ()
setSize sz' = P $ \ (s,st,_) -> return ((),s,st,sz')

sub1Size :: MidiReader ()
sub1Size = P $ \ (s,st,sz) -> return ((),s,st,(sz-1))

readSize :: MidiReader Int
readSize = P $ \ (s,st,sz) -> return (sz,s,st,sz)
```

```
-- instance MonadZero (Parser s w c sz) where

instance MonadPlus (Parser s w c sz) where
  mzero = P $ \ _ -> mzero
  p `mplus` q = P $ \ (s,st,sz) -> unP p (s,st,sz) `mplus` unP q (s,st,sz)

-- Wadler's force function
force             :: Parser s w c sz a -> Parser s w c sz a
force (P p)        = P $ \ (s,st,sz) -> let x = p (s,st,sz)
                                        in Just (fromJust x)

zeroOrMore        :: Parser s w c sz a -> Parser s w c sz [a]
zeroOrMore p       = force (oneOrMore p `mplus` return [])

oneOrMore         :: Parser s w c sz a -> Parser s w c sz [a]
oneOrMore p        = do {x <- p; xs <- zeroOrMore p; return (x:xs)}
```

6.1.6 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 `Music.T` object.

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

import Data.Ratio ((%))
import Data.FiniteMap (FiniteMap, listToFM, fmToList, addToFM,
                       lookupWithDefaultFM)
import Data.List (groupBy)
import Data.Maybe (mapMaybe)

import Haskore.Music
             (note, rest, line, chord, (+:+), (=:=),
              changeTempo, setInstrument,
              Dur, DurRatio, NoteAttribute)

import           Haskore.Interface.MIDI.File

import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Music as Music
import qualified Haskore.Music.Player as Player
import qualified Haskore.Music.PerformanceContext as Context
import qualified Haskore.Process.Optimization as Optimization
```

```
import qualified Haskore.Interface.MIDI.File         as MidiFile
import qualified Haskore.Interface.MIDI.General      as GeneralMidi
import qualified Haskore.Interface.MIDI.UserPatchMap as UserPatchMap
```

The main function. Note that we output a `UserPatchMap.T` and a `Context.T` as well as a `Music.T` object.

```
toMusic :: MidiFile.T -> (UserPatchMap.T, Context.T, Music.T)
toMusic mf@(MidiFile.C _ d trks) =
  let upm     = makeUPM trks
      upm'    = map (\(ch, progNum) ->
                       (GeneralMidi.numberToIName ch progNum, (ch, progNum)))
                    (fmToList upm)
      m       = format (readFullTrack d upm) mf
      context = Context.setPlayer Player.fancy $
                Context.setDur 1 $
                Context.deflt
  in (upm', context, m)

retrieveTracks :: MidiFile.T -> [[Music.T]]
retrieveTracks (MidiFile.C _ d trks) =
  let upm         = makeUPM trks
      m = map (map (readTrack (tDiv d) upm . fst) . map (getRest defltST)
             . splitBy isTempoChg . mergeNotes defltST . moveTempoToHead) trks
  in m

type UserPatchMap = FiniteMap MidiFile.Channel MidiFile.Program

readFullTrack :: Division -> UserPatchMap -> Track -> Music.T
readFullTrack dv upm trk =
  let trksrs = map (getRest defltST)
                   (splitBy isTempoChg (mergeNotes defltST
                       (moveTempoToHead trk)))
      readTempoTrack (t,r) =
             changeTempo r (readTrack (tDiv dv) upm t)
  in  Optimization.all $ line $ map readTempoTrack trksrs
```

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) -> MidiFile.T -> Music.T
format tm (MidiFile.C MidiFile.Mixed    _ [trk]) = tm trk
format _  (MidiFile.C MidiFile.Mixed    _ _)
   = error ("toMusic: Only one track allowed for MIDI file type 0.")
format tm (MidiFile.C MidiFile.Parallel _ trks)  = chord (map tm trks)
format tm (MidiFile.C MidiFile.Serial   _ trks)  = line  (map tm trks)
```

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

```
makeUPM :: [Track] -> UserPatchMap
makeUPM trks =
   let getPC (MidiEvent ch (ProgChange num)) = Just (ch, num)
       getPC _ = Nothing
   in  listToFM $ concatMap (mapMaybe (getPC . snd)) trks
```

Translate `Divisions` into the number of ticks per quarter note.

```
tDiv :: Division -> Int
tDiv (Ticks x) = x
tDiv (SMPTE _ _) = error "Sorry, SMPTE not yet implemented."
```

`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 `splitBy isTempoChg`.

```
moveTempoToHead :: Track -> Track
moveTempoToHead es@((0, MetaEvent (SetTempo _)):_) = es
moveTempoToHead es@((0, _):_) = skipStartEvent es
moveTempoToHead es = (0, MetaEvent (SetTempo defltST)) : es

skipStartEvent :: Track -> Track
skipStartEvent (e:es) =
   let (tempo:es') = moveTempoToHead es in (tempo : e : es')
skipStartEvent [] =
   error "skipStartEvent: Lists returned by moveTempoToHead must contain at least o
```

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.

```
getRest :: Int -> [RichEvent] -> ([RichEvent], DurRatio)
getRest d ((_, Event (MetaEvent (SetTempo tempo))) : es) =
   (es, toInteger d % toInteger tempo)
getRest _ trk = (trk, 1)
```

`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. Used to split a track into sub-tracks by tempo. 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 = groupBy (\_ x -> not (p x))
```

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

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

The function `readTrack` could also directly map the stream of note events to a big parallel composition where each channel consists of one note. (The normal form as described in Hudak's Temporal Media paper.) But we try to avoid obviously unnecessary parallelism by watching for non-overlapping notes. Nevertheless the structure of data returned by `readTrack` is not very nice.

```
type RichEvent = (ElapsedTime, RichEvent')
data RichEvent' =
     Event MidiFile.Event'
   | Note  ElapsedTime (Velocity,Velocity) Channel Pitch

readTrack :: Int -> UserPatchMap -> [RichEvent] -> Music.T
readTrack _     _   [] = rest 0
readTrack ticks upm ((t0, re0) : es0) =
   let readTrack' = readTrack ticks upm
       body =
          case re0 of
             Note d (v,_) ch mp ->
                let p = Pitch.fromInt (mp - MidiFile.zeroKey)
                    plainNote = note p (fromTicks ticks d) (makeVel v)
                    n = if d>=0
                        then setInstrument (lookupUPM upm ch) plainNote
                        else error "readTrack: note of negative duration"
                in  case es0 of
                       ((t1, re1) : es1) ->
                          if t1 >= d
                          then n +:+ readTrack' ((t1-d, re1) : es1)
                          else n =:= readTrack' es0
                       [] -> n
             Event (MidiEvent ch (ProgChange num)) ->
                readTrack ticks (progChange ch num upm) es0
             _ ->
                readTrack' es0
   in  if t0 < 0
       then error "readTrack: NoteOn events out of order"
       else if t0 > 0
            then rest (fromTicks ticks t0) +:+ body
            else body
```

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

```
fromTicks :: Int -> Int -> Dur
fromTicks ticks d = toInteger d  %  toInteger (ticks * defltDurT)
```

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

```
lookupUPM :: UserPatchMap -> Int -> String
lookupUPM upm ch =
   GeneralMidi.numberToIName ch
      (lookupWithDefaultFM upm
          (error "Invalid channel in user patch map") ch)
```

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

```
progChange :: MidiFile.Channel -> MidiFile.Program ->
   UserPatchMap -> UserPatchMap
progChange ch num upm = addToFM upm ch num
```

Load the velocity. This shouldn't be mixed up with the volume. The volume which is controlled by the MIDI Volume controller simply scales the signal whereas the velocity is an instrument specific value that corresponds to the intensity with which the instrument is played.

```
makeVel :: Int -> [NoteAttribute]
makeVel x = [Music.Velocity (fromIntegral x / 64)]
```

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 :: Int -> Track -> [RichEvent]
mergeNotes dur = mergeNotes' 0 dur
  where
    mergeNotes' :: Int -> Int -> Track -> [RichEvent]
    mergeNotes' _   _   []        = []
    mergeNotes' _   _   ((newStt, e@(MetaEvent (SetTempo newStv))):es) =
       (newStt, Event e) : mergeNotes' newStt newStv es
    mergeNotes' _   _   ((_, MidiEvent _ (NoteOff _ _)) : _) =
       error "NoteOff before NoteOn"
    mergeNotes' stt stv ((t, MidiEvent c (NoteOn p v)) : es) =
       let (e, leftover) = searchNoteOff 0 stv 1 c p v es
       in  (t, e) : mergeNotes' stt stv leftover
    mergeNotes' stt stv ((et,e):es) =
       (et, Event e) : mergeNotes' stt stv es
```

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 -}
   Int -> Double -> {- SetTempo values: the one at the NoteOn and
                       the ratio between the current tempo and the first one. -}
   Channel -> Pitch -> Velocity ->
                  -- channel and pitch of NoteOn (NoteOff must match)
   Track ->       -- the remainder of the track to be searched
   (RichEvent', Track) -- the needed event and the remainder of the track
searchNoteOff int _   str c0 p0 v0 ((t1, MidiEvent c1 (NoteOff p1 v1)) : es)
      | c0 == c1 && p0 == p1  =
   let d = round (addInterval str t1 int)
       es' = case es of
                ((t2, me) : ess) -> (t1+t2, me) : ess
                [] -> []
   in  (Note d (v0,v1) c0 p0, es')
searchNoteOff int ost str c  p  v  (e@(t1, MetaEvent (SetTempo nst)) : es) =
   let (e', es') = searchNoteOff (addInterval str t1 int) ost
                       (fromIntegral ost / fromIntegral nst) c p v es
   in  (e', e : es')
searchNoteOff int ost str c  p  v  (e:es) =
   let (e', es') = searchNoteOff (addInterval str (fst e) int) ost str c p v es
   in  (e', e : es')
searchNoteOff _   _   _   _  _  _  [] =
   error "ReadMidi.searchNoteOff: no corresponding NoteOff"

addInterval :: Double -> Int -> Double -> Double
addInterval str t int = (int + fromIntegral t * str)
```

6.1.7 General Midi

```
module Haskore.Interface.MIDI.General where

import qualified Data.List  as List
import           Data.Array
import qualified Haskore.Interface.MIDI.File as MidiFile

type Name = String
type Table = [(Name, MidiFile.Program)]
```

```
numberToName :: MidiFile.Program -> Name
numberToName =
   (array (0,127)
      (List.map (\(x,y)->(y,x)) Haskore.Interface.MIDI.General.map) !)

numberToIName ::
   MidiFile.Channel -> MidiFile.Program -> Name
numberToIName ch num =
   if ch == 9 then "drums" else numberToName num

map :: Table
map = [
 ("Acoustic Grand Piano",0),       ("Bright Acoustic Piano",1),
 ("Electric Grand Piano",2),       ("Honky Tonk Piano",3),
 ("Rhodes Piano",4),               ("Chorused Piano",5),
 ("Harpsichord",6),                ("Clavinet",7),
 ("Celesta",8),                    ("Glockenspiel",9),
 ("Music Box",10),                 ("Vibraphone",11),
 ("Marimba",12),                   ("Xylophone",13),
 ("Tubular Bells",14),             ("Dulcimer",15),
 ("Hammond Organ",16),             ("Percussive Organ",17),
 ("Rock Organ",18),                ("Church Organ",19),
 ("Reed Organ",20),                ("Accordion",21),
 ("Harmonica",22),                 ("Tango Accordion",23),
 ("Acoustic Guitar (nylon)",24),   ("Acoustic Guitar (steel)",25),
 ("Electric Guitar (jazz)",26),    ("Electric Guitar (clean)",27),
 ("Electric Guitar (muted)",28),   ("Overdriven Guitar",29),
 ("Distortion Guitar",30),         ("Guitar Harmonics",31),
 ("Acoustic Bass",32),             ("Electric Bass (fingered)",33),
 ("Electric Bass (picked)",34),    ("Fretless Bass",35),
 ("Slap Bass 1",36),               ("Slap Bass 2",37),
 ("Synth Bass 1",38),              ("Synth Bass 2",39),
 ("Violin",40),                    ("Viola",41),
 ("Cello",42),                     ("Contrabass",43),
 ("Tremolo Strings",44),           ("Pizzicato Strings",45),
 ("Orchestral Harp",46),           ("Timpani",47),
 ("String Ensemble 1",48),         ("String Ensemble 2",49),
 ("Synth Strings 1",50),           ("Synth Strings 2",51),
 ("Choir Aahs",52),                ("Voice Oohs",53),
 ("Synth Voice",54),               ("Orchestra Hit",55),
 ("Trumpet",56),                   ("Trombone",57),
 ("Tuba",58),                      ("Muted Trumpet",59),
 ("French Horn",60),               ("Brass Section",61),
 ("Synth Brass 1",62),             ("Synth Brass 2",63),
 ("Soprano Sax",64),               ("Alto Sax",65),
```

```
 ("Tenor Sax",66),                ("Baritone Sax",67),
 ("Oboe",68),                     ("Bassoon",69),
 ("English Horn",70),             ("Clarinet",71),
 ("Piccolo",72),                  ("Flute",73),
 ("Recorder",74),                 ("Pan Flute",75),
 ("Blown Bottle",76),             ("Shakuhachi",77),
 ("Whistle",78),                  ("Ocarina",79),
 ("Lead 1 (square)",80),          ("Lead 2 (sawtooth)",81),
 ("Lead 3 (calliope)",82),        ("Lead 4 (chiff)",83),
 ("Lead 5 (charang)",84),         ("Lead 6 (voice)",85),
 ("Lead 7 (fifths)",86),          ("Lead 8 (bass+lead)",87),
 ("Pad 1 (new age)",88),          ("Pad 2 (warm)",89),
 ("Pad 3 (polysynth)",90),        ("Pad 4 (choir)",91),
 ("Pad 5 (bowed)",92),            ("Pad 6 (metallic)",93),
 ("Pad 7 (halo)",94),             ("Pad 8 (sweep)",95),
 ("FX1 (train)",96),              ("FX2 (soundtrack)",97),
 ("FX3 (crystal)",98),            ("FX4 (atmosphere)",99),
 ("FX5 (brightness)",100),        ("FX6 (goblins)",101),
 ("FX7 (echoes)",102),            ("FX8 (sci-fi)",103),
 ("Sitar",104),                   ("Banjo",105),
 ("Shamisen",106),                ("Koto",107),
 ("Kalimba",108),                 ("Bagpipe",109),
 ("Fiddle",110),                  ("Shanai",111),
 ("Tinkle Bell",112),             ("Agogo",113),
 ("Steel Drums",114),             ("Woodblock",115),
 ("Taiko Drum",116),              ("Melodic Drum",117),
 ("Synth Drum",118),              ("Reverse Cymbal",119),
 ("Guitar Fret Noise",120),       ("Breath Noise",121),
 ("Seashore",122),                ("Bird Tweet",123),
 ("Telephone Ring",124),          ("Helicopter",125),
 ("Applause",126),                ("Gunshot",127)]
```

6.2 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 *instruments* 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:

```
type Inst    = Int
type Name    = String
```

6.2.1 The Score File

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

import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Music as Music
import qualified Haskore.Music.Performance as Performance
import qualified Haskore.Music.Player as Player
import System.IO
import Data.List (find)
import Haskore.General.Utility (flattenTuples2, flattenTuples3, flattenTuples4)
import Haskore.Interface.CSound (Name, Inst)
```

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  Inst StartTime Duration Pch Volume [Pfield]
               | Table Table CreatTime TableSize Normalize GenRoutine
     deriving Show

type Bpm       = Int
type StartTime = Float
type Duration  = Float
data Pch       = AbsPch Pitch.Absolute | Cps Float deriving Show
type Volume    = Float
type Pfield    = Float
type Table     = Int
type CreatTime = Float
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 `GenRoutine`:

```
data GenRoutine = GenRoutine GenNum [GenArg]
                | SoundFile SFName SkipTime ChanNum
     deriving Show

type SFName   = String
type SkipTime = Float
```

```
type ChanNum  = Float
type GenNum   = Int
type GenArg   = Float
```

`GenRoutine 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 6.2.2). 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 -> GenRoutine
soundFile = SoundFile
```

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

```
tableValues :: [GenArg] -> GenRoutine
tableValues gas = GenRoutine 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 -> GenRoutine
polynomial (x1,x2) cfs = GenRoutine 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)] -> GenRoutine
exponential1 sp xs = GenRoutine 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] -> GenRoutine
exponential2 pts = GenRoutine 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)] -> GenRoutine
cubic sp pts = GenRoutine 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)] -> GenRoutine
lineSeg1 sp pts = GenRoutine 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] -> GenRoutine
lineSeg2 pts = GenRoutine 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)] -> GenRoutine
cubicSpline sp pts = GenRoutine 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] -> GenRoutine
compSine1 pss = GenRoutine 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)] -> GenRoutine
compSine2 args = GenRoutine 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)] -> GenRoutine
compSine3 args = GenRoutine 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 -> GenRoutine
cosineHarms n l m = GenRoutine 11 [fromIntegral n, fromIntegral l, m]

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

GEN21. Produces tables having selected random distributions.

```
randomTable :: RandDist -> GenRoutine
randomTable rd = GenRoutine 21 [fromIntegral rd]

type RandDist = Int

uniform, linear, triangular, expon,
  biexpon, gaussian, cauchy, posCauchy :: Int
uniform    = 1
linear     = 2
triangular = 3
expon      = 4
biexpon    = 5
gaussian   = 6
cauchy     = 7
posCauchy  = 8
```

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.

```
type NameMap = [(Name, Int)]
```

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:

```
makeNameMap :: [Name] -> NameMap
makeNameMap nms = zip nms [1..]
```

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

```
addToMap :: NameMap -> Name -> Int -> NameMap
addToMap nmap nm i =
  case (getId nmap nm) of
    Nothing -> (nm,i) : nmap
    Just _  -> error (" addToMap: the name " ++ nm ++
                                    " is already in the name map")

getId :: NameMap -> Name -> Maybe Int
getId nmap nm =
  fmap snd (find (\(n,_) -> n==nm) nmap)
```

Note the use of the function `find` imported from the `List` library.

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:

```
fromPerformance :: NameMap -> Performance.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:

```
fromMusic :: NameMap -> Player.Map
                  -> Tables -> Performance.Context -> Music.T -> IO ()
fromMusic nmap pmap tables cont m =
  saveIA (tables ++ fromPerformance nmap (Performance.fromMusic pmap cont m))

type Tables = T
```

The `Tables` argument is a user-defined set of function tables, represented as a sequence of `Statement`s (specifically, `Table` constructors). (See Section 6.2.1.)

From Performance.T to Score.T The translation between `Performance.Event`s and score `CSoundScore.Note`s 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.

```
fromPerformance nmap pf =
   zipWith (statementFromEvent nmap)
           (tail (scanl (+) 0 (map Performance.eTime pf))) pf

statementFromEvent :: NameMap -> StartTime -> Performance.Event -> Statement
statementFromEvent nmap t (Performance.Event _ i p d v pfs) =
  case (getId nmap i) of
    Nothing  -> error ("CSoundScore.statementFromEvent: instrument " ++ i ++ " is u
    Just num -> Note num t d (AbsPch p) (velocityToDb v) pfs
```

```
velocityToDb :: Float -> Float
velocityToDb = (50*)

-- still unused, but it should be implemented this way
amplitudeToDb :: Float -> Float
amplitudeToDb v = 6 * logBase 2 v
```

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
```

In the following we will come across several instances where we will need to write a list of floating point numbers into the file, one number at a time, separated by spaces. To do this, we will need to convert the list to a string. This is done by the following function:

```
listToString :: [Float] -> String
listToString []       = ""
listToString (n : ns) = " " ++ show n ++ listToString ns
```

Finally, the `statementToString` function:

```
statementToString :: Statement -> String
statementToString (Tempo t) =
   "t 0 " ++ show t
statementToString (Note i st d p v pfs) =
   "i " ++ show i ++ " " ++ show st ++ " " ++ show d ++ " " ++
           pchToString p ++ " " ++ show v ++ listToString pfs
statementToString (Table t ct s n gr)   =
```

```
   "f " ++ show t ++ " " ++ show ct ++ " " ++ show s ++
           (if n then " " else " -") ++ showGenRoutine gr
  where showGenRoutine (SoundFile nm st cn) =
                       "1 " ++ nm ++ " " ++ show st ++ " 0 " ++ show cn
        showGenRoutine (GenRoutine gn gas) =
                       show gn ++ listToString gas

{- Offset to map from Haskore's pitch 0
   to the corresponding pitch of CSound -}
zeroKey :: Int
zeroKey = 84

-- 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+zeroKey) 12
   in  show oct ++ "." ++ (if 0 < note && note < 10 then "0" else "") ++ show note
pchToString (Cps freq) = show freq
```

6.2.2 The Orchestra File

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

import System.IO
import Data.List (find, nub, intersperse)
import Data.Ix (Ix)
import Data.Array (listArray, (!))
import Haskore.General.Utility (flattenTuples2)
import Haskore.Interface.CSound (Name, Inst)
```

The orchestra file consists of two parts: a *header*, and one or more *instrument blocks*. The header sets global parameters controlling sampling rate, control rate, and number of output channels. 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:

```
type T = (Header, [InstBlock])
```

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

```
type Header = (AudRate, CtrlRate, Chnls)
```

```
type AudRate  = Int  -- samples per second
type CtrlRate = Int  -- samples per second
type Chnls    = Int  -- mono=1, stereo=2, surround=4
```

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 a unique identifying integer, as well as an *orchestra expression* that defines the audio signal characterizing that instrument:

```
type InstBlock = (Inst, Expression)
```

Recall that `Inst` 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 `Expression` 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 `Expression`. 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 `Expression` data type declaration is shown in Figure 11. In what follows, we describe each of the various constructors in turn.

Constants `Const 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, noteVol :: Expression
noteDur = Pfield 3
notePit = Pfield 4
noteVol = Pfield 5
```

```
data Expression = Const Float
            | Pfield Int

            | Function  Function  Expression
            | Function2 Function2 Expression Expression

            | Comparison Comparison Expression Expression Expression Expression
            | MonoOut       Expression
            | LeftOut       Expression
            | RightOut      Expression
            | StereoOut     Expression Expression
            | FrontLeftOut  Expression
            | FrontRightOut Expression
            | RearRightOut  Expression
            | RearLeftOut   Expression
            | QuadOut       Expression Expression Expression Expression

            | Line       EvalRate Start Durn Finish
            | Expon      EvalRate Start Durn Finish
            | LineSeg    EvalRate Start Durn Finish [(Durn, Finish)]
            | ExponSeg   EvalRate Start Durn Finish [(Durn, Finish)]
            | Env        EvalRate Sig RTime Durn DTime RShape SAttn DAttn Steep
            | Phasor     EvalRate Freq InitPhase
            | TblLookup  EvalRate Index Table IndexMode
            | TblLookupI EvalRate Index Table IndexMode
            | Osc        EvalRate Amp Freq Table
            | OscI       EvalRate Amp Freq Table
            | FMOsc      Amp Freq CarFreq ModFreq ModIndex Table
            | FMOscI     Amp Freq CarFreq ModFreq ModIndex Table
            | SampOsc    Amp Freq Table
            | Random     EvalRate Amp
            | RandomHold EvalRate Amp HoldHz
            | RandomI    EvalRate Amp HoldHz
            | GenBuzz    Amp Freq NumHarms LoHarm Multiplier Table
            | Buzz       Amp Freq NumHarms Table
            | Pluck      Amp Freq Table DecayMethod DecArg1 DecArg2
            | Delay      MaxDel AudioSig
            | DelTap     TapTime DelLine
            | DelTapI    TapTime DelLine
            | DelayW     AudioSig
            | Comb       AudioSig RevTime LoopTime
            | AlPass     AudioSig RevTime LoopTime
            | Reverb     AudioSig RevTime
    deriving (Show, Eq)
```

Figure 11: The `Expression` Data Type

```
data Function =
               Int
             | Frac
             | Neg
             | Abs
             | Sqrt
             | Sin
             | Cos
             | Tan
             | SinInv
             | CosInv
             | TanInv
             | SinH
             | CosH
             | TanH
             | Exp
             | Log

             | AmpToDb
             | DbToAmp
             | PchToHz
             | HzToPch
     deriving (Show, Eq, Ord, Ix)

data Function2 =
               Plus
             | Minus
             | Times
             | Divide
             | Power
             | Modulo
     deriving (Show, Eq, Ord, Ix)

data Comparison =
               GreaterThan
             | LessThan
             | GreaterOrEqTo
             | LessOrEqTo
             | Equals
             | NotEquals
     deriving (Show, Eq, Ord, Ix)
```

Figure 12: Smaller building blocks of CSound expressions

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 :: Expression
p1 = Pfield 6
p2 = Pfield 7
p3 = Pfield 8
p4 = Pfield 9
p5 = Pfield 10
p6 = Pfield 11
p7 = Pfield 12
p8 = Pfield 13
p9 = Pfield 14
```

Arithmetic and Transcendental Functions Arithmetic expressions are represented by the constructors `Plus`, `Minus`, `Times`, `Divide`, `Power`, and `Modulo`, each taking two `Expression`s as arguments. In addition, there are a number of unary arithmatic functions: `Int x` and `Frac x` represent the integer and fractional parts, respectively, of `x`. `Abs x`, `Neg x`, `Sqrt x`, `Sin x`, and `Cos x` represent the absolute value, negation, square root, sine and cosine (in radians) of `x`, respectively. `Exp x` represents e raised to the power `x`, and `Log x` is the natural logarithm of `x`.

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

```
instance Num Expression where
  (+)    = Function2 Plus
  (-)    = Function2 Minus
  (*)    = Function2 Times
  negate = Function Neg
  abs    = Function Abs
  signum x = x / abs x
     -- should be replaced by something more sophisticated
  fromInteger = Const . fromInteger

instance Fractional Expression where
  (/) = Function2 Divide
  fromRational = Const . fromRational

instance Floating Expression where
 exp   = Function Exp
 log   = Function Log
 sqrt  = Function Sqrt
 (**)  = Function2 Power
 pi    = Const pi
 sin   = Function Sin
 cos   = Function Cos
 tan   = Function Tan
```

```
asin  = Function SinInv
acos  = Function CosInv
atan  = Function TanInv
sinh  = Function SinH
cosh  = Function CosH
tanh  = Function 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
```

For example, `Plus (Pfield 3) (Power (Sin (Pfield 6)) (Const 2))` can now be written simply as `noteDur + sin p6 ** 2`.

Pitch and Volume Coercions The next set of constructors represent functions that convert between different CSound pitch and volume representations. Recall that the `Score.Note` constructor uses decibels for volume and "pch" notation for pitch. If these values are to be used as inputs into a signal generating or modifying routine, they must first be converted into units of raw amplitude and hertz, respectively. This is accomplished by the functions represented by `DbToAmp` and `PchToHz`, and their inverses are `AmpToDb` and `HzToPch`. Thus note that `PchToHz (notePit + 0.01)` raises the pitch by one semitone, whereas `PchToHz notePit + 0.01` raises the pitch by 0.01 Hz.

Comparison Operators `Expression` also includes comparison constructors `GreaterThan`, `LessThan`, `GreaterOrEqTo`, `LessOrEqTo`, `Equals`, and `NotEquals`. Each takes four `Expression` arguments: the values of the first two are compared, and if the result is true, the expression evaluates to the third argument; otherwise, it takes on the value of the fourth.[6]

Output Operators The next group of constructors represent CSound's *output statements*. The constructors are `MonoOut`, `LeftOut`, `RightOut`, `StereoOut`, `FrontLeftOut`, `FrontRightOut`, `RearRightOut`, `RearLeftOut`, and `QuadOut`. `StereoOut` takes two `Expression` arguments, `QuadOut` takes four, and the rest take one.

The top-level of an instrument's `Expression` (i.e., the one in the `InstBlock` value) will normally be an application of one of these. Furthermore, the constructor used must be in agreement with the number of output channels specified in the orchestra header—for example, using `LeftOut` when the header declares the resulting sound file to be mono will result in an error.

Signal Generation and Modification The most sophisticated `Expression` constructors are those that emulate CSound's signal generation and modification functions. There are quite a few of them, and they are all described here, although the reader is encouraged to read the CSound manual for further details.

[6] This design emulates that of CSound. A more conventional design would have comparison operators that return a Boolean value, and a conditional expression to choose between two values based on a Boolean. One could then add Boolean operators such as "and", "or", etc. It seems possible to do this in Haskore, but its translation into CSound would be more difficult, and thus we take the more conservative approach for now.

Before defining each constructor, however, there are two general issues to discuss:

First, 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)
```

Second, note in Figure 11 that this collection of constructors uses quite a few other type names. In all cases, however, these are simply type synonyms for `Expression`, and are used only for clarity. These type synonyms are listed here in one fell swoop:

```
type Start       = Expression
type Durn        = Expression
type Finish      = Expression
type Sig         = Expression
type RTime       = Expression
type DTime       = Expression
type RShape      = Expression
type SAttn       = Expression
type DAttn       = Expression
type Steep       = Expression
type Freq        = Expression
type InitPhase   = Expression
type Index       = Expression
type Table       = Expression
type IndexMode   = Expression
type Amp         = Expression
type CarFreq     = Expression
type ModFreq     = Expression
type ModIndex    = Expression
type HoldHz      = Expression
type NumHarms    = Expression
type LoHarm      = Expression
type Multiplier  = Expression
type DecayMethod = Expression
type DecArg1     = Expression
type DecArg2     = Expression
type MaxDel      = Expression
type AudioSig    = Expression
type TapTime     = Expression
type DelLine     = Expression
type RevTime     = Expression
```

```
type LoopTime    = Expression
```

We can now discuss each constructor in turn:

1. `Line evalrate start durn 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 `durn` seconds.

2. `Expon` is similar to `Line`, but `Expon evalrate start durn finish` produces an exponential curve instead of a straight line.

3. If a more elaborate signal is required, one can use the constructors `LineSeg` or `ExponSeg`, which take arguments of type `EvalRate`, `Start`, `Durn`, and `Finish`, as above, and also `[(Durn, Finish)]`. The first four arguments work as before, but only for the first of a number of segments. The subsequent segment lengths and endpoints are given in the fifth argument. A signal containing both straight line and exponential segments can be obtained by adding a `LineSeg` signal and `ExponSeg` signal together in an appropriate way.

4. `Env evalrate sig rtime durn dtime rshape sattn dattn steep` 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 exopnentially 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.

5. `Phasor evalrate freq initphase` 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, `Phasor` can simulate the behavior of an oscillator.

6. Table lookup constructors `TblLookup` and `TblLookupI` both take `EvalRate`, `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:

   ```
   rawIndex, normalIndex :: Expression
   rawIndex    = 0.0
   normalIndex = 1.0
   ```

   Both `TblLookup` and `TblLookupI` return values stored in the specified table at the given index. The difference is that `TblLookupI` 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.

   As mentioned, the output of a `Phasor` can be used as input to a table lookup to simulate an oscillator whose frequencey is controlled by the note pitch. This can be accomplished easily by the following piece of Haskore code:

[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.

```
      in  TblLookupI AR index table normalIndex
```

where `table` is some given function table ID. If `osc` is given as argument to an output operator such as `MonoOut`, then this `Expression` coupled with an instrument ID number (say, 1) produces a complete instrument block:

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, `Osc` and `OscI`, which differ in the same way as `TblLookup` and `TblLookupI`. Each oscillator constructor takes arguments of type `EvalRate`, `Amp` (in raw amplitude), `Freq` (in Hertz), and `Table`. The result is a signal that oscillates through the function table at the given frequency. Thus the following value is equivalent to `osc` 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*. `FMOsc amp freq carfreq modfreq modindex table` is 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. `FMOscI` behaves similarly. Note that there is no `EvalRate` argument, since these functions work at audio rate only. The given function table normally contains a sine wave. This oscillator setup is known as the *chowning FM* setup.

9. `SampOsc amp freq table` 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.

10. `Random evalrate amp` produces a random number series between `-amp` and `+amp` at either control or audio rate. `RandomHold evalrate amp holdhz` does the same but will hold each number for `holdhz` cycles before generating a new one. `RandomI evalrate amp holdhz` will in addition provide straight line interpolation between successive numbers.

    All the remaining constructors only operate at audio rate, and thus do not have `EvalRate` arguments.

11. `GenBuzz amp freq numharms loharm multiplier table` 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.

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.

    Note that the above two constructors have an analog in the generating routine GEN11 and the related function `cosineHarms` (see Section 6.2.1). `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 amp freq table decaymethod decarg1 decarg2` 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 `freq`. It is produced by iterating through an internal buffer that initially contains a copy of `table` and is smoothed on every pass 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:

```
simpleSmooth, stretchSmooth, simpleDrum, stretchDrum,
  weightedSmooth, filterSmooth :: Expression
simpleSmooth   = 1.0
stretchSmooth  = 2.0
simpleDrum     = 3.0
stretchDrum    = 4.0
weightedSmooth = 5.0
filterSmooth   = 6.0
```

14. `Delay deltime audiosig` establishes a digital delay line, where `audiosig` is the source, and `deltime` is the delay time in seconds.

    The delay line can also be *tapped* by `DelayTap deltime delline` and `DelayTapI deltime delline`, where `deltime` is the tap delay, and `delline` must be a delay line created by the `Delay` constructor above. Again, `DelayTapI` uses interpolation, and may take up to twice as long as `DelayTap` to run, but produces higher precision results and thus a cleaner signal.

    (Note: the constructor `DelayW` is used in the translation described later to mark the end of a sequence of delay taps, and is not intended for use by the user.)

15. Reverberation can be added to a signal using `Comb audiosig revtime looptime`, `AlPass audiosig revtime looptime`, and `Reverb audiosig revtime`. `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.

```
mkList :: Expression -> [(EvalRate, Expression)]
mkList (Const _)                  = []
mkList (Pfield _)                 = []
mkList (Function  _ x)            = mkList x
mkList (Function2 _ x1 x2)        = mkListAll [x1,x2]
mkList (Comparison _  x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
mkList (QuadOut       x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
mkList (StereoOut x1 x2)          = mkListAll [x1,x2]
mkList (MonoOut x)                = mkList x
mkList (LeftOut x)                = mkList x
mkList (RightOut x)               = mkList x
mkList (FrontLeftOut x)           = mkList x
mkList (FrontRightOut x)          = mkList x
mkList (RearRightOut x)           = mkList x
mkList (RearLeftOut x)            = mkList x
mkList oe@(Line     er x1 x2 x3)   = (er,oe) : mkListAll [x1,x2,x3]
mkList oe@(Expon    er x1 x2 x3)   = (er,oe) : mkListAll [x1,x2,x3]
mkList oe@(LineSeg  er x1 x2 x3 xs)
                  = (er,oe) : mkListAll (x1:x2:x3: flattenTuples2 xs)
mkList oe@(ExponSeg er x1 x2 x3 xs)
                  = (er,oe) : mkListAll (x1:x2:x3: flattenTuples2 xs)
mkList oe@(Env er x1 x2 x3 x4 x5 x6 x7 x8)
                  = (er,oe) : mkListAll [x1,x2,x3,x4,x5,x6,x7,x8]
mkList oe@(Phasor    er x1 x2)     = (er,oe) : mkListAll [x1,x2]
mkList oe@(TblLookup er x1 x2 x3)  = (er,oe) : mkListAll [x1,x2,x3]
mkList oe@(TblLookupI er x1 x2 x3) = (er,oe) : mkListAll [x1,x2,x3]
mkList oe@(Osc       er x1 x2 x3)  = (er,oe) : mkListAll [x1,x2,x3]
mkList oe@(OscI      er x1 x2 x3)  = (er,oe) : mkListAll [x1,x2,x3]
mkList oe@(Random       er x)      = (er,oe) : mkList x
mkList oe@(RandomHold   er x1 x2)  = (er,oe) : mkListAll [x1,x2]
mkList oe@(RandomI      er x1 x2)  = (er,oe) : mkListAll [x1,x2]
mkList oe@(FMOsc x1 x2 x3 x4 x5 x6)= (AR,oe) : mkListAll [x1,x2,x3,x4,x5,x6]
mkList oe@(FMOscI x1 x2 x3 x4 x5 x6)
                                   = (AR,oe) : mkListAll [x1,x2,x3,x4,x5,x6]
mkList oe@(SampOsc x1 x2 x3)       = (AR,oe) : mkListAll [x1,x2,x3]
mkList oe@(GenBuzz x1 x2 x3 x4 x5 x6)
                                   = (AR,oe) : mkListAll [x1,x2,x3,x4,x5,x6]
mkList oe@(Buzz x1 x2 x3 x4)       = (AR,oe) : mkListAll [x1,x2,x3,x4]
mkList oe@(Pluck x1 x2 x3 x4 x5 x6)= (AR,oe) : mkListAll [x1,x2,x3,x4,x5,x6]
mkList oe@(Delay x1 x2)            = (AR,oe) : mkListAll [x1,x2]
mkList oe@(DelTap  _ x2)           = (AR,oe) : mkList x2
mkList oe@(DelTapI _ x2)           = (AR,oe) : mkList x2
mkList    (DelayW _)               = error "DelayW not for you!"
mkList oe@(Comb x1 x2 x3)          = (AR,oe) : mkListAll [x1,x2,x3]
mkList oe@(AlPass x1 x2 x3)        = (AR,oe) : mkListAll [x1,x2,x3]
mkList oe@(Reverb x1 x2)           = (AR,oe) : mkListAll [x1,x2]
```

Figure 13: The `mkList` Function

Converting Orchestra Values to Orchestra Files We must now convert the `Expression` 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 13, and creates an entry in the list for every signal generating, modifying, or outputting constructor. It uses the following auxiliary functions:

```
mkListAll :: [Expression] -> [(EvalRate, Expression)]
mkListAll = foldr (++) [] . map mkList

addNames :: [(EvalRate,Expression)] -> [(Name,Expression)]
addNames ls = zipWith counter ls [(1::Int)..]
```

```
                where counter (er,x) n =
                        let var = case er of
                                    AR -> 'a' : show n
                                    CR -> 'k' : show n
                                    NR -> 'i' : show n
                        in (var,x)
```

Putting all of the above together, here is a function that converts an `Expression` into a list of proper name / `Expression` pairs. Each one of these will eventually result in one statement in the CSound orchestra file. (The result of `mkList` 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.)

```
processExp :: Expression -> [(Name, Expression)]
processExp = addNames . nub . procDelay . reverse . mkList
```

The function `procDelay` is used to process delay lines, which require special treatment because a delay line followed by a number of taps must be ended in Csound with a `DelayW` command.

```
procDelay :: [(EvalRate, Expression)] -> [(EvalRate, Expression)]
procDelay []                          = []
procDelay (x@(AR, d@(Delay _ _)) : xs) = [x] ++ procTaps d xs ++ procDelay xs
procDelay (x : xs)                    = x : procDelay xs

procTaps :: Expression -> [(EvalRate, Expression)] -> [(EvalRate, Expression)]
procTaps (Delay _ sig) []           = [(AR, DelayW sig)]
procTaps d (x@(AR,DelTap t dl) : xs)  =
          if d == dl then (mkList t ++ [x] ++ procTaps d xs)
                     else procTaps d xs
procTaps d (x@(AR,DelTapI t dl): xs)  =
          if d == dl then (mkList t ++ [x] ++ procTaps d xs)
                     else procTaps d xs
procTaps d (_ : xs)                   = procTaps d xs
procTaps _ _ = error "unhandled case in procTaps"
```

The functions that follow are used to write the orchestra file. `saveIA` is similar to `saveScoreIA`: 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 :: T -> IO ()
saveIA orch =
   do putStr "\nName your orchestra file "
      putStr "(.orc extension will be added): "
      name <- getLine
      save name orch

save :: FilePath -> T -> 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 :: T -> String
toString (hdr,ibs) =
   headerToString hdr ++ concatMap instBlockToString ibs
```

Writing the header is relatively simple, and is accomplished by the following function:

```
headerToString :: Header -> String
headerToString (a,k,nc) =
     "sr     = " ++ show a ++
   "\nkr     = " ++ show k ++
   "\nksmps  = " ++ show (fromIntegral a / fromIntegral k :: Double) ++
   "\nnchnls = " ++ show nc ++
   "\n"
```

`instBlockToString` writes a single instrument block.

```
instBlockToString :: InstBlock -> String
instBlockToString (num,ox) =
     "\ninstr " ++ show num ++ "\n" ++
     writeExps (processExp ox ++ [("",ox)]) ++
     "endin\n"
```

Recall that after processing, the `Expression` becomes a list of `(Name, Expression)` pairs. The last few functions write each of these named `Expression`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, Expression)` pairs.

6.2.3 An Orchestra Example

Figure 16 shows a typical CSound orchestra file. Figure 17 shows how this same functionality would be achieved in Haskore using an `CSound.Orchestra.T` value. Finally, Figure 18 shows the result of applying `Orchestra.saveIA` to `orc1` shown in Figure 17. Figures 16 and 18 should be compared: you will note that except for name changes, they are the same, as they should be.

6.2.4 Tutorial

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

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

```
writeExps :: [(Name,Expression)] -> String
writeExps noes = concatMap writeExp noes where
   writeExp :: (Name,Expression) -> String
   writeExp (_,  MonoOut x)           = "out "   ++ writeArgs [x]
   writeExp (_,  LeftOut x)           = "outs1 " ++ writeArgs [x]
   writeExp (_,  RightOut x)          = "outs2 " ++ writeArgs [x]
   writeExp (_,  StereoOut x1 x2)     = "outs "  ++ writeArgs [x1,x2]
   writeExp (_,  FrontLeftOut x)      = "outq1 " ++ writeArgs [x]
   writeExp (_,  FrontRightOut x)     = "outq2 " ++ writeArgs [x]
   writeExp (_,  RearRightOut x)      = "outq3 " ++ writeArgs [x]
   writeExp (_,  RearLeftOut x)       = "outq4 " ++ writeArgs [x]
   writeExp (_,  QuadOut x1 x2 x3 x4) = "outq "  ++ writeArgs [x1,x2,x3,x4]
   writeExp (nm, Line _ x1 x2 x3) =
     (nm ++ " line ")    ++ writeArgs [x1,x2,x3]
   writeExp (nm, Expon _ x1 x2 x3) =
     (nm ++ " expon ")   ++ writeArgs [x1,x2,x3]
   writeExp (nm, LineSeg _ x1 x2 x3 xlist)  =
     (nm ++ " linseg ")  ++ writeArgs (x1:x2:x3: flattenTuples2 xlist)
   writeExp (nm, ExponSeg _ x1 x2 x3 xlist) =
     (nm ++ " expseg ")  ++ writeArgs (x1:x2:x3: flattenTuples2 xlist)
   writeExp (nm, Env _ x1 x2 x3 x4 x5 x6 x7 x8) =
     (nm ++ " envlpx ")  ++ writeArgs [x1,x2,x3,x4,x5,x6,x7,x8]
   writeExp (nm, Phasor _ x1 x2) =
     (nm ++ " phasor ")  ++ writeArgs [x1,x2]
   writeExp (nm, TblLookup _ x1 x2 x3) =
     (nm ++ " table ")   ++ writeArgs [x1,x2,x3]
   writeExp (nm, TblLookupI _ x1 x2 x3) =
     (nm ++ " tablei ")  ++ writeArgs [x1,x2,x3]
   writeExp (nm, Osc _ x1 x2 x3) =
     (nm ++ " oscil ")   ++ writeArgs [x1,x2,x3]
   writeExp (nm, OscI _ x1 x2 x3) =
     (nm ++ " oscili ")  ++ writeArgs [x1,x2,x3]
   writeExp (nm, FMOsc x1 x2 x3 x4 x5 x6) =
     (nm ++ " foscil ")  ++ writeArgs [x1,x2,x3,x4,x5,x6]
   writeExp (nm, FMOscI x1 x2 x3 x4 x5 x6) =
     (nm ++ " foscili ") ++ writeArgs [x1,x2,x3,x4,x5,x6]
   writeExp (nm, SampOsc x1 x2 x3) =
     (nm ++ " loscil ")  ++ writeArgs [x1,x2,x3]
   writeExp (nm, Random _ x) =
     (nm ++ " rand ")    ++ writeArgs [x]
   writeExp (nm, RandomHold _ x1 x2) =
     (nm ++ " randh ")   ++ writeArgs [x1,x2]
   writeExp (nm, RandomI _ x1 x2) =
     (nm ++ " randi ")   ++ writeArgs [x1,x2]
   writeExp (nm, GenBuzz x1 x2 x3 x4 x5 x6) =
     (nm ++ " gbuzz ")   ++ writeArgs [x1,x2,x3,x4,x5,x6]
   writeExp (nm, Buzz x1 x2 x3 x4) =
     (nm ++ " buzz ")    ++ writeArgs [x1,x2,x3,x4]
   writeExp (nm, Pluck x1 x2 x3 x4 x5 x6) =
     (nm ++ " pluck ")   ++ writeArgs [x1,x2,x2,x3,x4,x5,x6]
   writeExp (nm, Delay x1 _)      = (nm ++ " delayr ")  ++ writeArgs [x1]
   writeExp (_,  DelayW x)        = ("    delayw ")     ++ writeArgs [x]
   writeExp (nm, DelTap x1 _)     = (nm ++ " deltap ")  ++ writeArgs [x1]
   writeExp (nm, DelTapI x1 _)    = (nm ++ " deltapi ") ++ writeArgs [x1]
   writeExp (nm, Comb x1 x2 x3)   = (nm ++ " comb ")    ++ writeArgs [x1,x2,x3]
   writeExp (nm, AlPass x1 x2 x3) = (nm ++ " alpass ")  ++ writeArgs [x1,x2,x3]
   writeExp (nm, Reverb x1 x2)    = (nm ++ " reverb ")  ++ writeArgs [x1,x2]
   writeExp _ = error "writeExp: unknown constructor\n"

   writeArgs :: [Expression] -> String
   writeArgs = (++"\n") . concat . intersperse ", " . map (showExp noes)
```

Figure 14: The Function `writeExp`

```
showExp :: [(Name, Expression)] -> Expression -> String
showExp _ (Const x)         = show x
showExp _ (Pfield p)        = "p" ++ show p
showExp xs (Function  f x)  = showFunc xs f x
showExp xs (Function2 f x1 x2) = showBin  xs f x1 x2
showExp xs (Comparison c x1 x2 x3 x4) = showComp xs c x1 x2 x3 x4
showExp xs ox = case find (\(_,oexp) -> ox==oexp) xs of
                  Just (nm,_) -> nm
                  Nothing     -> error ("showExp "++show ox++": constructor not fou

showFunc :: [(Name, Expression)] -> Function -> Expression -> String
showFunc xs HzToPch x = "pchoct (octcps(" ++ showExp xs x ++ "))"
showFunc xs f x =
   let s = listArray (Int,PchToHz)
             ["int",    "frac",
              "abs",    "-",      "sqrt",
              "sin",    "cos",    "tan",
              "sininv", "cosinv", "taninv",
              "sinh",   "cosh",   "tanh",
              "exp",    "log",
              "dbamp",  "ampdb",
              "cpspch"] ! f
   in  s ++ "(" ++ showExp xs x ++ ")"

showBin :: [(Name, Expression)] -> Function2 -> Expression -> Expression -> String
showBin xs f x1 x2 =
   let s = listArray (Plus,Modulo)
               ["+", "-", "*", "/", "^", "%"] ! f
   in  "(" ++ showExp xs x1 ++ " " ++ s ++ " " ++ showExp xs x2 ++ ")"

showComp :: [(Name, Expression)] -> Comparison
                -> Expression -> Expression -> Expression -> Expression -> String
showComp xs c x1 x2 x3 x4 =
   let s = listArray (GreaterThan,NotEquals)
               [">", "<", ">=", "<=", "==", "!="] ! c
   in  "(" ++ showExp xs x1 ++ " " ++ s ++ " " ++ showExp xs x2 ++ " ? " ++
               showExp xs x3 ++ " : " ++ showExp xs x4 ++ ")"
```

Figure 15: The Function showExp

```
 = 48000
 = 24000
mps = 2
hnls = 2

str 4

ote = cpspch(p5)

 envlpx ampdb(p4), .001, p3, .05, 6, -.1, .01
 envlpx ampdb(p4), .0005, .1, .1, 6, -.05, .01
 envlpx ampdb(p4), .001, p3, p3, 6, -.3, .01

 oscili k1, inote, 1
 oscili k1, inote * 1.004, 1
 oscili k2, inote * 16, 1
 oscili k3, inote, 5
 oscili k3, inote * 1.004, 5

ts  (a2 + a3 + a4) * .75, (a1 + a3 + a5) * .75

din
```

Figure 16: Sample CSound Orchestra File

```
pchToHz, dbToAmp :: Expression -> Expression
pchToHz = Function PchToHz
dbToAmp = Function DbToAmp

orc1 :: T
orc1 =
  let hdr   = (48000, 24000, 2)
      inote = pchToHz p5
      k1    = Env  CR (dbToAmp p4) 0.001  p3  0.05 6 (-0.1)  0.01 0
      k2    = Env  CR (dbToAmp p4) 0.0005 0.1 0.1  6 (-0.05) 0.01 0
      k3    = Env  CR (dbToAmp p4) 0.001  p3  p3   6 (-0.3)  0.01 0
      a1    = OscI AR k1  inote        1
      a2    = OscI AR k1 (inote*1.004) 1
      a3    = OscI AR k2 (inote*16)    1
      a4    = OscI AR k3  inote        5
      a5    = OscI AR k3 (inote*1.004) 5
      out   = StereoOut ((a2+a3+a4) * 0.75) ((a1+a3+a5) * 0.75)
      ib    = (4,out)
  in (hdr,[ib])

t1 :: [(Name, Expression)]
t1 = processExp (snd (head (snd orc1)))
```

Figure 17: Haskore Orchestra Definition

```
     = 48000
     = 24000
mps  = 2.0
hnls = 2

str 4
 envlpx ampdb(p4), 0.001, p3, p3, 6.0, -0.3, 0.01, 0.0
 osci k1, (cpspch(p5) * 1.004), 5.0
 envlpx ampdb(p4), 0.0005, 0.1, 0.1, 6.0, -0.05, 0.01, 0.0
 osci k3, (cpspch(p5) * 16.0), 1.0
 envlpx ampdb(p4), 0.001, p3, 0.05, 6.0, -0.1, 0.01, 0.0
 osci k5, cpspch(p5), 1.0
 osci k1, cpspch(p5), 5.0
 osci k5, (cpspch(p5) * 1.004), 1.0
ts (((a8 + a4) + a7) * 0.75), (((a6 + a4) + a2) * 0.75)
din
```

Figure 18: Result of `Orchestra.saveIA orc1`

```
import System.IO
import System.Cmd( system )

          import Haskore.Interface.CSound(Name)
import Haskore.Interface.CSound.Orchestra as Orchestra
import Haskore.Interface.CSound.Score     as Score

type OrcExp = Orchestra.Expression
```

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 CSound module:

```
pureToneTN :: Int
pureToneTN = 1
pureToneTable :: Orchestra.Table
pureToneTable = fromIntegral pureToneTN
pureTone :: Score.Statement
pureTone = Score.Table pureToneTN 0 8192 True (compSine1 [1.0])
```

`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 `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:

```
tune1 :: Music.T

tune1 = line (map (flip ($) [Music.Velocity 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])
```

Recall that the tune, a value of type `Music.T`, must first be converted to a value of type `Performance` using the function `Performance.fromMusic` defined in the module `Music`, and then the `Performance` must turned into a `Score.T`, using `perfToScore` defined in the `CSound` module. Since the function `perform` expects to see a *name map*, we must create one. We won't be using more than two simple instruments in this tutorial:

```
instNames :: NameMap
instNames = [("inst1", 1), ("inst2", 2)]
```

```
inst1, inst2 :: Int
inst1 = 1
inst2 = 2
```

We can now create a complete Score.T as follows:

```
context :: Context.T
context =
   Context.setInstrument "" $
   Context.setDur 1 $
   Context.deflt

scored :: Music.T -> Score.T
scored m = Score.fromPerformance instNames
               (Performance.fromMusic Player.fancyMap context m)

score1 = pureTone : scored (setInstrument "inst1" tune1)
```

Let's define an instrument in the orchestra file that will use the function table `pureTone`:

```
oe1 :: OrcExp
oe1 = let signal = Osc AR (dbToAmp noteVol) (pchToHz notePit) pureToneTable
      in  StereoOut signal signal
```

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 `noteVol`. Note that the `oe1` expression above is an `OrcExp`, not a complete `Orchestra.T`. We need to define a *header* and associate `oe1` with the instrument that's playing it:

```
hdr :: (Int, Int, Int)
hdr = (44100, 4410, 2)

o1, o2, o3, o4, o5, o6, o7, o8, o9, o10,
 o11, o12, o13, o14, o15, o16, o17, o18, o19, o20
   :: ((Int, Int, Int), [(Int, Orchestra.Expression)])
tut1, tut2, tut3, tut4, tut5, tut6, tut7, tut8, tut9, tut10,
 tut11, tut12, tut13, tut14, tut15, tut16, tut17, tut18, tut19, tut20
   :: (Name, Score.T, Orchestra.T)
score1, score2, score3, score4, score5, score6, score7, score8
   :: [Score.Statement]

o1 = let i = (inst1, oe1)
     in  (hdr, [i])
```

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 = "src/Test/CSound"
-- 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 `Csound` 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 :: Int
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 = twoHarms : scored (setInstrument "inst1" tune1)

oe2 :: OrcExp
oe2 = let signal = Osc AR (dbToAmp noteVol) (pchToHz notePit)
                           (fromIntegral twoHarmsTN)
      in  StereoOut signal signal

o2 = let i = (inst1, oe2)
     in  (hdr, [i])

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 `CSound` 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 :: Int
```

```
manySinesTN = 3
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 = manySines : scored (setInstrument "inst1" tune1)

oe3 :: OrcExp
oe3 = let ampenv = Line CR 1.0 noteDur 0.0
          signal = Osc AR (ampenv * dbToAmp noteVol) (pchToHz notePit)
                          (fromIntegral manySinesTN)
      in  StereoOut signal signal

o3 = let i = (inst1, oe3)
     in  (hdr, [i])

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 :: OrcExp
oe4 = let pitch      = pchToHz notePit
          amp        = dbToAmp noteVol
          mkLine t   = LineSeg CR 0 (noteDur*t) 1 [(noteDur * (1-t), 0)]
          aenv1      = Line 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      = Line CR 0 noteDur 1
```

```
          mkOsc ae p = Osc AR (ae * amp) (pitch * p) pureToneTable
          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  StereoOut out out

o4 = let i = (inst1, oe4)
     in  (hdr, [i])

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 :: Int
coolEnvTN = 4
coolEnv :: Score.Statement
coolEnv = Score.Table coolEnvTN 0 8192 True (cubicSpline 1 [(1692, 0.2),
                                          (3000, 1), (3500, 0)])
```

Let us also add some *pfields* to the notes in our score. The two pfields 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 :: Music.T
tune2 = let attr start end = [Velocity 1.4, PFields [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 `CSound` 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 :: Int
triangleTN = 5
squareTN   = 6
sawtoothTN = 7
triangleT, squareT, sawtoothT :: Score.Statement
triangleT = triangle triangleTN
squareT   = square   squareTN
sawtoothT = sawtooth sawtoothTN

score4 = squareT : triangleT : sawtoothT : coolEnv :
                   scored (changeTempo 0.5 (setInstrument "inst1" tune2))

oe5 :: OrcExp
oe5 = let pitch  = pchToHz notePit
          amp    = dbToAmp noteVol
          pan    = Line CR p1 noteDur p2
          oscF   = 1 / noteDur
          ampen  = Osc CR amp oscF (fromIntegral coolEnvTN)
          signal = Osc AR ampen pitch (fromIntegral squareTN)
          left   = signal * pan
          right  = signal * (1-pan)
      in  StereoOut left right

o5 = let i = (inst1, oe5)
     in  (hdr, [i])

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 = manySines : pureTone : scored (setInstrument "inst1" tune1) ++
                                scored (setInstrument "inst2" tune1)

oe6 :: OrcExp
oe6 = let ampenv = Line CR 1.0 noteDur 0.0
```

```
          signal = Osc AR (ampenv * dbToAmp noteVol)
                          (pchToHz (notePit - 2)) (fromIntegral manySinesTN)
          left   = 0.8 * signal
          right  = 0.2 * signal
      in  StereoOut left right

oe7 :: OrcExp
oe7 = let pitch      = pchToHz notePit
          amp        = dbToAmp noteVol
          mkLine t   = LineSeg CR 0 (noteDur*t) 0.5 [(noteDur * (1-t), 0)]
          aenv1      = Line 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      = Line CR 0 noteDur 0.5
          mkOsc ae p = Osc AR (ae * amp) (pitch * p) pureToneTable
          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  StereoOut left right

o6 = let i1 = (inst1, oe6)
         i2 = (inst2, oe7)
     in  (hdr, [i1, i2])

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 1 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 :: OrcExp
oe8 = let modfreq = Line CR 0.0 noteDur 200.0
          modamp  = Osc AR 1.0 modfreq pureToneTable
          modsig  = (modamp + 1.0) * 0.5 * noteVol
          carrier = Osc AR modsig notePit pureToneTable
      in  StereoOut carrier carrier

o7 = let i = (inst1, oe8)
     in  (hdr, [i])

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 :: OrcExp
oe9 = let modfreq = Line CR 0.0 noteDur 200.0
          modamp  = modfreq
          modsig  = Osc AR modamp modfreq pureToneTable
          carrier = Osc AR noteVol (notePit + modsig) pureToneTable
      in  StereoOut carrier carrier

o8 = let i = (inst1, oe9)
     in  (hdr, [i])

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 :: OrcExp
oe10 = 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  = Osc AR modamp 80.0 pureToneTable
           carrier = Osc AR noteVol (notePit + modsig) pureToneTable
       in  StereoOut carrier carrier
```

```
o9 = let i = (inst1, oe10)
     in  (hdr, [i])

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 p1 in the score file. The modulation index will be a linear slope going from 0 to p2 over the duration of each note. Let us also add panning control as in additive synthesis - p3 will be the initial left channel percentage, and p4 the final left channel percentage:

```
oe11 :: OrcExp
oe11 = let carfreq = pchToHz notePit
           caramp  = dbToAmp noteVol
           modfreq = carfreq * p1
           modind  = Line CR 0 noteDur p2
           modamp  = modfreq * modind
           modsig  = Osc AR modamp modfreq pureToneTable
           carrier = Osc AR caramp (carfreq + modsig) pureToneTable
           mainamp = Osc CR 1.0 (1/noteDur) (fromIntegral coolEnvTN)
           pan     = Line CR p3 noteDur p4
           left    = mainamp * pan * carrier
           right   = mainamp * (1 - pan) * carrier
       in  StereoOut left right

o10 = let i = (inst1, oe11)
      in  (hdr, [i])
```

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 -> Music.T

quickChord :: [Music.Dur -> [Music.NoteAttribute] -> Music.T]
   -> Float -> Float -> Float -> Float -> Music.T
quickChord ns x y z w = chord $
   map (\p -> p wn [Velocity 1.4, PFields [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 :: Music.T
tune3 = 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 = pureTone : coolEnv : scored (changeTempo 0.5 (setInstrument "inst1" 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 :: OrcExp
oe12 = let car1freq = pchToHz notePit
           car2freq = pchToHz (notePit + 1)
           car1amp  = dbToAmp noteVol
           car2amp  = dbToAmp noteVol * 0.7
           modfreq  = car1freq * p1
           modind   = Line CR 0 noteDur p2
           modamp   = modfreq * modind
           modsig   = Osc AR modamp modfreq pureToneTable
           carrier1 = Osc AR car1amp (car1freq + modsig) pureToneTable
           carrier2 = Osc AR car2amp (car2freq + modsig) pureToneTable
           mainamp  = Osc CR 1.0 (1/noteDur) (fromIntegral coolEnvTN)
           pan      = Line CR p3 noteDur p4
           left     = mainamp * pan * (carrier1 + carrier2)
```

```
           right    = mainamp * (1 - pan) * (carrier1 + carrier2)
       in  StereoOut left right

o11 = let i = (inst1, oe12)
      in  (hdr, [i])

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 :: OrcExp
oe13 = let carfreq  = pchToHz notePit
           caramp   = dbToAmp noteVol
           mod1freq = carfreq * p1
           mod2freq = mod1freq * 2.0
           modind   = Line CR 0 noteDur p2
           mod1amp  = mod1freq * modind
           mod2amp  = mod1amp * 3.0
           mod1sig  = Osc AR mod1amp mod1freq pureToneTable
           mod2sig  = Osc AR mod2amp (mod2freq + mod1sig) pureToneTable
           carrier  = Osc AR caramp (carfreq + mod2sig) pureToneTable
           mainamp  = Osc CR 1.0 (1/noteDur) (fromIntegral coolEnvTN)
           pan      = Line CR p3 noteDur p4
           left     = mainamp * pan * carrier
           right    = mainamp * (1 - pan) * carrier
       in  StereoOut left right

o12 = let i = (inst1, oe13)
      in  (hdr, [i])

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 :: OrcExp
oe14 = let numharms = Line CR 70 noteDur 0
           signal   = Buzz (dbToAmp noteVol) (pchToHz notePit)
                       numharms pureToneTable
       in  StereoOut signal signal

o13 = let i = (inst1, oe14)
      in  (hdr, [i])

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 :: OrcExp
oe15 = let numharms = Line CR 0 noteDur 70
           signal   = Buzz (dbToAmp noteVol) (pchToHz notePit)
                       numharms pureToneTable
       in  StereoOut signal signal

o14 = let i = (inst1, oe15)
      in  (hdr, [i])

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 :: OrcExp
oe16 = let signal = Pluck (dbToAmp noteVol) (pchToHz notePit) 0
                          simpleSmooth 0 0
       in  StereoOut signal signal

o15 = let i = (inst1, oe16)
      in  (hdr, [i])

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 :: Int
midHumpTN = 8
midHump :: Score.Statement
midHump = Score.Table midHumpTN 0 8192 True (cubicSpline 0.0 [(4096, 1.0),
                                             (4096, 0.0)])
```

```
score8 = pureTone : midHump : scored (setInstrument "inst1" tune1)

oe17 :: OrcExp
oe17 = let string = Pluck (dbToAmp noteVol) (pchToHz notePit) 0
                          stretchSmooth 1.5 0
           drum   = Pluck 6000 (pchToHz notePit) 0 simpleDrum 0.3 0
           env    = Osc CR 1.0 (1 / noteDur) (fromIntegral midHumpTN)
           left   = (0.65 * drum) + (0.35 * env * string)
           right  = (0.35 * drum) + (0.65 * env * string)
       in  StereoOut left right

o16 = let i = (inst1, oe17)
      in  (hdr, [i])

tut16 = example "tut16" score8 o16
```

Let us now turn our attention to the effects we can achieve using a *delay line*. Let's return to a simple instrument we defined at the beginning of the tutorial - `oe3` specifes an instrument containing both harmonic and inharmonic partials, with a linearly decaying amplitude envelope. Here we take that instrument and add a little echo to it using delay:

```
oe18 :: OrcExp
oe18 = let ampenv = Line CR 1.0 noteDur 0.0
           sig    = Osc AR (ampenv * dbToAmp noteVol) (pchToHz notePit)
                           (fromIntegral manySinesTN)
           dline  = Delay 0.1 sig
           dsig1  = DelTap 0.05 dline
           dsig2  = DelTap 0.1  dline
           left   = (0.65 * sig) + (0.35 * dsig2) + (0.5 * dsig1)
           right  = (0.35 * sig) + (0.65 * dsig2) + (0.5 * dsig1)
       in  StereoOut left right

o17 = let i = (inst1, oe18)
      in  (hdr, [i])

tut17 = example "tut17" score3 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 `CSound` 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 :: OrcExp
oe19 = let ampenv = Line CR 1.0 noteDur 0.0
           sig    = Osc AR (ampenv * dbToAmp noteVol) (pchToHz notePit)
                           (fromIntegral manySinesTN)
           rev    = Reverb sig 0.15
           left   = (0.65 * sig) + (0.35 * rev)
           right  = (0.35 * sig) + (0.65 * rev)
       in  StereoOut left right

o18 = let i = (inst1, oe19)
      in  (hdr, [i])

tut18 = example "tut18" score5 o18
```

The other two reverb constructors 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 :: OrcExp
oe20 = let ampenv = Line CR 1.0 noteDur 0.0
           sig    = Osc AR (ampenv * dbToAmp noteVol) (pchToHz notePit)
                           (fromIntegral manySinesTN)
           rev    = Comb sig 1.0 0.1
       in  StereoOut rev rev

o19 = let i = (inst1, oe20)
      in  (hdr, [i])

tut19 = example "tut19" score3 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 very 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 :: OrcExp
oe21 = let carfreq = pchToHz notePit
           ampenv  = Osc CR 1.0 (1/noteDur) (fromIntegral coolEnvTN)
           caramp  = dbToAmp noteVol * ampenv
           modfreq = carfreq * p1
```

```
           modind  = Line CR 0 noteDur p2
           modamp  = modfreq * modind
           modsig  = Osc AR modamp modfreq pureToneTable
           carrier = Osc AR caramp (carfreq + modsig) pureToneTable
           dline   = Delay 1.0 carrier
           ftime   = Osc AR 0.01 2 pureToneTable
           flange  = DelTapI (0.5 + ftime) dline
           flanger = ampenv * flange
           signal  = carrier + flanger
           pan     = Line CR p3 noteDur p4
           left    = pan * signal
           right   = (1 - pan) * signal
       in  StereoOut left right

o20 = let i = (inst1, oe21)
      in  (hdr, [i])

tut20 = example "tut20" score7 o20
```

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

```
example :: Name -> Score.T -> Orchestra.T -> (Name, Score.T, Orchestra.T)
example = (,,)

test :: (Name, Score.T, Orchestra.T) -> IO ()
test = play csoundDir

play :: FilePath -> (Name, Score.T, Orchestra.T) -> IO ()
play dir (name, s, o) =
   let scorename = name ++ ".sco"
       orchname  = name ++ ".orc"
--     wavename  = name ++ ".wav"
       ((rate, _, channels), _) = o
   in  do writeFile (dir++"/"++scorename) (Score.toString s)
          writeFile (dir++"/"++orchname)  (Orchestra.toString o)
{-
          system ("cd "++dir++" ; csound32 -d -W -o "
                    ++ wavename ++ " " ++ orchname ++ " " ++ scorename
                    ++ " ; play " ++ wavename)
-}
          system ("cd "++dir++" ; csound32 -d -o stdout -s "
                    ++ orchname ++ " " ++ scorename
                    ++ " | play -c " ++ show channels ++
                              " -r " ++ show rate ++ " -t sw -")
```

```
{-
          system ("cd "++dir++" ; csound32 -d -o dac "  -- /dev/dsp makes some chac
                    ++ orchname ++ " " ++ scorename)
-}
{-
          system (dir ++ "/csound.exe -W -o " ++ wavename
                    ++ " " ++ orchname ++ " " ++ scorename)
-}
          return ()
```

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, reedy2, flute
   :: (Name, Score.T, Orchestra.T)

pianoOrc, reedyOrc, reedy2Orc, fluteOrc
   :: ((Int, Int, Int), [(Int, OrcExp)])

pianoScore, reedyScore, fluteScore :: Score.T
pianoEnv, reedyEnv, fluteEnv :: Score.Statement
pianoWave, reedyWave, fluteWave :: Score.Statement

pianoEnvTN, pianoWaveTN :: Int
pianoEnvTN  = 10
pianoWaveTN = 11
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 = pianoEnv : pianoWave : scored (setInstrument "inst1" tune1)

pianoOE :: OrcExp
pianoOE = let ampenv = Osc CR (dbToAmp noteVol) (1/noteDur)
                               (fromIntegral pianoEnvTN)
              signal = Osc AR ampenv (pchToHz notePit)
                                      (fromIntegral pianoWaveTN)
          in  StereoOut signal signal
```

```
pianoOrc = let i = (inst1, pianoOE)
           in  (hdr, [i])

piano = example "piano" pianoScore pianoOrc
```

Here is another instrument with a reedy sound to it

```
reedyEnvTN, reedyWaveTN :: Int
reedyEnvTN  = 12
reedyWaveTN = 13
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 = reedyEnv : reedyWave : scored (setInstrument "inst1" tune1)

reedyOE :: OrcExp
reedyOE = let ampenv = Osc CR (dbToAmp noteVol) (1/noteDur)
                               (fromIntegral reedyEnvTN)
              signal = Osc AR ampenv (pchToHz notePit)
                                      (fromIntegral reedyWaveTN)
          in  StereoOut signal signal

reedyOrc = let i = (inst1, reedyOE)
           in  (hdr, [i])

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 :: OrcExp
reedy2OE = let ampenv = Osc CR (dbToAmp noteVol) (1/noteDur)
                               (fromIntegral reedyEnvTN)
               freq   = pchToHz notePit
               a1     = Osc AR ampenv freq (fromIntegral reedyWaveTN)
               a2     = Osc AR (ampenv * 0.44) (freq + (0.023 * freq))
                               (fromIntegral reedyWaveTN)
               a3     = Osc AR (ampenv * 0.26) (freq + (0.019 * freq))
                               (fromIntegral reedyWaveTN)
               left   = (a1 * 0.5) + (a2 * 0.35) + (a3 * 0.65)
               right  = (a1 * 0.5) + (a2 * 0.65) + (a3 * 0.35)
           in  StereoOut left right

reedy2Orc = let i = (inst1, reedy2OE)
```

```
            in  (hdr, [i])

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 constructor, which generates a signal that interpolates between 2 random numbers over a certain number of samples that is specified by teh second parameter.

```
fluteTune :: Music.T

fluteTune = Music.line
               (map (flip id [Velocity 1.6, PFields [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, fluteWaveTN :: Int
fluteEnvTN  = 14
fluteWaveTN = 15
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 = fluteEnv : fluteWave : scored (setInstrument "inst1" fluteTune)

fluteOE :: OrcExp
fluteOE = let vol    = dbToAmp noteVol
              rand   = RandomI AR ((vol/100) * p1) p2
              ampenv = OscI AR (rand + vol) (1 / noteDur)
                               (fromIntegral fluteEnvTN)
              signal = OscI AR ampenv (pchToHz notePit)
                                      (fromIntegral fluteWaveTN)
          in  StereoOut signal signal

fluteOrc = let i = (inst1, fluteOE)
           in  (hdr, [i])

flute = example "flute" fluteScore fluteOrc
```

6.3 MML

```
module Haskore.Interface.MML where
```

```
import Data.Ratio((%))
import qualified Data.List as List
import Control.Monad.State

import qualified Haskore.Basic.Pitch as Pitch
import qualified Haskore.Music as Music
```

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 -> ([Music.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 (Music.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 c

toMusicState :: String -> State Accum [Music.T]
toMusicState s = State (barToMusic s)

toMusic :: Pitch.Octave -> String -> Music.T
toMusic oct s = Music.line (evalState (toMusicState s) (0, oct))
```

7 Processing and Analysis

7.1 Optimization

This module provides functions that simplify the structure of a `Music.T` according to the rules proven in Section 4.1

```
module Haskore.Process.Optimization where

import qualified Media
import qualified Media.Temporal
import qualified Haskore.Music as Music
import Data.Maybe (catMaybes, fromMaybe, mapMaybe)
```

`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, duration, tempo, transpose, instrument, volume ::
   Music.T -> Music.T
all = tempo . transpose . volume . instrument . singleton . rest
```

Remove rests of zero duration.

```
rest = Music.mapList
   (curry id)
   (curry id)
   (filter (/= Music.rest 0))
   (filter (/= Music.rest 0))
```

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 -> if d == 0
          then const Nothing
          else Just . Music.atom d)
   (fmap . Music.control)
   (Just . Media.serial   . catMaybes)
   (Just . Media.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 they are merged to 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 -> Music.T) -> (a -> a -> a) -> Music.T -> Music.T
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.foldListFlat (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.

```
mergeControl :: Eq a => (Music.Control -> Maybe a)
   -> (a -> Music.T -> Music.T) -> (a -> a -> a) -> Music.T -> Music.T
mergeControl extract control merge =
   let collectControl =
```

```
         Music.foldListFlat
            (\d n -> ([], Music.atom d n))
            (\c m -> let cm = collectControl m
                     in  (c : fst cm, snd cm))
            ((,) [] . Music.line  . map recurse)
            ((,) [] . Music.chord . map recurse)
      recurse m =
         let cm = collectControl m
             xs = mapMaybe extract (fst cm)
             x  = foldl1 merge xs
             cs' = filter ((==) Nothing . extract) (fst cm)
             collectedCtrl = if null xs then id else control x
         in  collectedCtrl (foldr id (snd cm) (map Music.control cs'))
   in  recurse
```

The function `removeNeutral` removes controllers that have no effect.

```
removeNeutral :: (Music.Control -> Bool) -> Music.T -> Music.T
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 (+)
```

Remove unnecessary `Instrument`s. A more sophisticated algorithm could find more Instrument change redundancies, but since it is planned to turn instruments into a note attribute (rather than a global control) this is not so important for now.

```
instrument =
   let maybeInstrument (Music.Instrument t) = Just t
       maybeInstrument _                    = Nothing
   in  mergeControl maybeInstrument Music.setInstrument (flip const)
```

Change repeated Volume Note Attributes to Phrase Attributes.

```
volume = id

{-
   Is it really necessary to have different possibilities for setting the volume?
   It is wrong to merge (Phrase p1 (Phrase p2 m)) to (Phrase p1 m) if p1 == p2
   because the Loudness should multiply rather than replace the old one.
volume m1 = let (m2,changed) = volume' m1
            in if changed then volume m2
                          else m2
volume' ((m1 +:+ m2) +:+ m3) = (m1 +:+ m2 +:+ m3, True)
volume' ((m1 =:= m2) =:= m3) = (m1 =:= m2 =:= m3, True)
volume' (Phrase p1 (Phrase p2 m)) | p1 == p2  = (Phrase p1 m, True)
                                  | otherwise = (Phrase (p1 ++ p2) m, True)
volume' (Note p1 d1 [Volume v1] +:+ Note p2 d2 [Volume v2]) | v1 == v2 =
    (Phrase [Dyn (Loudness v1)] (Note p1 d1 [] +:+ Note p2 d2 []), True)
volume' (Note p1 d1 [Volume v1] =:= Note p2 d2 [Volume v2]) | v1 == v2 =
    (Phrase [Dyn (Loudness v1)] (Note p1 d1 [] =:= Note p2 d2 []), True)
volume' (Note p1 d1 [Volume v1] +:+ Note p2 d2 [Volume v2] +:+ x) | v1 == v2 =
   ((Phrase [Dyn (Loudness v1)] (Note p1 d1 [] +:+ Note p2 d2 [])) +:+ x, True)
volume' (Note p1 d1 [Volume v1] =:= Note p2 d2 [Volume v2] =:= x) | v1 == v2 =
   ((Phrase [Dyn (Loudness v1)] (Note p1 d1 [] =:= Note p2 d2 [])) =:= x, True)
volume' (Phrase p1 m1 +:+ Phrase p2 m2) | p1 == p2 = (Phrase p1 (m1 +:+ m2), True)
volume' (Phrase p1 m1 =:= Phrase p2 m2) | p1 == p2 = (Phrase p1 (m1 =:= m2), True)
volume' (Phrase p1 m1 +:+ Phrase p2 m2 +:+ x) | p1 == p2 =
    (Phrase p1 (m1 +:+ m2 +:+ x),True)
volume' (Phrase p1 m1 =:= Phrase p2 m2 =:= x) | p1 == p2 =
    (Phrase p1 (m1 =:= m2 =:= x),True)
volume' (Phrase phr@[Dyn (Loudness l)] m +:+ Note p d [Volume v]) | l == v =
    (Phrase phr (m +:+ Note p d []), True)
volume' (Phrase phr@[Dyn (Loudness l)] m =:= Note p d [Volume v]) | l == v =
    (Phrase phr (m =:= Note p d []), True)
volume' (Phrase phr@[Dyn (Loudness l)] m +:+ Note p d [Volume v] +:+ x) | l == v =
    (Phrase phr (m +:+ Note p d []) +:+ x,True)
volume' (Phrase phr@[Dyn (Loudness l)] m =:= Note p d [Volume v] =:= x) | l == v =
    (Phrase phr (m =:= Note p d []) =:= x,True)
volume' (Note p d [Volume v] +:+ Phrase phr@[Dyn (Loudness l)] m) | l == v =
    (Phrase phr (Note p d [] +:+ m),True)
volume' (Note p d [Volume v] =:= Phrase phr@[Dyn (Loudness l)] m) | l == v =
    (Phrase phr (Note p d [] =:= m),True)
volume' (Note p d [Volume v] +:+ Phrase phr@[Dyn (Loudness l)] m +:+ x) | l == v =
    ((Phrase phr (Note p d [] +:+ m)) +:+ x, True)
volume' (Note p d [Volume v] =:= Phrase phr@[Dyn (Loudness l)] m =:= x) | l == v =
    ((Phrase phr (Note p d [] =:= m)) =:= x, True)
volume' (Rest r +:+ Note p d [Volume v]) =
```

```
    (Phrase [Dyn (Loudness v)] (Rest r +:+ Note p d []),True)
volume' (Rest r =:= Note p d [Volume v]) =
    (Phrase [Dyn (Loudness v)] (Rest r =:= Note p d []),True)
volume' (Note p d [Volume v] +:+ Rest r) =
    (Phrase [Dyn (Loudness v)] (Note p d [] +:+ Rest r),True)
volume' (Note p d [Volume v] =:= Rest r) =
    (Phrase [Dyn (Loudness v)] (Note p d [] =:= Rest r),True)
volume' (Rest r +:+ Phrase p m) = (Phrase p (Rest r +:+ m), True)
volume' (Rest r =:= Phrase p m) = (Phrase p (Rest r =:= m), True)
volume' (Phrase p m +:+ Rest r) = (Phrase p (m +:+ Rest r), True)
volume' (Phrase p m =:= Rest r) = (Phrase p (m =:= Rest r), True)
volume' (m1 +:+ m2) = let (m3,c3) = volume' m1
                          (m4,c4) = volume' m2
                      in (m3 +:+ m4, c3 || c4)
volume' (m1 =:= m2) = let (m3,c3) = volume' m1
                          (m4,c4) = volume' m2
                      in (m3 =:= m4, c3 || c4)
volume' (Tempo a m)   = (Tempo a   $ volume m, False)
volume' (Transpose a m)   = (Transpose a   $ volume m, False)
volume' (Instrument a m)   = (Instrument a   $ volume m, False)
volume' (Player a m)  = (Player a  $ volume m, False)
volume' (Phrase a m)  = (Phrase a  $ volume m, False)
volume' x             = (x, False)
-}
```

Eliminate `Serial` and `Parallel` composition if they contain only one member. This can be done very general for `Media.T`, but in this case it doesn't descend into `Control`. Thus we have also a version which works satisfyingly on `Music.T`.

```
singletonMedia :: (Media.Temporal.Class a, Media.Class media) =>
   media a -> media a
singletonMedia =
   Media.foldList Media.prim
      (\ms -> case ms of {[x] -> x; _ -> Media.serial   ms})
      (\ms -> case ms of {[x] -> x; _ -> Media.parallel ms})

singleton :: Music.T -> Music.T
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})
```

7.2 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 `Media` 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 Media.ContextFreeGrammar where

import Data.List (sort, tails, isPrefixOf, maximumBy, findIndex)
import Data.Maybe (fromJust)
import Data.FiniteMap

import Control.Monad.State

import Media (prim, serial1, parallel1)
import qualified Media
import qualified Media.List
```

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.

```
data Tag key prim =
     Prim prim
   | Call key
   | CallMulti Int key
   deriving (Eq, Ord, Show)
type TagMedia key prim = Media.List.T (Tag key prim)

-- True is for cyclic infixes
type T key prim = [(key, TagMedia key prim)]

fromMedia :: (Ord key, Ord prim) =>
   [key] -> Int -> Media.List.T prim -> T key prim
fromMedia (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)]
fromMedia _ _ _ =
   error ("No key given."++
      " Please provide an infinite or at least huge number of macro names.")
```

The inverse of `fromMedia`: 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 recurse 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 Media. 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 Media. 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 `toMedia` computes the expansion of the first macro. Thus `toMedia` quite inverts `fromMedia`.

```
toMedia :: (Show key, Ord key, Ord prim) =>
   T key prim -> Media.List.T prim
toMedia = snd . head . expand

expand :: (Show key, Ord key, Ord prim) =>
   T key prim -> [(key, Media.List.T prim)]
expand grammar =
   let notFound key = error ("The non-terminal '" ++ show key ++ "' is unknown.")
       dict  = listToFM grammar
       dict' = mapFM (\_ -> Media.foldList expandSub serial1 parallel1) dict
       expandSub (Prim p) = prim p
       expandSub (Call key) =
          lookupWithDefaultFM dict' (notFound key) key
       expandSub (CallMulti n key) =
          serial1 (replicate n (lookupWithDefaultFM dict' (notFound key) key))
   in  map (fromJust . lookupFM (mapFM (curry id) dict') . fst) grammar
```

Do monadic actions until the condition `p` fails. This is implemented for State Monads, because in plain Monads one could not reset the state and thus the state wouldn't be that after the last successful (with respect to the predicate `p`) action.

```
whileM :: (MonadState s m) => (a -> Bool) -> [m a] -> m [a]
whileM _ [] = return []
whileM p (m:ms) =
   do s <- get
      x <- m
      if p x then whileM p ms >>= return . (x:)
             else put s -- reset to the old state
                    >> return []
```

Find the longest common infix over all parts of the music and replace it in all of them.

```
condense :: (Ord key, Ord prim) =>
      key
   -> T key prim
   -> (Int, T key prim)
```

```
condense key x =
   let getSerials = Media.foldListFlat
          (const [])
          (\xs -> xs : concatMap getSerials xs)
          (\xs ->      concatMap getSerials xs)
       infx = smallestCycle (maximumCommonInfixMulti length
                 (concatMap (getSerials . snd) x))
       absorbSingleton _ [m] = m
       absorbSingleton collect ms = collect ms
       replaceRec = Media.foldList prim
          (absorbSingleton serial1 . map joinTag . replaceInfix key infx)
          (absorbSingleton parallel1)
   in  (length infx, (key, serial1 infx) : map (\(k, ms) -> (k, replaceRec ms)) x)

joinTag :: Tag key (TagMedia key prim) -> TagMedia key prim
joinTag (Prim m)        = m
joinTag (Call k)        = prim (Call k)
joinTag (CallMulti n k) = prim (CallMulti n k)
```

Replace all occurences of the infix by its key. Collect accumulated occurences in one `CallMulti`.

```
replaceInfix :: (Eq a, Eq b) =>
      a
   -> [b]
   -> [b]
   -> [Tag a b]
replaceInfix key infx sequ =
   let recurse [] = []
       recurse xa@(x:xs) =
          let pref = commonPrefix (cycle infx) xa
              (num, r) = divMod (length pref) (length infx)
              len = length pref - r
          in  if num == 0
              then Prim x : recurse xs
              else ((if num == 1 then Call key else CallMulti num key)
                      : recurse (drop len xa))
   in  recurse sequ
```

A common infix indicates a loop if its occurences overlap. We can detect this by checking if there is a suffix of our list which is also a prefix of this list.

```
isCyclic :: Eq a => [a] -> Bool
isCyclic x = any (flip isPrefixOf x) (init (tail (tails x)))
```

Find the shortest list `y`, where `x` is a prefix of `cycle y`. If `x` has no loop, then `x == y`.

```
smallestCycle :: Eq a => [a] -> [a]
```

```
smallestCycle x =
   take (1 + fromJust (findIndex (flip isPrefixOf x) (tail (tails x)))) x
```

Finding common infixes is a prominent application of suffix trees. But since I don't have an implementation of suffix trees I'll stick to a sorted list of suffices.

```
maximumCommonInfix :: (Ord a, Ord b) => ([a] -> b) -> [a] -> [a]
maximumCommonInfix mag x =
   let xSort = sort (tails x)
       commonInfixes = zipWith commonPrefix xSort (tail xSort)
   in  maximumByMag mag commonInfixes
```

Find common infixes across multiple strings. This would be an application for generalized suffix trees.

```
maximumCommonInfixMulti :: (Ord a, Ord b) => ([a] -> b) -> [[a]] -> [a]
maximumCommonInfixMulti mag x =
   let xSort = sort (concatMap tails x)
       commonInfixes = zipWith commonPrefix xSort (tail xSort)
   in  maximumByMag mag commonInfixes
```

Find the maximum element with respect to the magnitude function 'mag'.

```
maximumByMag :: Ord b => (a -> b) -> [a] -> a
maximumByMag mag = maximumBy (\x y -> compare (mag x) (mag y))
```

Find the longest common prefix.

```
commonPrefix :: Eq a => [a] -> [a] -> [a]
commonPrefix (x:xs) (y:ys) =
   if x == y then x : commonPrefix xs ys
             else []
commonPrefix _ _ = []
```

7.3 Markov Chains

Markov chains can be used to recompose a list of elements respecting the fact that the probability of a certain element depends on preceding elements in the list. We will use this for recomposition of music streams as demonstrated in module `Kantate147`.

```
module Haskore.General.MarkovChain where

import Data.FiniteMap (FiniteMap, lookupWithDefaultFM, addListToFM_C, emptyFM)
import System.Random (RandomGen, randomR)
import Control.Monad.State
```

Creates a chain of characters according to the probabilities of possible successor

```
walk :: (Ord a, RandomGen g) => Int -> [a] -> Int -> g -> [a]
walk n     -- size of look-ahead buffer
     dict  -- text to walk through randomly
     start -- index to start the random walk within 'dict'
     g     -- random generator state
   = let fm = createMap n dict
         {- This is the main function of this program.
            It is quite involved.
            If you want to understand it,
            imagine that the list 'y' completely exists
            before the computation. -}
         y = take n (drop start dict) ++
               -- run them on the initial random generator state
               evalState
                 -- this turns the list of possible successors
                 -- into an action that generate a list
                 -- of randomly chosen items
                 (mapM randomItem
                   -- lookup all possible successors of each infix
                   (map (lookupWithDefaultFM fm
                           (error ("key is not contained in dictionary")))
                     -- list all infixes of length n
                     (map (take n) (iterate tail y)))) g
     in  y
```

chose a random item from a list

```
randomItem :: (RandomGen g) => [a] -> State g a
randomItem x = fmap (x!!) (State (randomR (0, length x - 1)))
```

create a map that lists for each string all possible successors

```
createMap :: (Ord a) => Int -> [a] -> FiniteMap [a] [a]
createMap n x =
   let xc = cycle x
       -- list of the map keys
       sufxs   = map (take n) (iterate tail xc)
       -- list of the map images, i.e. single element lists
       imgxs   = map (:[]) (drop n xc)
       mapList = take (length x) (zip sufxs imgxs)
   in  addListToFM_C (++) emptyFM mapList
```

7.4 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.Music.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 Media
import qualified Haskore.Music as Music
import Media.ContextFreeGrammar as Grammar
import Data.FiniteMap (FiniteMap, listToFM, lookupFM)
import qualified Data.Char as Char
import Data.List(intersperse)
```

Format a grammar as computed with the module `Media.ContextFreeGrammar`.

```
prettyGrammarMedia :: (Show prim) => Grammar.T String prim -> String
prettyGrammarMedia = prettyGrammar prim

prettyGrammarMusic :: Grammar.T String Music.Primitive -> String
prettyGrammarMusic = prettyGrammar primMusic

prettyGrammar :: (Int -> prim -> ShowS) -> Grammar.T String prim -> String
prettyGrammar primSyntax g =
   let text = unlines (map (flip id "" . bind primSyntax) g)
       Parser.ParseOk (Syntax.HsModule _ _ _ _ code) =
          Parser.parseModule text
   in  unlines (map Pretty.prettyPrint code) -- show code
```

Format a `Media` object that contains references to other media objects.

```
bind :: (Int -> prim -> ShowS) -> (String, Grammar.TagMedia String prim) -> ShowS
bind primSyntax (key, ms) =
   showString key . showString " = " . tagMedia 0 primSyntax ms

tagMedia :: Int -> (Int -> prim -> ShowS) -> Grammar.TagMedia String prim -> ShowS
tagMedia prec 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  Media.foldList (flip primSyntax')
          (listFunc "serial")
          (listFunc "parallel") 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.

```
prettyMusic :: Music.T -> String
prettyMusic m = prettyExp (music 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.

```
music :: Int -> Music.T -> ShowS
music prec m =
   Media.foldList
      (flip primMusic)
      (listFunc "line")
      (listFunc "chord") m prec

primMusic :: Int -> Music.Primitive -> ShowS
primMusic prec (Music.Atom d at)      = atom prec d at
primMusic prec (Music.Control ctrl m) = control prec ctrl m

atom :: Int -> Music.Dur -> Music.Atom -> ShowS
atom prec d (Music.Note (o,pc) nas) =
   enclose prec 10 (showString (map Char.toLower (show pc)) .
      showString " " . showsPrec 10 o . showString " " . durSyntax id "n" d .
      showString " " . showsPrec 10 nas)
atom prec d Music.Rest =
```

```
   durSyntax (\dStr -> enclose prec 10 (showString "rest " . dStr)) "nr" d

control :: Int -> Music.Control -> Music.T -> ShowS
control prec c m =
   let controlSyntax name arg =
          enclose prec 10
             (showString name . showString " " . arg . showString " " . music 10 m)
   in  case c of
          Music.Tempo d      -> controlSyntax "changeTempo"   (showsPrec 10 d)
          Music.Transpose p  -> controlSyntax "transpose"     (showsPrec 10 p)
          Music.Instrument i -> controlSyntax "setInstrument" (showsPrec 10 i)
          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 -> Music.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
```

```
durDict :: FiniteMap Music.Dur String
durDict =
   let durs    = zip (iterate (/2) 2)
                     ["b", "w", "h", "q", "e", "s", "t", "sf"]
       ddurs   = map (\(d,s) -> (3/2*d, "d" ++s)) durs
       dddurs  = map (\(d,s) -> (7/4*d, "dd"++s)) durs
   in  listToFM (durs ++ ddurs ++ dddurs)

durSyntax :: (ShowS -> ShowS) -> String -> Music.Dur -> ShowS
durSyntax showRatio suffix d =
   maybe (showRatio (showsPrec 10 d))
         (\s -> showString (s++suffix)) (lookupFM durDict d)
```

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 expr =
   if outerPrec >= innerPrec
   then showString "(" . expr . showString ")"
   else expr
```

8 Related and Future Research

Many proposals have been put forth for programming languages targeted for computer music composition [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**], 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 [**?**], 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 [**?**], 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 [**?**]. 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 [**?**]. 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 [**?**].

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 [**?**]).

A Convenient Functions for Getting Started With Haskore

```
module Haskore.Interface.MIDI.Render where

import System( system )

import qualified Haskore.Music as Music
import qualified Haskore.Music.Performance as Performance
import qualified Haskore.Music.PerformanceContext as Context
```

```
import qualified Haskore.Music.Player as Player

import qualified Haskore.Interface.MIDI.File         as MidiFile
import qualified Haskore.Interface.MIDI.UserPatchMap as UserPatchMap
import qualified Haskore.Interface.MIDI.Write        as WriteMidi
import qualified Haskore.Interface.MIDI.Save         as SaveMidi
import qualified Haskore.Interface.MIDI.General      as GeneralMidi
```

Given a `Player.Map`, `Context.T`, `UserPatchMap.T`, and file name, we can write a `Music.T` value into a midi file:

```
musicToFile :: FilePath ->
   (UserPatchMap.T, Context.T, Music.T) -> IO ()
musicToFile fn m =
   SaveMidi.toFile fn (WriteMidi.fromMusic m)
```

A.1 Test routines

Using the defaults above, from a `Music.T` object, we can:

1. generate a `Performance.T`

   ```
   testPerform  :: Music.T -> Performance.T
   testPerform  = Performance.fromMusic  Player.defltMap Context.deflt
   testPerform' :: Music.T -> Performance.T'
   testPerform' = Performance.fromMusic' Player.defltMap Context.deflt
   ```

2. generate a `MidiFile.T` data structure

   ```
   testMidi :: Music.T -> MidiFile.T
   testMidi = WriteMidi.fromPerformance UserPatchMap.deflt . testPerform

   testGeneralMidi :: Music.T -> MidiFile.T
   testGeneralMidi = WriteMidi.fromPerformanceGM . testPerform

   testMixedMidi :: Music.T -> MidiFile.T
   testMixedMidi = WriteMidi.fromPerformanceMixed UserPatchMap.deflt . testPerform

   testMixedGeneralMidi :: Music.T -> MidiFile.T
   testMixedGeneralMidi = WriteMidi.fromPerformanceMixedGM . testPerform
   ```

3. generate a MIDI file

   ```
   test     :: Music.T -> IO ()
   test     = SaveMidi.toFile "test.mid" . testMidi
   ```

4. generate and play a MIDI file on Windows 95, Windows NT, or Linux

```
testPlay :: String -> Music.T -> IO ()
testPlay cmd m = do
                   test m
                   system cmd
                   return ()

testWin95, testNT, testLinux, testTimidity :: Music.T -> IO ()
testWin95    = testPlay "mplayer test.mid"
testNT       = testPlay "mplay32 test.mid"
testLinux    = testPlay "playmidi -rf test.mid"
testTimidity = testPlay "timidity -B8,9 test.mid"
```

Alternatively, just run `test m` manually, and then invoke the midi player on your system using `play`, defined below for NT:

```
play :: IO ()
play = do
         system "mplay32 test.mid"
         return ()
```

A.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 :: UserPatchMap.T
gmUpm = map (\(gmn, n) -> (gmn, (mod n 16 + 1, n))) GeneralMidi.map
```

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.map)
       mu  = Music.line (map (simple . fst) gMM)
       simple inm = Music.setInstrument inm Music.cMajArp
   in  musicToFile "test.mid" (gmUpm, Context.deflt, mu)
```

B Examples

B.1 Haskore in Action

```
module Haskore.Example.Miscellaneous where

import IO
import Data.Ratio ((%))
import Haskore.General.Utility (fst3, snd3, thd3)

import qualified Haskore.Basic.Pitch as Pitch
import           Haskore.Basic.Trill as Trill
import           Haskore.Basic.Drum as Drum

import           Haskore.Music as Music
import qualified Haskore.Music.PerformanceContext as Context
import qualified Haskore.Interface.MIDI.File         as MidiFile
import qualified Haskore.Interface.MIDI.UserPatchMap as UserPatchMap
import qualified Haskore.Interface.MIDI.Write        as WriteMidi
import qualified Haskore.Interface.MIDI.Save         as SaveMidi
import qualified Haskore.Interface.MIDI.Read         as ReadMidi
import qualified Haskore.Interface.MIDI.Load         as LoadMidi
import           Haskore.Interface.MIDI.Render (testMidi, testGeneralMidi)

import qualified Haskore.Example.SelfSim        as SelfSim
import qualified Haskore.Example.ChildSong6     as ChildSong6
import qualified Haskore.Example.Ssf            as Ssf

t0, t1, t2, t3, t4, t5,
 t10s, t12, t12a, t13, t13a, t13b, t13c, t13d, t13e,
 t14, t14b, t14c, t14d, cs6, ssf0 :: MidiFile.T
```

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 = testMidi (setInstrument "piano" ChildSong6.song)
```

C Major scale for use in examples below:

```
cms', cms :: Music.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'
```

```
st of various articulations and dynamics:
```

```
t1 = testGeneralMidi (setInstrument "percussion"
       (staccato 0.1 cms +:+
        cms              +:+
        legato   1.1 cms    ))

temp, mu2 :: Music.T
temp = setInstrument "piano" (crescendo 4.0 (c 0 en []))

mu2 = setInstrument "vibes"
       (diminuendo 0.75 cms +:+
        crescendo 3.0 (loudness 0.25 cms))
t2 = testMidi mu2

t3 = testMidi (setInstrument "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 -> Music.T) -> (Music.T -> Music.T) -> Int -> Music.T -> Music.T
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 :: Music.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' :: Music.T -> MidiFile.T
t4' x     = testMidi (setInstrument "piano" x)
t4        = testMidi (setInstrument "piano"
              (cascades +:+ Music.reverse cascades))
```

What happens if we simply reverse the `f` and `g` arguments?

```
run', cascade', cascades' :: Music.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        = testMidi (setInstrument "piano" cascades')
```

Example from the SelfSim module.

```
t10s   = testMidi (rep (delay SelfSim.durss) (transpose 4) 2 SelfSim.ss)
```

Example from the ChildSong6 module.

```
cs6 = testMidi ChildSong6.song
```

Example from the Ssf (Stars and Stripes Forever) module.

```
ssf0 = testMidi Ssf.song
```

Midi percussion test. Plays all ”notes” in a range. (Requires adding an instrument for percussion to the `UserPatchMap`.)

```
drums :: Pitch.Absolute -> Pitch.Absolute -> Music.T
drums dr0 dr1 = setInstrument "drums"
                  (line (map (\p -> note (Pitch.fromInt p) sn []) [dr0..dr1]))

t11 :: Pitch.Absolute -> Pitch.Absolute -> MidiFile.T
t11 dr0 dr1 = testMidi (drums dr0 dr1)
```

Test of `Music.take` and shorten.

```
t12 = testMidi (Music.take 4 ChildSong6.song)
t12a = testMidi (cms /=: ChildSong6.song)
```

Tests of the trill functions.

```
t13note :: Music.T
t13note = c 1 qn []
t13 =  testMidi (trill   1 sn t13note)
t13a = testMidi (trill’  2 dqn t13note)
t13b = testMidi (trilln  1 5 t13note)
t13c = testMidi (trilln’ 3 7 t13note)
t13d = testMidi (roll tn t13note)
t13e = testMidi (changeTempo (2/3) (transpose 2 (setInstrument "piano" (trilln’ 2 7
```

Tests of drum.

```
t14 = testMidi (setInstrument "Drum" (Drum.toMusic AcousticSnare qn []))
```

A ”funk groove”

```
t14b = let p1 = Drum.toMusic LowTom        qn []
           p2 = Drum.toMusic AcousticSnare en []
       in testMidi (changeTempo 3 (setInstrument "Drum" (line (replicate 4
                  (line [p1, qnr, p2,  qnr, p2,
                         p1, p1,  qnr, p2,  enr]
                   =:= roll en (Drum.toMusic ClosedHiHat 2 ])))))))
```

A ”jazz groove”

```
t14c = let p1 = Drum.toMusic CrashCymbal2  qn []
           p2 = Drum.toMusic AcousticSnare en []
           p3 = Drum.toMusic LowTom       qn []
       in testMidi (changeTempo 3 (setInstrument "Drum" (line (replicate 8
                ((p1 +:+ changeTempo (3%2) (p2 +:+ enr +:+ p2))
                 =:= (p3 +:+ qnr)) ))))

t14d = let p1 = Drum.toMusic LowTom       en []
           p2 = Drum.toMusic AcousticSnare hn []
       in testMidi (setInstrument "Drum"
                  (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. `Music.T` into a MIDI file.

```
tab :: Music.T -> IO ()
tab m = SaveMidi.toFile "test.mid" $
           WriteMidi.fromMusic (UserPatchMap.deflt, Context.deflt, m)
```

`Music.T` to a MidiFile datatype and back to Music.

```
tad :: Music.T -> (UserPatchMap.T, Context.T, Music.T)
tad m = ReadMidi.toMusic (testMidi m)
```

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 `Music.T`, a `UserPatchMap`, and a `Context`.

```
tcd file = do
             x <- LoadMidi.fromFile file
             print $ fst3 $ ReadMidi.toMusic x
```

```
            print $ snd3 $ ReadMidi.toMusic x
            print $ thd3 $ ReadMidi.toMusic x
```

A MIDI file to `Music.T` and back to a MIDI file.

```
tcdab file =
   LoadMidi.fromFile file >>=
      (SaveMidi.toFile "test.mid" . WriteMidi.fromMusic . ReadMidi.toMusic)
```

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.Music as Music hiding (dur)
```

note updaters for mappings

```
fdb, fd :: t -> (t -> [Music.NoteAttribute] -> m) -> m
fdb dur n  = n dur [Velocity (10/13)]
fd  dur n  = n dur v

vel :: ([Music.NoteAttribute] -> m) -> m
vel  n   = n   v

v :: [Music.NoteAttribute]
v        = []

lmap :: (a -> Music.T) -> [a] -> Music.T
lmap func l = line (map func l)


bassLine, mainVoice, song :: Music.T
```

Baseline:

```
b1, b2, b3 :: Music.T
b1 = lmap (fdb dqn) [b  3, fs 4, g  4, fs 4]
b2 = lmap (fdb dqn) [b  3, es 4, fs 4, es 4]
b3 = lmap (fdb dqn) [as 3, fs 4, g  4, fs 4]

bassLine = line $ concat [replicate 3 b1, replicate 2 b2,
                          replicate 4 b3, replicate 5 b1]
```

Main Voice:

```
v1, v1a, v1b :: Music.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 :: Music.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 = line (replicate 3 v1 ++ [v2])
```

Putting it all together:

```
song = setInstrument "piano" (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           Haskore.Music as Music hiding (a, d)
import           Haskore.Interface.MIDI.Render (testMidi)
import qualified Haskore.Interface.MIDI.File         as MidiFile

 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) => a -> Pat -> [SNote]
simFringe n pat = fringe n (Cl [(0,0)] (sim pat))

fringe :: (Num 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)]] -> Music.T
simToHask s = let mkNote (p,d) = note (Pitch.fromInt p) d []
              in line (map (chord . map mkNote) s)

-- and here are some examples of it being applied:

sim1, sim2, sim12, sim3, sim4, sim4s :: Int -> Music.T
t6, t7, t8, t9, t10 :: MidiFile.T

sim1 n = setInstrument "bass"
           (transpose (-12)
              (changeTempo 4 (simToHask (simFringe n pat1))))
t6 = testMidi (sim1 4)

sim2 n = setInstrument "piano"
           (transpose 5
              (changeTempo 4 (simToHask (simFringe n pat2))))
t7 = testMidi (sim2 4)

sim12 n = sim1 n =:= sim2 n
t8 = testMidi (sim12 4)

sim3 n = setInstrument "vibes"
           (transpose 0
              (changeTempo 4 (simToHask (simFringe n pat3))))
t9 = testMidi (sim3 3)

sim4 n  = (transpose 12
              (changeTempo 2 (simToHask (simFringe n pat4'))))

sim4s n = let s = sim4 n
```

```
              l1 = setInstrument "flute" s
              l2 = setInstrument "bass"  (transpose (-36) (Music.reverse s))
          in l1 =:= l2

ss :: Music.T
ss     = sim4s 3
durss :: Music.Dur
durss  = dur ss

t10    = testMidi 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)]]
```

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 `C`. Btw. is there a better name? `C` could also be used for a type class if this is the main object described in a module. A function in a module don't need a prefix related to the principal type.

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.

`Media.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 `Media.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. `Media.Binary` uses binary constructors `:+:`, `:=:`
2. `Media.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 `Media.T`. Import your prefered module to `Media`.

Each of these data structures has its advantages:

`Media.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.

`Media.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 identically 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 `Media.T` almost forms an algebraic ring where `=:=` is like a sum (commutative) and `+:+` is like a product (non-commutative). Unfortunately `Media.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 `Media.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.

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. The problem is that the range of notes of the enumeration `PitchClass` overlaps with notes from neighboured octaves. 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`

Overlapping `PitchClass`es, 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 is (8,A) = 440 Hz. 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 should be 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 Changing an instrument by surrounding a piece of music with an `Instr` 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 only once and that setting an instrument has an effect, we should distinguish between (instrument-less) melodies and music (with instrument information). 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 Each of the currently implemented instances of `PhraseFun` could be implemented as well with essentially the type `(T, Dur) -> (T, Dur)` instead of `Music.T -> (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.

Phrase Instead of a list of `PhraseAttribute`s the constructor `Phrase` allows only one attribute in order to make the order application transparent to the user.

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.