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)

1 Introduction

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

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

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

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

Building on the results of the functional programming community's Haskell effort has several important advantages: First, and most obvious, we can avoid the difficulties involved in new programming language design, and at the same time take advantage of the many years

of effort that went into the design of Haskell. Second, the resulting system is both *extensible* (the user is free to add new features in substantive, creative ways) and *modifiable* (if the user doesn't like our approach to a particular musical idea, she is free to change it).

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

1.1 Acknowledgements

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

2 The Architecture of Haskore

Figure 1 shows the overall structure of Haskore. Note the independence of high level structures from the "music platform" on which Haskore runs. Originally, the goal was for Haskore compositions to run equally well as conventional midi-files [IMA90], NeXT MusicKit score files [JB91], and csound score files [Ver86], and for Haskore compositions to be displayed and printed in traditional notation using the CMN (Common Music Notation) subsystem. In reality, only one of these platforms is currently supported: midi. It is probably the most popular platform for users, and its use with Haskore is described in detail in this tutorial.[1]

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 HaskoreLoader (module HaskoreLoader, module Basics,
>                       module Performance,
>                       -- module Players,
>                       module HaskToMidi, module TestHaskore, module ReadMidi)
>        where
>
```

[1] (1) The NeXT music platform is obsolete. (2) There exists a translation to csound for an earlier version of Haskore that I would happily share with anyone interested. (3) We have abandoned CMN entirely, as there are now better candidates for notation packages into which Haskore could be mapped.

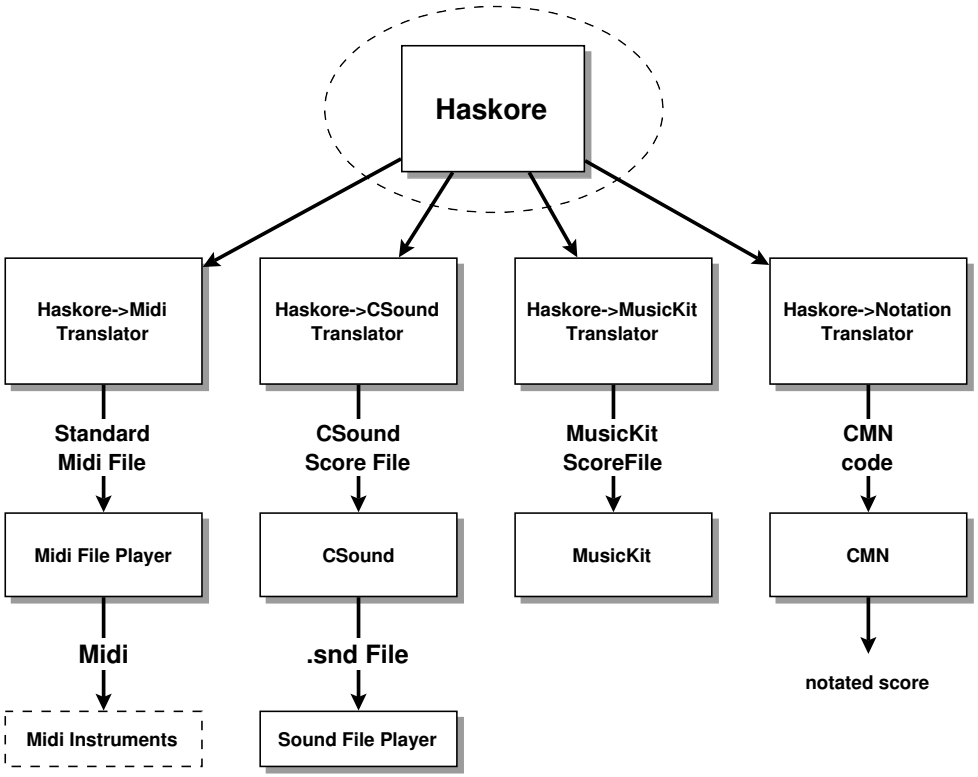


Figure 1: Overall System Diagram

```
> import Basics         -- described in Section 3
> import Performance    -- described in Section 4
> -- import Players     -- described in Section 5
> import HaskToMidi     -- described in Section 6
> import TestHaskore
> import ReadMidi
```

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 `Players` in this document are commented out, which in effect makes it part of module `Performance` (with which it is mutually recursive).

A final word before beginning: As various musical ideas are presented in this Haskore tutorial, I urge the reader to question the design decisions that are made. There is no supreme theory of music that dictates my decisions, and what I present is actually one of several versions that I have developed over the years (this version is much richer than the one described in [HMGW96]; it is the "Haskore in practice" version alluded to in Section 6 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 The Basics

```
> module Basics where
> import Ix
> import Ratio
> infixr 5 :+:, :=:
```

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

```
> type Pitch      = (PitchClass, Octave)
> data PitchClass = 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,Show,Read)
> type Octave     = Int
```

So a `Pitch` 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 Ab) may be important (especially for notation). By convention, A440 = `(A,4)`.

Musical objects are captured by the `Music` datatype:[2]

```
> data Music = Note Pitch Dur [NoteAttribute]   -- a note \ atomic
>            | Rest Dur                         -- a rest /    objects
>            | Music :+: Music                  -- sequential composition
>            | Music :=: Music                  -- parallel composition
>            | Tempo  (Ratio Int) Music         -- scale the tempo
>            | Trans  Int Music                 -- transposition
>            | Instr  IName Music               -- instrument label
>            | Player PName Music               -- player label
>            | Phrase [PhraseAttribute] Music   -- phrase attributes
>     deriving (Show, Eq)
>
> type Dur   = Ratio Int                        -- in whole notes
> type IName = String
> type PName = String
```

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 two atomic constructors we can build more complex musical objects using the other constructors, as follows:

- `m1 :+: m2` is the sequential composition of `m1` and `m2`; i.e. `m1` and `m2` are played in sequence.
- `m1 :=: m2` is the parallel composition of `m1` and `m2`; i.e. `m1` and `m2` are played simultaneously.

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

- `Tempo a m` scales the rate at which `m` is played (i.e. its tempo) by a factor of `a`.
- `Trans i m` transposes `m` by interval `i` (in semitones).
- `Instr iname m` declares that `m` is to be performed using instrument `iname`.
- `Player pname m` declares that `m` is to be performed by player `pname`.
- `Phrase pas m` declares that `m` is to be played using the phrase attributes (described later) in the list `pas`.

It is convenient to represent these ideas in Haskell as a recursive datatype 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.

3.1 Convenient Auxiliary Functions

In anticipation of the need to translate between different number types, we define the following coercion functions:

```
> rtof  :: Ratio Int -> Float
> rtof r = float (numerator r) / float (denominator r)
>
> float :: Int -> Float
> float  = fromInteger . toInteger
```

Treating pitches simply as integers is also useful in many settings, so let's also define some functions for converting between `Pitch` values and `AbsPitch` values (integers):

```
> type AbsPitch = Int
>
> absPitch :: Pitch -> AbsPitch
> absPitch (pc,oct) = 12*oct + pitchClass pc
>
> pitch    :: AbsPitch -> Pitch
> pitch    ap       = ( [C,Cs,D,Ds,E,F,Fs,G,Gs,A,As,B] !! mod ap 12,
>                       quot ap 12)
>
> pitchClass :: PitchClass -> Int
> pitchClass 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?
```

We can also define a function `trans`, which transposes pitches (analogous to `Trans`, which transposes values of type `Music`):

```
> trans    :: Int -> Pitch -> Pitch
> trans i p = pitch (absPitch p + i)
```

Finally, for convenience, let's create simple names for familiar notes, durations, and rests, as shown in Figure 2. Despite the large number of them, these names are sufficiently "unusual" that name clashes are unlikely.

Exercise 1 *Show that* `abspitch . pitch = id`*, and, up to enharmonic equivalences,* `pitch . abspitch = id`*.*

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

3.2 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 :: [Music] -> Music
> line  = foldr1 (:+:)
> chord = foldr1 (:=:)
```

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

```
> cMaj = [ n 4 qn [] | n <- [c,e,g] ]  -- octave 4, quarter notes
>
> cMajArp = line  cMaj
> cMajChd = chord cMaj
```

```
> cf,c,cs,df,d,ds,ef,e,es,ff,f,fs,gf,g,gs,af,a,as,bf,b,bs ::
>    Octave -> Dur -> [NoteAttribute] -> Music
>
> cf o = Note (Cf,o);  c o = Note (C,o);  cs o = Note (Cs,o)
> df o = Note (Df,o);  d o = Note (D,o);  ds o = Note (Ds,o)
> ef o = Note (Ef,o);  e o = Note (E,o);  es o = Note (Es,o)
> ff o = Note (Ff,o);  f o = Note (F,o);  fs o = Note (Fs,o)
> gf o = Note (Gf,o);  g o = Note (G,o);  gs o = Note (Gs,o)
> af o = Note (Af,o);  a o = Note (A,o);  as o = Note (As,o)
> bf o = Note (Bf,o);  b o = Note (B,o);  bs o = Note (Bs,o)
>
> 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  :: Music
> dwnr, dhnr, dqnr, denr, dsnr, dtnr :: Music
> ddhnr, ddqnr, ddenr                :: Music
>
> 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 = 3%2       ; dwnr = Rest dwn     -- dotted whole note rest
> dhn = 3%4       ; dhnr = Rest dhn     -- dotted half note rest
> dqn = 3%8       ; dqnr = Rest dqn     -- dotted quarter note rest
> den = 3%16      ; denr = Rest den     -- dotted eighth note rest
> dsn = 3%32      ; dsnr = Rest dsn     -- dotted sixteenth note rest
> dtn = 3%64      ; dtnr = Rest dtn     -- dotted thirty-second note rest
>
> ddhn = 7%8      ; ddhnr = Rest ddhn   -- double-dotted half note rest
> ddqn = 7%16     ; ddqnr = Rest ddqn   -- double-dotted quarter note rest
> dden = 7%32     ; ddenr = Rest dden   -- double-dotted eighth note rest
```

Figure 2: Convenient note names and pitch conversion functions.

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 :=: Trans 7 m`." Similarly, a canon-like structure involving `m` can be expressed as "`m :=: delay d m`," where:

```
> delay :: Dur -> Music -> Music
> delay d m = 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:

```
> repeatM :: Music -> Music
> repeatM m = m :+: repeatM 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:

```
> lineToList :: Music -> [Music]
> lineToList n@(Rest 0) = []
> lineToList (n :+: ns) = n : lineToList ns
>
> retro, invert, retroInvert, invertRetro :: Music -> Music
> retro    = line . reverse . lineToList
> invert m = line (map inv l)
>   where l@(Note r _ _: _)  = lineToList m
>         inv (Note p d nas) = Note (pitch (2*(absPitch r) - absPitch p)) d nas
>         inv (Rest d)       = Rest d
> retroInvert = retro  . invert
> invertRetro = invert . retro
```

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

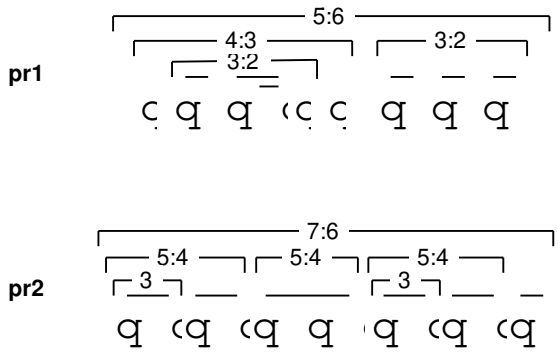


Figure 3: Nested Polyrhythms

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 3. 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 -> Music
> pr1 p = Tempo (5%6) (Tempo (4%3) (mkLn 1 p qn :+:
>                                   Tempo (3%2) (mkLn 3 p en :+:
>                                               mkLn 2 p sn :+:
>                                               mkLn 1 p qn    ) :+:
>                                   mkLn 1 p qn) :+:
>                     Tempo (3%2) (mkLn 6 p en))
>
> pr2 p = Tempo (7%6) (m1 :+:
>                     Tempo (5%4) (mkLn 5 p en) :+:
>                     m1 :+:
>                     mkLn 2 p en)
>         where m1 = Tempo (5%4) (Tempo (3%2) m2 :+: m2)
>               m2 = mkLn 3 p en
>
> mkLn n p d = line (take n (repeat (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
> pr12 = pr1 (C,5) :=: pr2 (G,5)
```

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:

```
> (=:=) :: Dur -> Dur -> Music -> Music
> old =:= new  =  Tempo (new/old)
```

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

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

```
> dur :: Music -> Dur
>
> dur (Note _ d _)  = d
> dur (Rest d)      = d
> dur (m1 :+: m2)   = dur m1   +   dur m2
> dur (m1 :=: m2)   = dur m1 `max` dur m2
> dur (Tempo  a  m) = dur m / a
> dur (Trans  _  m) = dur m
> dur (Instr  _  m) = dur m
> dur (Player _  m) = dur m
> dur (Phrase _  m) = dur m
```

Super-retrograde. Using `dur` we can define a function `revM` that reverses any `Music` value (and is thus considerably more useful than `retro` defined earlier). Note the tricky treatment of `(:=:)`.

```
> revM :: Music -> Music
> revM n@(Note _ _ _) = n
> revM r@(Rest _)     = r
> revM (Tempo a  m)   = Tempo a    (revM m)
> revM (Trans i  m)   = Trans i    (revM m)
> revM (Instr i  m)   = Instr i    (revM m)
> revM (Phrase pas m) = Phrase pas (revM m)
> revM (m1 :+: m2)    = revM m2 :+: revM m1
> revM (m1 :=: m2)    =
>   let d1 = dur m1
```

```
>       d2 = dur m2
>   in if d1>d2 then revM m1 :=: (Rest (d1-d2) :+: revM m2)
>               else (Rest (d2-d1) :+: revM m1) :=: revM m2
```

Truncating Parallel Composition Note that the duration of `m1 :=: m2` is the maximum of the durations of {`m1` 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 `cut` such that `cut d m` is the musical object `m` "cut short" to have at most duration `d`:

```
> cut :: Dur -> Music -> Music
> cut newDur m | newDur <= 0   = Rest 0
> cut newDur (Note x oldDur y) = Note x (min oldDur newDur) y
> cut newDur (Rest oldDur)     = Rest (min oldDur newDur)
> cut newDur (m1 :=: m2)       = cut newDur m1 :=: cut newDur m2
> cut newDur (m1 :+: m2)       = let m1' = cut newDur m1
>                                    m2' = cut (newDur - dur m1') m2
>                                in m1' :+: m2'
> cut newDur (Tempo  a m)      = Tempo a  (cut (newDur * a) m)
> cut newDur (Trans  a m)      = Trans a  (cut newDur m)
> cut newDur (Instr  a m)      = Instr a  (cut newDur m)
> cut newDur (Player a m)      = Player a (cut newDur m)
> cut newDur (Phrase a m)      = Phrase a (cut newDur m)
```

Note that `cut` is equipped to handle a `Music` object of infinite length.

With `cut`, the definition of `(/=:)` is now straightforward:

```
> (/=:) :: Music -> Music -> Music
> m1 /=: m2 = cut (min (dur m1) (dur m2)) (m1 :=: m2)
```

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

Exercise 4 *Define a version of* `(/=:)` *that shortens correctly when either or both of its arguments are infinite in duration.*

Trills A *trill* is an ornament that alternates rapidly between two (usually adjacent) pitches. We will define two versions of a trill function, both of which take the starting note and an interval for the trill note as arguments (the interval is usually one or two, but can actually be anything). One version will additionally have an argument that specifies how long each trill note should be, whereas the other will have an argument that specifies how many trills should occur. In both cases the total duration will be the same as the duration of the original note.

Here is the first trill function:

```
> trill :: Int -> Dur -> Music -> Music
> trill i sDur (Note p tDur x) =
>       if sDur >= tDur then Note p tDur x
>       else Note p sDur x
>            :+: trill (negate i) sDur (Note (trans i p) (tDur-sDur) x)
> trill i d (Tempo  a m) = Tempo  a (trill i (d*a) m)
> trill i d (Trans  a m) = Trans  a (trill i d m)
> trill i d (Instr  a m) = Instr  a (trill i d m)
> trill i d (Player a m) = Player a (trill i d m)
> trill i d (Phrase a m) = Phrase a (trill i d m)
> trill _ _ _ = error "Trill input must be a single note."
```

It is simple to define a version of this function that starts on the trill note rather than the start note:

```
> trill' :: Int -> Dur -> Music -> Music
> trill' i sDur m =
>       trill (negate i) sDur (Trans i m)
```

The second way to define a trill is in terms of the number of subdivided notes to be included in the trill. We can use the first trill function to define this new one:

```
> trilln :: Int -> Int -> Music -> Music
> trilln i nTimes m =
>       trill i (dur m / (nTimes%1)) m
```

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

```
> trilln' :: Int -> Int -> Music -> Music
> trilln' i nTimes m =
>       trilln (negate i) nTimes (Trans i m)
```

```
> data PercussionSound =
>         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 4: General Midi Percussion Names

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

```
> roll  :: Dur -> Music -> Music
> rolln :: Int -> Music -> Music
>
> roll  dur    m = trill  0 dur m
> rolln nTimes m = trilln 0 nTimes m
```

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 4, 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` object; i.e. a `Note`:

```
> perc :: PercussionSound -> Dur -> [NoteAttribute] -> Music
> perc ds dur na = Note (pitch (fromEnum ds + 35)) dur na
```

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

```
> funkGroove
>   = let p1 = perc LowTom        qn []
>         p2 = perc AcousticSnare en []
>     in Tempo 3 (Instr "Drums" (cut 8 (repeatM
>          ( (p1 :+: qnr :+: p2 :+: qnr :+: p2 :+:
>             p1 :+: p1 :+: qnr :+: p2 :+: enr)
>            :=: roll en (perc ClosedHiHat 2 []) )
>        )))
```

Exercise 5 *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.*

Appendix C shows the first 28 bars of Chick Corea's "Children's Song No. 6" encoded in Haskore.

3.3 Phrasing and Articulation

Recall that the `Note` constructor contained a field of `NoteAttributes`. 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 `PhraseAttributes`. These two attribute datatypes cover a wide range of attributions found in common practice notation, and are shown in Figure 5. 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.

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:

```
> data NoteAttribute = Volume Float        -- by convention: 0=min, 100=max
>                    | Fingering Int
>                    | Dynamics String
>      deriving (Eq, Show)
>
> data PhraseAttribute = Dyn Dynamic
>                      | Art Articulation
>                      | Orn Ornament
>      deriving (Eq, Show)
>
> data Dynamic = Accent Float | Crescendo Float | Diminuendo Float
>              | PPP | PP | P | MP | SF | MF | NF | FF | FFF | Loudness Float
>              | Ritardando Float | Accelerando Float
>      deriving (Eq, 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, Show)
>
> data Ornament = Trill | Mordent | InvMordent | DoubleMordent
>               | Turn | TrilledTurn | ShortTrill
>               | Arpeggio | ArpeggioUp | ArpeggioDown
>               | Instruction String | Head NoteHead
>      deriving (Eq, Show)
>
> data NoteHead = DiamondHead | SquareHead | XHead | TriangleHead
>               | TremoloHead | SlashHead | ArtHarmonic | NoHead
>      deriving (Eq, Show)
```

Figure 5: Note and Phrase Attributes.

```
> legato, staccato  :: Articulation
> accent, bigAccent :: Dynamic
>
> legato    = Legato 1.1
> staccato  = Staccato 0.5
> accent    = 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 5 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 `Players` and `Performance` modules are mutually recursive).

4 Interpretation and Performance

```
> module Performance (module Performance, module Basics) -- module Players
>        where
>
> import Basics
> -- import Players
```

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 Performance = [Event]
>
> data Event = Event {eTime :: Time, eInst :: IName, ePitch :: AbsPitch,
>                     eDur  :: DurT, eVol  :: Volume, pFields :: [Float]}
>      deriving (Eq,Ord,Show)
>
> type Time      = Float
> type DurT      = Float
> type Volume    = Float
```

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, eVol`

= v} captures the fact that at start time s, instrument i sounds pitch p with volume 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 PNames in an abstract musical object to the actual players to be used. (We don't yet need a mapping from abstract INames to instruments, since this is handled in the translation from a performance into, say, Midi, such as defined in Section 6.)

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

```
> perform :: PMap -> Context -> Music -> Performance
>
> type PMap    = PName -> Player
> data Context = Context {cTime :: Time, cPlayer :: Player, cInst :: IName,
>                         cDur  :: DurT, cKey    :: Key,    cVol  :: Volume}
>      deriving Show
>
> type Key     = AbsPitch
```

```
perform 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    -> perform pmap c m1 ++
                    perform pmap (c {cTime = t + dur m1 * dt}) m2
    m1 :=: m2    -> merge (perform pmap c m1) (perform pmap c m2)
    Tempo  a   m -> perform pmap (c {cDur = dt / rtof a}) m
    Trans  p   m -> perform pmap (c {cKey = k + p}) m
    Instr  nm  m -> perform pmap (c {cInst = nm}) m
    Player nm  m -> perform 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 DurT field of the context.

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.

```
> perform pmap c m = fst (perf pmap c m)
>
> perf :: PMap -> Context -> Music -> (Performance, DurT)
>
> perf 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, rtof d *dt)
>     Rest d       -> ([], rtof d *dt)
>     m1 :+: m2    -> let (pf1,d1) = perf pmap c m1
>                         (pf2,d2) = perf pmap (c {cTime = t+d1}) m2
>                     in (pf1++pf2, d1+d2)
>     m1 :=: m2    -> let (pf1,d1) = perf pmap c m1
>                         (pf2,d2) = perf pmap c m2
>                     in (merge pf1 pf2, max d1 d2)
>     Tempo  a   m -> perf pmap (c {cDur = dt / rtof a}) m
>     Trans  p   m -> perf pmap (c {cKey = k + p}) m
>     Instr  nm  m -> perf pmap (c {cInst = nm}) m
>     Player nm  m -> perf pmap (c {cPlayer = pmap nm}) m
>     Phrase pas m -> interpPhrase pl pmap c pas m
```

Figure 6: The “real” `perform` function.

3. The `DurT` component of the context is the duration, in seconds, of one whole note. To make it easier to compute, 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 :: Float -> Dur -> DurT
> metro setting dur = 60 / (setting * rtof dur)
```

   Thus, for example, `metro 96 qn` creates a tempo of 96 quarter notes per minute.

4. In the treatment of `(:+:)`, 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 3.2) is used to compute this duration. Note that this results in a quadratic time complexity for `perform`. A more efficient solution is to have `perform` compute the duration directly, returning it as part of its result. This version of `perform` is shown in Figure 6.

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

```
> merge :: Performance -> Performance -> Performance

  merge a@(e1:es1) b@(e2:es2) =
        if e1 < e2 then e1 : merge es1 b
                   else e2 : merge a es2
  merge [] es2 = es2
  merge es1 [] = es1
```

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 11. Here is a more efficient version that will work just as well in practice:

```
> merge a@(e1:es1) b@(e2:es2) =
>   if eTime e1 < eTime e2 then e1 : merge es1 b
>                          else e2 : merge a es2
> merge [] es2 = es2
> merge es1 [] = es1
```

5 Players

```
module Players (module Players, module Music, module Performance)
       where

import Music
import Performance
```

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 3.3) are player and instrument dependent. For example, how should "legato" be interpreted in a performance? Or "diminuendo?" Different players interpret things in different ways, of course, but even more fundamental is the fact that a pianist, for example, realizes legato in a way fundamentally different from the way a violinist does, because of differences in their instruments. Similarly, diminuendo on a piano and a harpsichord are different concepts.

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

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

```
> data Player = MkPlayer { pName :: PName,
>                          playNote :: NoteFun,
>                          interpPhrase :: PhraseFun,
>                          notatePlayer :: NotateFun }
>      deriving Show

> type NoteFun   =
>      Context -> Pitch -> Dur -> [NoteAttribute] -> Performance
> type PhraseFun =
>      PMap -> Context -> [PhraseAttribute] -> Music -> (Performance,DurT)
> type NotateFun = ()
```

The last line above is because notation is currently not implemented. Note that both `NoteFun` and `PhraseFun` functions return a `Performance` (imported from module `Perform`).

```
> defPlayer :: Player
> defPlayer  = MkPlayer { pName        = "Default",
>                         playNote     = defPlayNote defNasHandler,
>                         interpPhrase = defInterpPhrase defPasHandler,
>                         notatePlayer = defNotatePlayer ()           }
>
> defPlayNote :: (Context->NoteAttribute->Event->Event) -> NoteFun
> defPlayNote nasHandler
>   c@(Context cTime cPlayer cInst cDur cKey cVol) p d nas =
>        [ foldr (nasHandler c)
>                (Event {eTime  = cTime, eInst  = cInst,
>                        ePitch = absPitch p + cKey,
>                        eDur   = rtof d * cDur, eVol = cVol})
>                nas ]
>
> defNasHandler :: Context-> NoteAttribute -> Event -> Event
> defNasHandler c (Volume v) ev = ev {eVol = (cVol c + v)/2}
> defNasHandler _              _          ev = ev
>
> defInterpPhrase :: (PhraseAttribute->Performance->Performance) -> PhraseFun
> defInterpPhrase pasHandler pmap context pas m =
>        let (pf,dur) = perf pmap context m
>        in  (foldr pasHandler pf pas, dur)
>
> defPasHandler :: PhraseAttribute -> Performance -> Performance
> defPasHandler (Dyn (Accent x))   pf = map (\e -> e {eVol = x * eVol e}) pf
> defPasHandler (Art (Staccato x)) pf = map (\e -> e {eDur = x * eDur e}) pf
> defPasHandler (Art (Legato   x)) pf = map (\e -> e {eDur = x * eDur e}) pf
> defPasHandler _                  pf = pf
>
> defNotatePlayer   :: () -> NotateFun
> defNotatePlayer _ = ()
```

Figure 7: Definition of default Player `defPlayer`.

5.1 Examples of Player Construction

A "default player" called `defPlayer` (not to be confused with "deaf player"!) is defined for use when none other is specified in the score; it also functions as a base from which other players can be derived. `defPlayer` responds only to the `Volume` note attribute and to the `Accent`, `Staccato`, and `Legato` phrase attributes. It is defined in Figure 7. Before reading this code, recall how players are invoked by the `perform` function defined in the last section; in particular, note the calls to `playNote` and `interpPhase` defined above. Then note:

1. `defPlayNote` is the only function (even in the definition of `perform`) that actually generates an event. It also modifies that event based on an interpretation of each note attribute by the function `defHasHandler`.
2. `defNasHandler` only recognizes the `Volume` attribute, which it uses to set the event volume accordingly.
3. `defInterpPhrase` calls (mutually recursively) `perform` to interpret a phrase, and then modifies the result based on an interpretation of each phrase attribute by the function `defPasHandler`.
4. `defPasHandler` only recognizes the `Accent`, `Staccato`, and `Legato` phrase attributes. For each of these it uses the numeric argument as a "scaling" factor of the volume (for `Accent`) and duration (for `Staccato` and `Lagato`). Thus `(Phrase [Legato 1.1] m)` effectively increases the duration of each note in `m` by 10% (without changing the tempo).

It should be clear that much of the code in Figure 7 can be re-used in defining a new player. For example, to define a player `weird` that interprets note attributes just like `defPlayer` but behaves differently with respect to phrase attributes, we could write:

```
weird :: Player
weird  = MkPlayer { pname        = "Weirdo",
                    playNote     = defPlayNote defNasHandler,
                    interpPhrase = defInterpPhrase myPasHandler
                    notatePlayer = defNotatePlayer ()          }
```

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

```
myPasHandler :: PhraseAttribute -> Performance -> Performance
myPasHandler (Dyn (Crescendo x)) pf = ...
myPasHandler  pa                 pf = defPasHandler pa pf
```

Exercise 6 *Fill in the* `...` *in the definition of* `myPasHandler` *according to the following strategy: Assume* $0 <$ `x` < 1. *Gradually scale the volume of each event by a factor of* 1.0 *through* 1.0 + `x`*, using linear interpolation.*

Exercise 7 *Choose some of the other phrase attributes and provide interpretations of them, such as* `Diminuendo`, `Slurred`, `Trill`, *etc. (The* `trill` *functions from section 3.2 may be useful here.)*

Figure 8 defines a relatively sophisticated player called `fancyPlayer` that knows all that `defPlayer` knows, and much more. Note that `Slurred` is different from `Legato` in that it doesn't extend the duration of the *last* note(s). The behavior of (`Ritardando` x) can be explained as follows. We'd like to "stretch" the time of each event by a factor from 0 to x, linearly interpolated based on how far along the musical phrase the event occurs. I.e., given a start time t_0 for the first event in the phrase, total phrase duration D, and event time t, the new event time t' is given by:

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

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

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

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

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

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

6 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

```
> fancyPlayer :: Player
> fancyPlayer  = MkPlayer { pName        = "Fancy",
>                           playNote     = defPlayNote defNasHandler,
>                           interpPhrase = fancyInterpPhrase,
>                           notatePlayer = defNotatePlayer ()        }
> fancyInterpPhrase :: PhraseFun
> fancyInterpPhrase pmap c [] m = perf pmap c m
> fancyInterpPhrase pmap c@Context {cTime = t, cPlayer = pl, cInst = i,
>                                   cDur = dt, cKey = k,     cVol = v}
>                   (pa:pas) m =
>  let pfd@(pf,dur) = fancyInterpPhrase pmap c pas m
>      loud x       = fancyInterpPhrase pmap c (Dyn (Loudness x) : pas) m
>      stretch x = let t0 = eTime (head pf);  r  = x/dur
>                      upd (e@Event {eTime = t, eDur = d}) =
>                          let dt = t-t0
>                              t' = (1+dt*r)*dt + t0
>                              d' = (1+(2*dt+d)*r)*d
>                          in e {eTime = t', eDur = d'}
>                  in (map upd pf, (1+x)*dur)
>      inflate x = let t0 = eTime (head pf);  r  = x/dur
>                      upd (e@Event {eTime = t, eVol = v}) =
>                          e {eVol = (1+(t-t0)*r)*v}
>                  in (map upd pf, dur)
>  in case pa of
>     Dyn (Accent x)      -> (map (\e-> e {eVol = x * eVol e}) pf, dur)
>     Dyn PPP -> loud  40  ;  Dyn  PP -> loud  50  ;  Dyn   P -> loud  60
>     Dyn  MP -> loud  70  ;  Dyn  SF -> loud  80  ;  Dyn  MF -> loud  90
>     Dyn  NF -> loud 100  ;  Dyn  FF -> loud 110  ;  Dyn FFF -> loud 120
>     Dyn (Loudness x)    -> fancyInterpPhrase pmap c {cVol = x} pas m
>     Dyn (Crescendo x)   -> inflate   x ; Dyn (Diminuendo x)  -> inflate (-x)
>     Dyn (Ritardando  x) -> stretch   x ; Dyn (Accelerando x) -> stretch (-x)
>     Art (Staccato x)    -> (map (\e-> e {eDur = x * eDur e}) pf, dur)
>     Art (Legato   x)    -> (map (\e-> e {eDur = x * eDur e}) pf, dur)
>     Art (Slurred  x)    ->
>         let lastStartTime = foldr (\e t -> max (eTime e) t) 0 pf
>             setDur e      = if eTime e < lastStartTime
>                             then e {eDur = x * eDur e}
>                             else e
>         in (map setDur pf, dur)
>     Art _               -> pfd  -- Remaining articulations:
>                 --   Tenuto | Marcato | Pedal | Fermata  | FermataDown
>                 -- | Breath | DownBow | UpBow | Harmonic | Pizzicato
>                 -- | LeftPizz | BartokPizz | Swell | Wedge | Thumb | Stopped
>     Orn _               -> pfd  -- Remaining ornamenations:
>                 --   Trill | Mordent | InvMordent | DoubleMordent | Turn
>                 -- | TrilledTurn | ShortTrill | Arpeggio | ArpeggioUp
>                 -- | ArpeggioDown | Instruction String | Head NoteHead
>     -- Design Bug: To do these right we need to keep the KEY SIGNATURE
>     -- around so that we can determine, for example, what the trill note is.
>     -- Alternatively, provide an argument to Trill to carry this info.
```


Figure 8: Definition of Player `fancyPlayer`.

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.

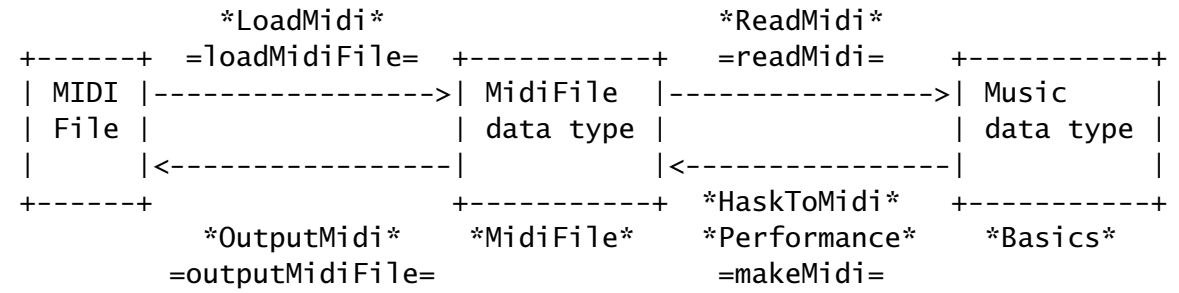
If you decide to use the General Midi features of your sound-card, you need to know about another set of conventions known as "Basic Midi." The most important aspect of Basic Midi is that Channel 10 (9 in Haskore's 0-based numbering) is dedicated to *percussion*. A future release of Haskore should make 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 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 HaskToMidi (performToMidi, UserPatchMap,
>                    module GeneralMidi, module OutputMidi,
>                    module LoadMidi, module MidiFile)
>        where
> import Basics
> import Performance
> import MidiFile
> import GeneralMidi
> import List(partition)
> import Char(toLower,toUpper)
> import OutputMidi
> import LoadMidi
```

Instead of converting a Haskore `Performance` 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 (=):



A UserPatchMap is a user-supplied table for mapping instrument names (IName's) 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 UserPatchMap = [(IName,GenMidiName,MidiChannel)]
```

See Appendix A for an example of a useful user patch map.

Given a UserPatchMap, a performance is converted to a datatype representing a Standard Midi File using the performToMidi function. If the given UserPatchMap is invalid, it creates a new one by matching the instrument names with General Midi names and mapping the instruments to channels one by one.

```
> performToMidi :: Performance -> UserPatchMap -> MidiFile
> performToMidi pf pMap =
>   let splitList   = splitByInst pf
>       insts       = map fst splitList
>       rightMap    = if (allValid pMap insts)
>                       then pMap
>                       else (makeGMMap insts)
>   in MidiFile mfType (Ticks division)
>          (map (performToMEvs rightMap) (splitByInst pf))
```

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

The following function is used to test whether or not every instrument in a list is found in a UserPatchMap.

```
> allValid :: UserPatchMap -> [IName] -> Bool
> allValid upm = and . map (lookupB upm)
>
> lookupB :: UserPatchMap -> IName -> Bool
> lookupB [] _           = False
> lookupB ((y,_,_):ys) x = x `partialMatch` y || lookupB ys x
```

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

```
> makeGMMap :: [IName] -> UserPatchMap
> makeGMMap iList = makeGMMap' 0 iList
>   where makeGMMap' _ []        = []
>         makeGMMap' n _ | n>=15 =
>                    error "Too many instruments; not enough MIDI channels."
>         makeGMMap' n (i:is)    =
>                    if map toLower i `elem` percList
>                    then (i, "Acoustic Grand Piano", 9) : makeGMMap' n is
>                    else (i, gmMatch i genMidiMap, chanList !! n)
>                                                      : makeGMMap' (n+1) is
>         gmMatch i ((gmInst,chan):gcs) =
>                   if i `partialMatch` gmInst -- or use == ?
>                   then gmInst
>                   else gmMatch i gcs
>         gmMatch i [] = error("instrument " ++ i ++ " not found")
>         percList = ["percussion", "perc", "drums"]
>         chanList = [0..8] ++ [10..15]  -- 10th channel (#9) is for percussion
```

6.1 The Gory Details

Some preliminaries, otherwise known as constants:

```
> mfType   = 1   :: MFType -- midi-file type 1 always used
> division = 96  :: Int    -- time-code division: 96 ticks per quarter note
```

Since we are implementing 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.)

```
> splitByInst :: Performance ->  [(IName,Performance)]
> splitByInst [] = []
> splitByInst pf = (i,pf1) : splitByInst pf2
>                    where i         = eInst (head pf)
>    (pf1,pf2) = partition (\e -> eInst e == i) pf
```

The crux of the conversion process is `performToMEvs`, which converts a `Performance` into a stream of `MEvents`.

```
> performToMEvs :: UserPatchMap -> (IName,Performance) -> [MEvent]
> performToMEvs pMap (inm,perf) =
>   let (midiChan,progNum) = unMap pMap inm
>       setupInst          = MidiEvent 0 (ProgChange midiChan progNum)
>       setTempo           = MetaEvent 0 (SetTempo defST)
>       loop []     = []
>       loop (e:es) = let (mev1,mev2) = mkMEvents midiChan e
>                     in  mev1 : insertMEvent mev2 (loop es)
>   in  setupInst : setTempo : loop perf
```

A source of incompatibilty 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 :: MidiChannel -> Event -> (MEvent,MEvent)
> mkMEvents mChan (Event {eTime = t, ePitch = p, eDur = d, eVol = v})
>                   = (MidiEvent (toDelta t)     (NoteOn  mChan p v'),
>                      MidiEvent (toDelta (t+d)) (NoteOff mChan p v') )
>           where v' = min 127 (round v)
>
> toDelta t = round (t * 4.0 * float division)
```

The final critical function is `insertMEvent`, which inserts an `MEvent` into an already time-ordered sequence of `MEvents`.

```
> insertMEvent :: MEvent -> [MEvent] -> [MEvent]
> insertMEvent mev1  []         = [mev1]
> insertMEvent mev1@(MidiEvent t1 _) mevs@(mev2@(MidiEvent t2 _):mevs') =
>       if t1 <= t2 then mev1 : mevs
>                   else mev2 : insertMEvent mev1 mevs'
```

The following functions lookup `IName`'s in `UserPatchMaps` 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.

```
> unMap :: UserPatchMap -> IName -> (MidiChannel,ProgNum)
> unMap pMap iName = (channel, gmProgNum gmName)
>   where (gmName, channel) = lookup23 iName pMap
>
> gmProgNum :: GenMidiName -> ProgNum
> gmProgNum gmName = lookup2 gmName genMidiMap
>
> 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'
>
> lookup2  x ((y,z):ys) = if x `partialMatch` y then z else lookup2 x ys
> lookup2  x []         = error ("Instrument " ++ x ++ " unknown")
>
> lookup23 x ((y,z,q):ys) = if x `partialMatch` y then (z,q) else lookup23 x ys
> lookup23 x []         = error ("Instrument " ++ x ++ " unknown")
```

6.2 Midi-File Datatypes

```
> module MidiFile(
>  MidiFile(..), Division(..), Track, MFType, MEvent(..), ElapsedTime,
>  MPitch, Velocity, ControlNum, PBRange, ProgNum, Pressure,
>  MidiChannel, ControlVal,
>  MidiEvent(..),
```

```
>  MTempo, SMPTEHours, SMPTEMins, SMPTESecs, SMPTEFrames, SMPTEBits,
>  MetaEvent(..),
>  KeyName(..), Mode(..),
>        defST, defDurT
>  ) where

> import Ix



The datatypes for Midi Files and Midi Events
--------------------------------------------------------------------------

> data MidiFile = MidiFile MFType Division [Track] deriving (Show, Eq)
>
> data Division = Ticks Int | SMPTE Int Int
>      deriving (Show,Eq)
>
> type Track  = [MEvent]
> type MFType = Int
>
> data MEvent = MidiEvent ElapsedTime MidiEvent
>             | MetaEvent ElapsedTime MetaEvent
>             | NoEvent
>      deriving (Show,Eq)
>
> type ElapsedTime  = Int
>
> -- Midi Events
>
> type MPitch      = Int
> type Velocity    = Int
> type ControlNum  = Int
> type PBRange     = Int
> type ProgNum     = Int
> type Pressure    = Int
> type MidiChannel = Int
> type ControlVal  = Int
>
> data MidiEvent = NoteOff    MidiChannel MPitch Velocity
>                | NoteOn     MidiChannel MPitch Velocity
>                | PolyAfter  MidiChannel MPitch Pressure
>                | ProgChange MidiChannel ProgNum
>                | Control    MidiChannel ControlNum ControlVal
>                | PitchBend  MidiChannel PBRange
```

```
>                | MonoAfter  MidiChannel Pressure
>      deriving (Show, Eq)
>
> -- Meta Events
>
> type MTempo      = 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 MidiChannel
>                | EndOfTrack
>                | SetTempo MTempo
>                | SMPTEOffset SMPTEHours SMPTEMins SMPTESecs SMPTEFrames SMPTEBits
>                | TimeSig Int Int Int Int
>                | KeySig KeyName Mode
>                | SequencerSpecific [Int]
>                | Unknown String
>      deriving (Show, Eq)
>
```

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 KeyName = 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)
```

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.

```
> defDurT = 2 :: Float
> defST = truncate (1000000 / defDurT) :: Int
```

7 Outputting MIDI Files

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

```
> module OutputMidi (outputMidiFile, midiFileToString) where
> import MidiFile
> import IOExtensions (writeBinaryFile)
> import Monads (Output, runO, outO)
> import Bitops (bSplitAt, someBytes)
> import Ix
```

`OutputMidiFile` is the main function for writing `MidiFile` values to an actual file; its first argument is the filename:

```
> outputMidiFile :: String -> MidiFile -> IO ()
> outputMidiFile fn mf = writeBinaryFile fn (midiFileToString mf)
```

Exercise 8 *Take as many examples as you like from the previous sections, create one or more* `UserPatchMaps`*, write the examples to a file, and play them using a conventional Midi player.*

Appendix A defines some functions which should make the above exercise easier. Appendices B, C, and D 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`.

```
> midiFileToString :: MidiFile -> String
> midiFileToString = runM . outMF
>
> outMF :: MidiFile -> MidiWriter ()
> outMF (MidiFile mft divisn trks) =
>   do
>     outChunk "MThd" (do
>                         out 2 mft                   -- format (type 0, 1 or 2)
>                         out 2 (length trks)         -- length of tracks to come
>                         outputDivision divisn)      -- time unit
>     outputTracks trks
>
> outputDivision :: Division -> MidiWriter ()
> outputDivision (Ticks nticks)      = out 2 nticks
> outputDivision (SMPTE mode nticks) = do
>                                        out 1 (256-mode)
>                                        out 1 nticks
>
> outputTracks :: [Track] -> MidiWriter ()
> outputTracks trks = mapM_ outputTrack trks
>
> outputTrack :: Track -> MidiWriter ()
> outputTrack trk = outChunk "MTrk" (mapM_ outputEvent (delta trk))
```

`delta` converts a track using absolute time to one using delta time, adding EndOfTrack if not already there.

```
> delta :: Track -> Track
> delta [] = []
> delta trk | notEOT (last trk) = trk' ++ [MetaEvent 0 EndOfTrack]
>           | otherwise         = trk'
>  where
>   (t,trk') = mscanl delta' 0 trk
>   delta' :: Int ->       -- current time
>             MEvent ->    -- event
>             (Int,        -- new time
>              MEvent)     -- event
>   delta' t (MidiEvent dt e) = (dt  , MidiEvent (dt-t) e)
>   delta' t (MetaEvent dt e) = (dt  , MetaEvent (dt-t) e)
>   notEOT (MetaEvent _ EndOfTrack) = False
>   notEOT _                        = True
```

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

```
> outputEvent :: MEvent -> MidiWriter ()
> outputEvent (MidiEvent dt mevent) = do
>                                       outVar dt
>                                       outputMidiEvent mevent
> outputEvent (MetaEvent dt mevent) = do
>                                       outVar dt
>                                       outputMetaEvent mevent
> outputEvent _                     = outStr ""
>
> outputMidiEvent :: MidiEvent -> MidiWriter ()
> outputMidiEvent (NoteOff    c p v)   = outChan 128 c [p,v]
> outputMidiEvent (NoteOn     c p v)   = outChan 144 c [p,v]
> outputMidiEvent (PolyAfter  c p pr)  = outChan 160 c [p,pr]
> outputMidiEvent (Control    c cn cv) = outChan 176 c [cn,cv]
> outputMidiEvent (ProgChange c pn)    = outChan 192 c [pn]
> outputMidiEvent (MonoAfter  c pr)    = outChan 208 c [pr]
> outputMidiEvent (PitchBend  c pb)    = outChan 224 c [lo,hi] -- small-endian!!
>  where (hi,lo) = bSplitAt 8 pb
>
> -- output a channel event
> outChan :: Int -> MidiChannel -> [Int] -> MidiWriter ()
> outChan code chan bytes = do
>                             out 1 (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 [convert sf, fromMode mi]
>                                       where k = index (KeyCf,KeyCs) sf - 7
>                                             convert 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 = Output Char a
>
> out :: Int -> Int -> MidiWriter ()
> outVar :: Int -> MidiWriter ()
> outList :: [Int] -> MidiWriter ()
> outStr  :: String -> MidiWriter ()
>
> runM :: MidiWriter a -> String
```

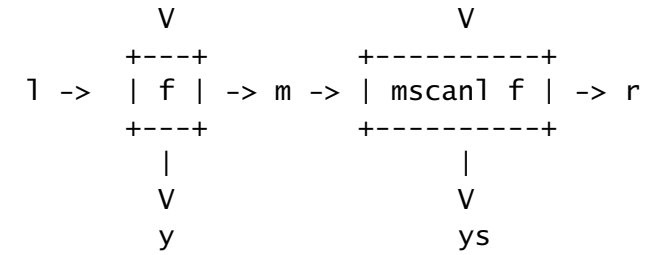
```
> runM m = snd (runO m)
>
> mLength  :: MidiWriter a -> Int
> mLength m = length (runM m)
>
> out 1 x = outO [toEnum x]
> out a x = mapM_ (out 1) (someBytes a x)
>
> outStr cs = outO cs
>
> outList xs = outStr (map toEnum xs)
```

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 n = do
>              outVarAux leftover
>              out 1 data7
>            where (leftover, data7) = bSplitAt 7 n
>                  outVarAux 0 = return ()
>                  outVarAux x = do
>                                  outVarAux leftover'
>                                  out 1 (128+data7')  --make signal bit 1
>                                where (leftover',data7') = bSplitAt 7 x
>
> fromMode :: Mode -> Int
> fromMode Major = 0
> fromMode Minor = 1
>
> -- Note: here I've chosen to compute the track twice
> -- rather than store it.  Other options are worth exploring.
>
> outChunk :: String -> MidiWriter a -> MidiWriter a
> outChunk tag m | length tag == 4 = do
>                                      outStr tag
>                                      out 4 (mLength m)
>                                      m
```

Mapping scan (used in function delta):

```
         x                 xs
         |                 |
```




```
> mscanl :: (a -> b -> (a,c)) -> a -> [b] -> (a,[c])
> mscanl f l [] = (l,[])
> mscanl f l (x:xs) = let (m, y ) = f l x
>                         (r, ys) = mscanl f m xs
>                     in (r, y:ys)
```

8 Loading MIDI Files

The LoadMidi 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 LoadMidi (loadMidiFile, showMidiFile)
>  where
> import MidiFile
> import IOExtensions (readBinaryFile)
> import Bitops (fromBytes, bshiftl, bTrunc, bSplitAt)
> import Maybe (fromJust)
> import HaskoreUtils (unlinesS, rightS, concatS)
> import Monad
```

The main load function.

```
> loadMidiFile :: String -> IO MidiFile
> loadMidiFile filename =
>   do
>     contents <- readBinaryFile filename
>     case runP parseMidiFile (contents, (AtBeginning,0),-1) of
>       Just (mf,"",_,_) -> return mf
>       Just (mf,_ ,_,_) -> do {print "Garbage left over." ; return mf}
>       Nothing          -> do print "Error reading midi file: unfamiliar format or fil
>                              return emptyMidiFile
```

```
>
> emptyMidiFile = MidiFile 0 (Ticks 0) [[]]
```

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`.

```
> parseMidiFile :: MidiReader MidiFile
> parseMidiFile = do
>                   chunk <- getChunk
>                   case chunk of
>                     Header (format, nTracks, div) ->
>                       do
>                         tracks <- getTracks nTracks
>                         return (MidiFile format div tracks)
>                     _                               -> parseMidiFile
```

Parse a number of track chunks. Like `parseMidiFile`, if a chunk is not a track chunk, it is just ignored.

```
> getTracks :: Int -> MidiReader [Track]
> getTracks 0 = return []
> getTracks n = do
>                  chunk <- getChunk
>                  tracks <- getTracks (n-1)
>                  case chunk of
>                    Track t -> return (t:tracks)
>                    _       -> return tracks
```

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"    -> do
>                                h <- header
>                                return (Header h)
>                   "MTrk"    -> do
>                                t <- track
>                                return (Track (undelta t))
>                   otherwise -> do
>                                g <- getN size
>                                return AlienChunk
>
> data Chunk = Header (Int, 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.

```
> header :: MidiReader (Int, Int, Division)
> header = do
>            format   <- get2
>            nTracks  <- get2
>            div      <- getDivision
>            return (format, nTracks, div)
```

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.

```
> track :: MidiReader [MEvent]
> track = do
>           size <- readSize
>           case size of
>             0 -> return []
>             _ -> do
>                    e  <- fancyEvent
>                    es <- track
>                    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.

```
> fancyEvent :: MidiReader MEvent
> fancyEvent = do
>                      time <- getVar
>                      e    <- event
>                      case e of
>                        Midi midiEvent -> return (MidiEvent time midiEvent)
>                        Meta metaEvent -> return (MetaEvent time metaEvent)
>                        _              -> return (NoEvent)
```

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.

```
> event :: MidiReader MidiFileEvent
> event = 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    <- metaEvent code size
>                         return (Meta e)
>          x | x>127 -> do
>                          firstData <- get1
>                          e <- midiEvent (decodeStatus tag) firstData
>                         return (Midi e)
>             _         -> do                        -- running status
>                          s <- readME
>                          e <- midiEvent s tag
>                          return (Midi e)
>
> data MidiFileEvent = SysexStart String         -- F0
>                    | SysexCont  String         -- F7
>                    | Midi       MidiEvent
>                    | Meta       MetaEvent
>                   deriving Show
```

Simpler versions of `fancyTrack` and `fancyEvent`, used in the Show functions.

```
> plainTrack :: MidiReader [(ElapsedTime, MidiFileEvent)]
> plainTrack = oneOrMore plainEvent
>
> plainEvent :: MidiReader (ElapsedTime, MidiFileEvent)
> plainEvent = do
>                time <- getVar
>                e    <- event
>                return (time,e)
>
> 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
>                 8  -> ItsaNoteOff
>                 9  -> ItsaNoteOn
>                 10 -> ItsaPolyAfter
>                 11 -> ItsaControl
>                 12 -> ItsaProgChange
>                 13 -> ItsaMonoAfter
>                 14 -> ItsaPitchBend
>                 _  -> error "invalid MidiEvent code"
>         (code, channel) = bSplitAt 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`.

```
> midiEvent :: Status -> Int -> MidiReader MidiEvent
> midiEvent s@(wME, channel) firstData
>   = do
>       setME s
>       case wME of
>         ItsaNoteOff    -> do v <- get1
>                              return (NoteOff channel firstData v)
>         ItsaNoteOn     -> do v <- get1
>                              case v of
>                                0 -> return (NoteOff channel firstData 0)
>                                _ -> return (NoteOn  channel firstData v)
>         ItsaPolyAfter  -> do v <- get1
>                              return (PolyAfter channel firstData v)
>         ItsaControl    -> do v <- get1
>                              return (Control channel firstData v)
>         ItsaProgChange -> return (ProgChange channel firstData)
>         ItsaMonoAfter  -> return (MonoAfter channel firstData)
>         ItsaPitchBend  -> do msb <- get1
>                              return (PitchBend channel (firstData+256*msb))
>         AtBeginning    -> error "AtBeginning"
```

Parse a MetaEvent.

```
> metaEvent :: Int -> Int -> MidiReader MetaEvent
```

```
> metaEvent   0 _    = do x <- get2;      return (SequenceNum x)
> metaEvent   1 size = do s <- getN size; return (TextEvent   s)
> metaEvent   2 size = do s <- getN size; return (Copyright   s)
> metaEvent   3 size = do s <- getN size; return (TrackName   s)
> metaEvent   4 size = do s <- getN size; return (InstrName   s)
> metaEvent   5 size = do s <- getN size; return (Lyric       s)
> metaEvent   6 size = do s <- getN size; return (Marker      s)
> metaEvent   7 size = do s <- getN size; return (CuePoint    s)
>
> metaEvent  32 _    = do c <- get1;   return (MIDIPrefix  c)
> metaEvent  47 _    =                 return (EndOfTrack   )
> metaEvent  81 _    = do t <- get3;   return (SetTempo    t)
>
> metaEvent  84 _    = do {hrs    <- get1 ; mins <- get1 ; secs <- get1;
>                          frames <- get1 ; bits <- get1 ;
>                          return (SMPTEOffset hrs mins secs frames bits)}
>
> metaEvent  88 _    = do
>                        n <- get1
>                        d <- get1
>                        c <- get1
>                        b <- get1
>                        return (TimeSig n d c b)
>
> metaEvent  89 _    = do
>                        sf <- get1
>                        mi <- get1
>                        return (KeySig (toKeyName sf) (toMode mi))
>
> metaEvent 127 size = do
>                        contents <- getN size
>                        return (SequencerSpecific (map fromEnum contents))
>
> metaEvent _   size = do {s <- getN size; return (Unknown s)}
>
> toKeyName sf = [ KeyCf .. ] !! ((sf+7) `mod` 15)
```

Convert a track using delta time to one using absolute time.

```
> undelta :: Track -> Track
> undelta trk = undelta' trk 0
>   where
>     undelta' :: Track -> Int -> Track
```

```
>     undelta' []                     _ = []
>     undelta' ((MidiEvent dt e):es) t = (MidiEvent (dt+t) e):undelta' es (dt+t)
>     undelta' ((MetaEvent dt e):es) t = (MetaEvent (dt+t) e):undelta' es (dt+t)
>     undelta' (_:es)                t = undelta' es t
>
> toMode :: Int -> Mode
> toMode 0 = Major
> toMode 1 = Minor
```

`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.

```
> get1 :: MidiReader Int
> get1 = do
>          c <- getCh
>          return (fromEnum c)
>
> get2 :: MidiReader Int
> get2 = do
>          x1 <- get1
>          x2 <- get1
>          return (fromBytes [x1,x2])
>
```

```
> get3 :: MidiReader Int
> get3 = do
>          x1 <- get1
>    x2 <- get1
>    x3 <- get1
>          return (fromBytes [x1,x2,x3])
>
> get4 :: MidiReader Int
> get4 = do
>          x1 <- get1
>          x2 <- get1
>          x3 <- get1
>          x4 <- get1
>          return (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 = getVarAux 0
>   where getVarAux n = do
>                         digit <- get1
>                         if digit < 128              -- if it's the last byte
>                           then return ((bshiftl 7 n) + digit)
>                           else getVarAux ((bshiftl 7 n) + (bTrunc 7 digit))
```

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

```
> showMidiFile :: String -> IO ()
> showMidiFile file = do
>                       s <- readBinaryFile file
>                       putStr (showChunks s)
>
> showChunks :: String -> String
> showChunks mf = showMR chunks (unlinesS . map pp) (mf, (AtBeginning,0),-1) ""
>  where
>   pp :: (String, String, Status, Int) -> ShowS
>   pp ("MThd",contents,st,sz) =
```

```
>     showString "Header: " .
>     showMR header shows (contents,st,sz)
>   pp ("MTrk",contents,st,sz) =
>     showString "Track:\n" .
>     showMR plainTrack (unlinesS . map showTrackEvent) (contents,st,sz)
>   pp (ty,contents,_,_) =
>     showString "Chunk: " .
>     showString ty .
>     showString " " .
>     shows (map fromEnum contents) .
>     showString "\n"
>
> showTrackEvent :: (ElapsedTime, MidiFileEvent) -> ShowS
> showTrackEvent (t,e) =
>   rightS 10 (shows t) . showString " : " . showEvent e
>
> showEvent :: MidiFileEvent -> ShowS
> showEvent (Midi e) =
>   showString "MidiEvent " .
>   shows e
> showEvent (Meta 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 chunks 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)

```
> chunks :: MidiReader [(String, String, Status, Int)]
> chunks = zeroOrMore plainChunk
>
> plainChunk :: MidiReader (String, String, Status, Int)
> plainChunk = 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)
```

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

9 Reading Midi files

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

```
> module ReadMidi (readMidi, module Basics, module Performance,
>                  module MidiFile)
>   where
```

```
>
> import Ratio
> import Basics
> import Performance
> import MidiFile
> import HaskToMidi
> import GeneralMidi
```

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

```
> readMidi :: MidiFile -> (Music, Context, UserPatchMap)
> readMidi mf@(MidiFile _ d trks) =
>   let trk1        = removeEOTs $ format mf
>       upm         = makeUPM trks
>       (trk2,dur)  = getFirst trk1
>       trksrs      = getRest dur (newTracks dur (splitBy (isTempoChg) trk2))
>       read (t,r)  = Tempo r (readTrack (tDiv d) upm t)
>       m           = optimize $ line $ map read trksrs
>   in (m,
>       Context {cTime   = 0,
>                cPlayer = fancyPlayer,
>                cInst   = getInst upm,
>                cDur    = float dur / 250000,
>                cKey    = 0,
>                cVol    = 100},
>       upm)
```

Remove the `EndOfTrack`'s that end up in the middle of merged or sequenced tracks.

```
> removeEOTs :: Track -> Track
> removeEOTs ((MetaEvent _ EndOfTrack):es) = removeEOTs es
> removeEOTs (e:es) = e:(removeEOTs es)
> removeEOTs [] = []
```

Make one big track out of the individual tracks of a MidiFile, using different methods depending on the format of the MidiFile.

```
> format (MidiFile 0 _ [trk]) = trk
```

```
> format (MidiFile 1 _ trks)  = mergeTracks trks
> format (MidiFile 2 _ trks)  = seqTracks trks
> format _                    = error ("readMidi: unfamiliar format or file corrupt")
```

Used for a list of simultaneous tracks (format 1). It merges together the events of all the tracks, putting the events in order by their elapsed time. A generalization of this algorithm could be useful as a merge of an arbitrary number of lists.

```
> mergeTracks :: [Track] -> Track
> mergeTracks [[]]      = []
> mergeTracks ([]:trks) = mergeTracks trks
> mergeTracks trks      = let minTs = foldl1 minT trks
>                             minT []              trk             = trk
>                             minT trk             []              = trk
>                             minT (trk1@(e1:e1s)) (trk2@(e2:e2s)) =
>                               if getTime e1 <= getTime e2 then trk1 else trk2
>                             newList (trk@(e:es):trks) trk'
>                               | e == (head trk') = es:trks
>                               | otherwise        = trk:(newList trks trk')
>                             newList ([]:trks) trk' = []:newList trks trk'
>                             newList [[]]      _    = [[]]
>                         in
>                          (head (minTs)) : mergeTracks (newList trks (minTs))
```

Used for a list of sequential tracks (format 2). It puts the events of the tracks in order one after the other. About its auxiliary functions: `seqTracks'` is the same as seqTracks, only it keeps a running total of the time increment; `incTrack` increments the elapsed-time of each event in the given track; `incEvent` increments the elapsed-time of a single event.

```
> seqTracks :: [Track] -> Track
> seqTracks trks      = seqTracks' 0 trks
>   where
>     seqTracks' _ []           = []
>     seqTracks' _ [[]]         = []
>     seqTracks' n ([]:trks)    = seqTracks' n trks
>     seqTracks' n (trk:trks)   = a ++ seqTracks' (b+n) trks
>        where (a,b) = incTrack n trk
>     incTrack n []             = error ("incTrack: Empty list")
>     incTrack n [e]            = ([incEvent n e],getTime e)
>     incTrack n (e:es) = ((incEvent n e):a, b)
>       where (a,b) = incTrack n es
```

```
>     incEvent n (MidiEvent t x) = MidiEvent (t+n) x
>     incEvent n (MetaEvent t x) = MetaEvent (t+n) x
```

Get the elapsed time of an event.

```
> getTime :: MEvent -> ElapsedTime
> getTime (MidiEvent t _) = t
> getTime (MetaEvent t _) = t
```

Look through the given tracks, using Program Changes to make a UserPatchMap.

```
> makeUPM :: [Track] -> UserPatchMap
> makeUPM trks = removeDups $ map makeTriple $ concatMap searchPC trks
>   where makeTriple (ch,num) = let gmName = lookupInst genMidiMap num
>                                   iName = case ch of
>                                             9 -> "drums"
>                                             _ -> gmName
>                               in (iName, gmName, ch)
>         searchPC ((MidiEvent _ (ProgChange ch num)):es)    = (ch, num) : searchPC es
>         searchPC ((MidiEvent t (NoteOn _ _ _)):es) | t > 0 = []
>         searchPC (e:es)                                    = searchPC es
>         searchPC []                                        = []
>         searchInst ((MetaEvent _ (InstrName iName)):es)      = Just iName
>         searchInst ((MidiEvent t (NoteOn _ _ _)):es) | t > 0 = Nothing
>         searchInst (e:es)                                    = searchInst es
>         searchInst []                                        = Nothing
```

Remove consecutive duplicates from a list, used in this case to take out redundant elements in the UserPatchMap.

```
> removeDups :: Eq a => [a] -> [a]
> removeDups []        = []
> removeDups [x]       = [x]
> removeDups (x:y:xys) | x == y    = removeDups (y:xys)
>                      | otherwise = x : removeDups (y:xys)
```

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."
```

`getFirst` 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 x y m`), but a SetTempo at time 0 should set the default tempo for the entire piece, by translating to Context tempo. `getFirst` takes care of all events that occur at time 0 so that if any SetTempo appears at time 0, it can translate to Context even if it is not the very first event.

```
> getFirst :: Track -> (Track, Int)
> getFirst trk1 = getFirst' trk1 []
>  where
>   getFirst' :: Track -> Track -> (Track, Int)
>   getFirst' ((MetaEvent 0 (SetTempo tempo)):es) tAcc = (((reverse tAcc)++es), tempo)
>   getFirst' (e@(MetaEvent 0 _):es)              tAcc = getFirst' es (e:tAcc)
>   getFirst' (e@(MidiEvent 0 _):es)              tAcc = getFirst' es (e:tAcc)
>   getFirst' es                                  tAcc = (((reverse tAcc)++es), defST)
>   getFirst' []                                  _    = ([]                  , defST)
```

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 -> [Track] -> [(Track,Ratio Int)]
> getRest d (((MetaEvent _ (SetTempo tempo)):es):trks) = (es,r) : getRest d trks
>   where r = d % tempo
> getRest d (trk:trks) = (trk,1) : getRest d trks
> getRest _ []         = []
```

Get the first instrument from the UserPatchMap, to use as the default in the Context.

```
> getInst :: UserPatchMap -> String
> getInst ((iName, gmName, channel):xs) = iName
> getInst []                            = "piano"
```

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 test xs = splitBy' test xs []
>   where splitBy' :: (a -> Bool) -> [a] -> [a] -> [[a]]
>         splitBy' test (x1:x2:xs) acc =
>           if (test x2)
>           then (reverse (x1:acc)) : (splitBy' test (x2:xs) [])
>           else splitBy' test (x2:xs) (x1:acc)
>         splitBy' test [x] acc = [(reverse (x:acc))]
>         splitBy' _    []  _   = [[]]

> isTempoChg :: MEvent -> Bool
> isTempoChg (MetaEvent _ (SetTempo _)) = True
> isTempoChg _                          = False
```

readTrack is the heart of the readMidi operation. It reads a track that has been "processed" by newTracks, and returns the track as Music.

```
> readTrack :: Int -> UserPatchMap -> Track -> Music
> readTrack _     _   []
>     = Rest 0
> readTrack ticks upm (nOn @(MidiEvent t1 (NoteOn  ch p v))
>                     : nOff@(MidiEvent t2 (NoteOff _  _ _))
>                     : nOn2@(MidiEvent t3 (NoteOn  _  _ _))
>                     : es)
>   | (t2 < t1) || (t3 < t1) = error "readTrack: elapsed time values out of order"
>   | t2 == t3        = n :+: readTrack ticks upm (nOn2:es)
>   | t1 == t3        = n :=: readTrack ticks upm (nOn2:es)
>   | t3 < t2         = n :=: (Rest (diff ticks t1 t3) :+: readTrack ticks upm (nOn2:es
>   | t3 > t2         = n :+:  Rest (diff ticks t2 t3) :+: readTrack ticks upm (nOn2:es
>       where plainNote = Note (pitch p) (diff ticks t1 t2) (makeVol v)
>             n = if ch == 9 then Instr "drums"            plainNote
>                            else Instr (lookupUPM ch upm) plainNote
> readTrack ticks upm (nOn@ (MidiEvent _ (NoteOn  _ _ _))
>                     : nOff@(MidiEvent _ (NoteOff _ _ _))
>                     : x
>                     : es)
```

```
>   = readTrack ticks upm (x : nOn : nOff : es)
> readTrack ticks upm (MidiEvent t1 (NoteOn  ch p v)
>                     : MidiEvent t2 (NoteOff _  _ _)
>                     : es)
>   | t2 < t1   = error "readTrack: elapsed time values of last note out of order"
>   | otherwise = n :+: readTrack ticks upm es
>      where plainNote = Note (pitch p) (diff ticks t1 t2) (makeVol v)
>            n = if ch == 9 then Instr "drums"            plainNote
>                           else Instr (lookupUPM ch upm) plainNote
> readTrack ticks upm ((MidiEvent t (ProgChange ch num)):es)
>     = readTrack ticks (progChange ch num upm) es
> readTrack ticks upm (e:es)
>     = readTrack ticks upm es
```

Take the division in ticks and two time values and calculates the note duration (quarter note, eighth note, etc.) that expresses their difference.

```
> diff :: Int -> Int -> Int -> Dur
> diff ticks t1 t2 = ((t2 - t1) % ticks) / 8
```

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

```
> lookupUPM :: Int -> UserPatchMap -> String
> lookupUPM ch ((name,_,midiChan):xs)
>   | ch == midiChan = name
>   | otherwise      = lookupUPM ch xs
> lookupUPM ch  [] = error ("Invalid channel in user patch map")
```

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

```
> progChange :: Int -> Int -> UserPatchMap -> UserPatchMap
> progChange ch num ((_, _, midiChan):xs) | ch == midiChan =
>   let n = lookupInst genMidiMap num
>   in ((n,n,ch):xs)
> progChange ch num (x:xs) = x:(progChange ch num xs)
> progChange _  _   []     = []
> progChange _  _   _      = error "progChange: Uh oh."
```

Look up an instrument in the General Midi table given its program number.

```
> lookupInst :: GenMidiTable -> Int -> IName
> lookupInst ((inst,num):xs) n | (num == n) = inst
>                              | otherwise  = lookupInst xs n
```

Interpret a MidiFile velocity, creating a Music NoteAttribute. The MIDI specification calls for some sort of exponential scale, but for now it's just linear.

```
> makeVol :: Int -> [NoteAttribute]
> makeVol x = [Volume (float x)]
```

The newTracks function changes the order of the events in a list of sub-tracks 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.

```
> newTracks :: Int -> [Track] -> [Track]
> newTracks dur = newTracks' 0 dur
>   where
>     newTracks' :: Int -> Int -> [Track] -> [Track]
>     newTracks' _   _   []        = []
>     newTracks' _   _   [[]]      = [[]]
>     newTracks' stt stv ([]:trks) = [] : newTracks' stt stv trks
>     newTracks' _   _   ((e@(MetaEvent newStt (SetTempo newStv)):es):trks) =
>       let (trk':trks') = newTracks' newStt newStv (es:trks)
>       in ((e:trk'):trks')
>     newTracks' stt stv ((e1@(MidiEvent tno (NoteOn c p _)):es):trks) =
>       let (e2, leftover) = search (stt-tno) stv stv tno stt c p (es:trks) [[]]
>           (trk':trks') = newTracks' stt stv leftover
>       in ((e1:e2:trk'):trks')
>     newTracks' stt stv ((e:es):trks) = let (trk':trks') = newTracks' stt stv (es:trks
>                                        in ((e:trk'):trks')
>     newTracks' stt stv (trk:trks)    = trk:(newTracks' stt stv trks)
```

search takes a track that has been divided by tempo into sub-lists; it looks through the list of sub-tracks 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 it's in a different sub-track than its NoteOn, it puts it in the NoteOn's subtrack and calculates

what the new elapsed-time should be after one or more tempo changes. 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.

```
> search :: Int ->           -- time interval between NoteOn and most recent SetTempo,
>                            --   in terms of the tempo at the NoteOn
>           Int -> Int ->    -- SetTempo values: the one at the NoteOn and the most
>                            --   recent one
>           Int -> Int ->    -- elapsed times of the NoteOn and the most recent SetTemp
>           Int -> Int ->    -- channel and pitch of NoteOn (NoteOff must match)
>           [Track] ->       -- the tracks left to be searched
>           [Track] ->       -- accumulator: what's left after NoteOff is found
>           (MEvent,[Track]) -- the needed event and the remainder of the tracks
> search int ost nst tno stt c1 p1 ((e@(MidiEvent t (NoteOff c2 p2 v)):es):trks)
>                                                                  (aTrk:aTrks)
>   | c1 == c2 && p1 == p2
>   = ((MidiEvent eTime (NoteOff c2 p2 v)),
>      (reverse aTrks)++(((reverse aTrk)++es):trks))
>     where eTime = tno + int + round ((float (t-stt))*(float ost)/(float nst))
> search int ost nst tno stt c  p  ((e@(MetaEvent t (SetTempo st)):es):trks)
>                                                                  (aTrk:aTrks) =
>      search newInt ost st  tno t   c p (es:trks) ((e:aTrk):aTrks)
>        where newInt = int + round ((float (t-stt))*(float ost)/(float nst))
> search _   _   _   tno _   c  p  [[]] (aTrk:aTrks) =
>      ((MidiEvent tno (NoteOff c p 0)), aTrks++[reverse aTrk])
> search _   _   _   tno _   c  p  []   (aTrk:aTrks) =
>      ((MidiEvent tno (NoteOff c p 0)), aTrks++[reverse aTrk])
> search int ost nst tno stt c  p  ((e:es):trks) (aTrk:aTrks) =
>      search int    ost nst tno stt c p (es:trks) ((e:aTrk):aTrks)
> search int ost nst tno stt c  p  ([]    :trks) (aTrk:aTrks) =
>      search int    ost nst tno stt c p trks      ([]:(reverse aTrk):aTrks)
> search _   _   _   _   _   _  _  []            _           =
>                                 error "newTracks: search fell thru"
```

Music objects that come out of `readMidi` almost always contain redundancies, like rests of zero duration and redundant instrument specifications. `optimize` reduces the redundancy to make a Music file less cluttered and more efficient to use.

```
> optimize, optRests, optTempo, optInstr, optVol :: Music -> Music
> optimize m = optTempo $ optVol $ optInstr $ optRests m
```

Remove rests of zero duration.

```
> optRests (m :+: Rest 0) = optRests m
> optRests (m :=: Rest 0) = optRests m
> optRests (Rest 0 :+: m) = optRests m
> optRests (Rest 0 :=: m) = optRests m
> optRests (m1 :+: m2) = optRests m1 :+: optRests m2
> optRests (m1 :=: m2) = optRests m1 :=: optRests m2
> optRests (Tempo a m)   = Tempo a  $ optRests m
> optRests (Trans a m)   = Trans a  $ optRests m
> optRests (Instr a m)   = Instr a  $ optRests m
> optRests (Player a m)  = Player a $ optRests m
> optRests (Phrase a m)  = Phrase a $ optRests m
> optRests x             = x
```

Remove redundant `Tempo`'s.

```
> optTempo (Tempo 1 m) = optTempo m
> optTempo (Tempo a (Tempo b m)) = Tempo (a*b) $ optTempo m
> optTempo (m1 :+: m2) = optTempo m1 :+: optTempo m2
> optTempo (m1 :=: m2) = optTempo m1 :=: optTempo m2
> optTempo (Tempo a m)   = Tempo a   $ optTempo m
> optTempo (Trans a m)   = Trans a   $ optTempo m
> optTempo (Instr a m)   = Instr a   $ optTempo m
> optTempo (Player a m)  = Player a  $ optTempo m
> optTempo (Phrase a m)  = Phrase a  $ optTempo m
> optTempo x             = x
```

Remove unnecessary `Instr`'s.

```
> optInstr m1 = let (m2,changed) = optInstr' m1
>               in if changed then optInstr m2
>                             else m2
>   where
>     optInstr' :: Music -> (Music, Bool)
>     optInstr' ((m1 :+: m2) :+: m3) = (m1 :+: m2 :+: m3, True)
>     optInstr' ((m1 :=: m2) :=: m3) = (m1 :=: m2 :=: m3, True)
>     optInstr' (Instr a (Instr b m)) = (Instr a m, True)
>     optInstr' (Instr a m1 :+: Instr b m2) | a == b = (Instr a (m1 :+: m2), True)
>     optInstr' (Instr a m1 :=: Instr b m2) | a == b = (Instr a (m1 :=: m2), True)
>     optInstr' (Instr a m1 :+: Instr b m2 :+: x)
>       | a == b = (Instr a (m1 :+: m2) :+: x, True)
```

```
>     optInstr' (Instr a m1 :=: Instr b m2 :=: x)
>       | a == b = (Instr a (m1 :=: m2) :=: x, True)
>     optInstr' (Instr a m :+: Rest r) = (Instr a (m :+: Rest r), True)
>     optInstr' (Instr a m :=: Rest r) = (Instr a (m :=: Rest r), True)
>     optInstr' (Rest r :+: Instr a m) = (Instr a (Rest r :+: m), True)
>     optInstr' (Rest r :=: Instr a m) = (Instr a (Rest r :=: m), True)
>     optInstr' (m1 :+: m2) = let (m3,c3) = optInstr' m1
>                                 (m4,c4) = optInstr' m2
>                             in (m3 :+: m4, c3 || c4)
>     optInstr' (m1 :=: m2) = let (m3,c3) = optInstr' m1
>                                 (m4,c4) = optInstr' m2
>                             in (m3 :=: m4, c3 || c4)
>     optInstr' (Tempo a m)   = (Tempo a   $ optInstr m, False)
>     optInstr' (Trans a m)   = (Trans a   $ optInstr m, False)
>     optInstr' (Instr a m)   = (Instr a   $ optInstr m, False)
>     optInstr' (Player a m)  = (Player a  $ optInstr m, False)
>     optInstr' (Phrase a m)  = (Phrase a  $ optInstr m, False)
>     optInstr' x             = (x, False)
```

Change repeated Volume Note Attributes to Phrase Attributes.

```
> optVol m1 = let (m2,changed) = optVol' m1
>             in if changed then optVol m2
>                           else m2
> optVol' ((m1 :+: m2) :+: m3) = (m1 :+: m2 :+: m3, True)
> optVol' ((m1 :=: m2) :=: m3) = (m1 :=: m2 :=: m3, True)
> optVol' (Phrase p1 (Phrase p2 m)) | p1 == p2  = (Phrase p1 m, True)
>                                   | otherwise = (Phrase (p1 ++ p2) m, True)
> optVol' (Note p1 d1 [Volume v1] :+: Note p2 d2 [Volume v2]) | v1 == v2 =
>     (Phrase [Dyn (Loudness v1)] (Note p1 d1 [] :+: Note p2 d2 []), True)
> optVol' (Note p1 d1 [Volume v1] :=: Note p2 d2 [Volume v2]) | v1 == v2 =
>     (Phrase [Dyn (Loudness v1)] (Note p1 d1 [] :=: Note p2 d2 []), True)
> optVol' (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)
> optVol' (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)
> optVol' (Phrase p1 m1 :+: Phrase p2 m2) | p1 == p2 = (Phrase p1 (m1 :+: m2), True)
> optVol' (Phrase p1 m1 :=: Phrase p2 m2) | p1 == p2 = (Phrase p1 (m1 :=: m2), True)
> optVol' (Phrase p1 m1 :+: Phrase p2 m2 :+: x) | p1 == p2 =
>     (Phrase p1 (m1 :+: m2 :+: x),True)
> optVol' (Phrase p1 m1 :=: Phrase p2 m2 :=: x) | p1 == p2 =
>     (Phrase p1 (m1 :=: m2 :=: x),True)
> optVol' (Phrase phr@[Dyn (Loudness l)] m :+: Note p d [Volume v]) | l == v =
```

```
>     (Phrase phr (m :+: Note p d []), True)
> optVol' (Phrase phr@[Dyn (Loudness l)] m :=: Note p d [Volume v]) | l == v =
>     (Phrase phr (m :=: Note p d []), True)
> optVol' (Phrase phr@[Dyn (Loudness l)] m :+: Note p d [Volume v] :+: x) | l == v =
>     (Phrase phr (m :+: Note p d []) :+: x,True)
> optVol' (Phrase phr@[Dyn (Loudness l)] m :=: Note p d [Volume v] :=: x) | l == v =
>     (Phrase phr (m :=: Note p d []) :=: x,True)
> optVol' (Note p d [Volume v] :+: Phrase phr@[Dyn (Loudness l)] m) | l == v =
>     (Phrase phr (Note p d [] :+: m),True)
> optVol' (Note p d [Volume v] :=: Phrase phr@[Dyn (Loudness l)] m) | l == v =
>     (Phrase phr (Note p d [] :=: m),True)
> optVol' (Note p d [Volume v] :+: Phrase phr@[Dyn (Loudness l)] m :+: x) | l == v =
>     ((Phrase phr (Note p d [] :+: m)) :+: x, True)
> optVol' (Note p d [Volume v] :=: Phrase phr@[Dyn (Loudness l)] m :=: x) | l == v =
>     ((Phrase phr (Note p d [] :=: m)) :=: x, True)
> optVol' (Rest r :+: Note p d [Volume v]) =
>     (Phrase [Dyn (Loudness v)] (Rest r :+: Note p d []),True)
> optVol' (Rest r :=: Note p d [Volume v]) =
>     (Phrase [Dyn (Loudness v)] (Rest r :=: Note p d []),True)
> optVol' (Note p d [Volume v] :+: Rest r) =
>     (Phrase [Dyn (Loudness v)] (Note p d [] :+: Rest r),True)
> optVol' (Note p d [Volume v] :=: Rest r) =
>     (Phrase [Dyn (Loudness v)] (Note p d [] :=: Rest r),True)
> optVol' (Rest r :+: Phrase p m) = (Phrase p (Rest r :+: m), True)
> optVol' (Rest r :=: Phrase p m) = (Phrase p (Rest r :=: m), True)
> optVol' (Phrase p m :+: Rest r) = (Phrase p (m :+: Rest r), True)
> optVol' (Phrase p m :=: Rest r) = (Phrase p (m :=: Rest r), True)
> optVol' (m1 :+: m2) = let (m3,c3) = optVol' m1
>                           (m4,c4) = optVol' m2
>                       in (m3 :+: m4, c3 || c4)
> optVol' (m1 :=: m2) = let (m3,c3) = optVol' m1
>                           (m4,c4) = optVol' m2
>                       in (m3 :=: m4, c3 || c4)
> optVol' (Tempo a m)   = (Tempo a   $ optVol m, False)
> optVol' (Trans a m)   = (Trans a   $ optVol m, False)
> optVol' (Instr a m)   = (Instr a   $ optVol m, False)
> optVol' (Player a m)  = (Player a  $ optVol m, False)
> optVol' (Phrase a m)  = (Phrase a  $ optVol m, False)
> optVol' x             = (x, False)
```

10 Representing Chords

Earlier I described how to represent chords as values of type `Music`. 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.[3]

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:

```
> type Chord = [AbsPitch]
```

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:

```
> pitToInt :: Chord -> Chord
> pitToInt ch = aux ch
>    where aux (n1:n2:ns) = (n2-n1) : aux (n2:ns)
```

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

```
>          aux [n]        = [head ch - n]
>
> intToPit :: Chord -> Chord
> intToPit ch = 0 : aux 0 ch
>    where aux p [n]    = []
>          aux p (n:ns) = n' : aux n' ns  where n' = p+n
```

Exercise 9 *Show that* `pitToInt` *and* `intToPit` *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:*

```
intToPit (pitToInt ch1) = ch1
pitToInt (intToPit 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:

```
> pitInvert (p1:p2:ps) = 0 : map (subtract p2) ps ++ [12-p2]
```

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:

```
> intInvert = pitToInt . pitInvert . intToPit
```

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:

```
> samePitChord ch1 ch2 =
>  let invs = take (length ch1) (iterate pitInvert ch1)
>  in  ch2 `elem` invs
>
> sameIntChord ch1 ch2 =
>  let invs = take (length ch1) (iterate intInvert ch1)
>  in  ch2 `elem` invs
```

For example, `samePitChord [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]`).

11 Equivalence of Literal Performances

A *literal performance* is one in which no aesthetic interpretation is given to a musical object. The function `perform` 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:

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

```
(∀imap,c) perform imap c m1 = perform 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."

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

Axiom 1 *For any* `r1`, `r2`, `r3`, `r4`, *and* `m`*:*

```
Tempo r1 r2 (Tempo r3 r4 m)  ≡  Tempo (r1*r3) (r2*r4) m
```

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

Proof:

```
perform dt (Tempo r1 r2 (Tempo r3 r4 m))
= perform (r2*dt/r1) (Tempo r3 r4 m)       -- unfolding perform
= perform (r4*(r2*dt/r1)/r3) m             -- unfolding perform
= perform ((r2*r4)*dt/(r1*r3)) m           -- simple arithmetic
= perform dt (Tempo (r1*r3) (r2*r4) m)     -- folding perform
```

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)`):

Axiom 2 *For any* `r1`, `r2`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 (m1 :+: m2)  ≡  Tempo r1 r2 m1 :+: Tempo r1 r2 m2
```

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

Proof:

```
perform (t,dt) (Tempo r1 r2 (m1 :+: m2))
= perform (t,r2*dt/r1) (m1 :+: m2)                      -- unfolding perform
= perform (t,r2*dt/r1) m1 ++ perform (t',r2*dt/r1) m2   -- unfolding perform
= perform (t,dt) (Tempo r1 r2 m1) ++
          perform (t',dt) (Tempo r1 r2 m2)              -- folding perform
= perform (t,dt) (Tempo r1 r2 m1) ++
          perform (t'',dt) (Tempo r1 r2 m2)             -- folding dur
= perform (t,dt) (Tempo r1 r2 m1 :+: Tempo r1 r2 m2)    -- folding perform
where t'  = t + (dur m1)*r2*dt/r1
      t'' = t + (dur (Tempo r1 r2 m1))*dt
```

An even simpler axiom is given by:

Axiom 3 *For any* `r` *and* `m`*:*

```
Tempo r r m  ≡  m
```

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

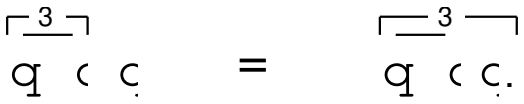


Figure 9: Equivalent Phrases

Proof:

```
perform (t,dt) (Tempo r r m)
= perform (t,r*dt/r) m                          -- unfolding perform
= perform (t,dt) m                              -- simple arithmetic
```

Note that the above proofs, being used to establish axioms, all involve the definition of `perform`. 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:

Theorem 1 *For any* `r1`, `r2`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 m1 :+: m2  ≡  Tempo r1 r2 (m1 :+: Tempo r2 r1 m2)
```

Proof:

```
Tempo r1 r2 (m1 :+: Tempo r2 r1 m2)
= Tempo r1 r2 m1 :+: Tempo r1 r2 (Tempo r2 r1 m2)     -- by Axiom 1
= Tempo r1 r2 m1 :+: Tempo (r1*r2) (r2*r1) m2         -- by Axiom 2
= Tempo r1 r2 m1 :+: Tempo (r1*r2) (r1*r2) m2         -- simple arithmetic
= Tempo r1 r2 m1 :+: m2                               -- by Axiom 3
```

For example, this fact justifies the equivalence of the two phrases shown in Figure 9.

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).

Axiom 4 `Tempo` *is* multiplicative *and* `Transpose` *is* additive*. That is, for any* `r1`, `r2`, `r3`, `r4`, `p`, *and* `m`*:*

```
Tempo r1 r2 (Tempo r3 r4 m)  ≡  Tempo (r1*r3) (r2*r4) m
      Trans p1 (Trans p2 m)  ≡  Trans (p1+p2) m
```

Axiom 5 *Function composition is* commutative *with respect to both tempo scaling and transposition. That is, for any* `r1`, `r2`, `r3`, `r4`, `p1` *and* `p2`*:*

```
Tempo r1 r2 . Tempo r3 r4  ≡  Tempo r3 r4 . Tempo r1 r2
     Trans p1 . Trans p2   ≡  Trans p2 . Trans p1
  Tempo r1 r2 . Trans p1   ≡  Trans p1 . Tempo r1 r2
```

Axiom 6 *Tempo scaling and transposition are* distributive *over both sequential and parallel composition. That is, for any* `r1`, `r2`, `p`, `m1`, *and* `m2`*:*

```
Tempo r1 r2 (m1 :+: m2)  ≡  Tempo r1 r2 m1 :+: Tempo r1 r2 m2
Tempo r1 r2 (m1 :=: m2)  ≡  Tempo r1 r2 m1 :=: Tempo r1 r2 m2
     Trans p (m1 :+: m2)  ≡  Trans p m1 :+: Trans p m2
     Trans p (m1 :=: m2)  ≡  Trans p m1 :=: Trans p m2
```

Axiom 7 *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
```

Axiom 8 *Parallel composition is* commutative. *That is, for any* `m0` *and* `m1`*:*

```
m0 :=: m1  ≡  m1 :=: m0
```

Axiom 9 `Rest 0` *is a* unit *for* `Tempo` *and* `Trans`*, and a* zero *for sequential and parallel composition. That is, for any* `r1`, `r2`, `p`, *and* `m`*:*

```
  Tempo r1 r2 (Rest 0)  ≡  Rest 0
    Trans p (Rest 0)  ≡  Rest 0
m :+: Rest 0  ≡  m  ≡  Rest 0 :+: m
m :=: Rest 0  ≡  m  ≡  Rest 0 :=: m
```

Exercise 10 *Establish the validity of each of the above axioms.*

12 Haskore Support for CSound

```
> module CSound where

> import Performance
> import IO
> import List (find, nub)
```

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

```
Reading file "CSound.lhs":
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.

12.1 The Score File

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

```
> type Score = [ScoreStmt]
```

The `ScoreStmt` 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 ScoreStmt = CSTempo Bpm
>                | CSNote Inst StartTime Duration Pch Vlm [Pfield]
>                | CSTable Tbl CreatTime TblSize Normalize GenRoutine
>      deriving Show
>
> type Bpm       = Int
> type Inst      = Int
> type StartTime = Float
> type Duration  = Float
> type Pch       = Float
> type Vlm       = Float
> type Pfield    = Float
> type Tbl       = Int
> type CreatTime = Float
> type TblSize   = Int
> type Normalize = Bool
```

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

12.1.1 Function Tables

Each function table must have a unique integer ID (`Tbl`), 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 12.2). 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 `TblSize` 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
```

`flattenTuples2` flattens a list of pairs into a list. Similarly, `flattenTuples3` flattens a list of 3-tuples into a list, and so on.

```
> flattenTuples2 :: [(a,a)]     -> [a]
> flattenTuples3 :: [(a,a,a)]   -> [a]
> flattenTuples4 :: [(a,a,a,a)] -> [a]
>
> flattenTuples2 []           = []
> flattenTuples2 ((x,y) : xs) = x : y : flattenTuples2 xs
>
> flattenTuples3 []             = []
> flattenTuples3 ((x,y,z) : xs) = x : y : z : flattenTuples3 xs
>
> flattenTuples4 []                   = []
> flattenTuples4 ((x, y, z, w) : xs) = x : y : z : w : flattenTuples4 xs
```

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 [float n, float 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 [float 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 :: Tbl -> ScoreStmt
>
> simpleSine n = CSTable n 0 8192 True
>                       (compSine1 [1])
> square     n = CSTable n 0 1024 True
>                       (lineSeg1 1 [(128, 1), (0, -1), (128, -1)])
> sawtooth   n = CSTable n 0 1024 True
>                       (lineSeg1 0 [(128, 1), (0, -1), (128, 0)])
> triangle   n = CSTable n 0 1024 True
>                       (lineSeg1 0 [(64, 1), (128, -1), (64, 0)])
> whiteNoise n = CSTable n 0 1024 True
>                       (randomTable uniform)
```

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

```
> compSine :: Tbl -> [PStrength] -> ScoreStmt
> compSine n s = CSTable 6 0 8192 True (compSine1 s)
```

12.1.2 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)]
> type Name    = String
```

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 =
>   case (find (\(n,_) -> n==nm) nmap) of
>     Nothing    -> Nothing
>     Just (n,i) -> Just i
```

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

12.1.3 Converting Haskore Music to a CSound Score File

To convert a `Music` value into a CSound score file, we need to:

1. Convert the `Music` value to a `Performance`.
2. Convert the `Performance` value to a `Score`.
3. Write the `Score` 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:

```
perfToScore :: NameMap -> Performance -> Score
writeScore  :: Score -> IO ()
```

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

```
> musicToCSound :: NameMap -> PMap -> Tables -> Context -> Music -> IO ()
> musicToCSound nmap pmap tables cont m =
>   writeScore (tables ++ perfToScore nmap (perform pmap cont m))
>
> type Tables = Score
```

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

From Performance to Score The translation between performance `Events` and score `CSNotes` is straightforward, the only tricky parts being:

- The unit of time in a `Performance` is the second, whereas in a `Score` 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.
- In a `Performance`, pitch is represented as an `AbsPitch` value, an integer denoting the absolute position of the pitch in semitones (MIDI uses the same representation). CSound, however, uses different pitch representations, one of them being *octave point pitch class*, or "pch." Using pch, a pitch is represented by a decimal number whose integer part represents the octave, and decimal part represents the semitone within the octave (thus 0.00 is the lowest C, 0.11 the lowest B, 8.00 is middle C, etc.). The function `absToPch` defined below performs the required conversion for us.

```
> perfToScore :: NameMap -> Performance -> Score
> perfToScore _ [] = []
> perfToScore nmap (Event t i p d v pfs : evs) =
>   case (getId nmap i) of
>     Nothing  -> error ("perfToScore: instrument " ++ i ++ " is unknown")
>     Just num -> CSNote num t d (absToPch p) v pfs : perfToScore nmap evs
>
> absToPch :: AbsPitch -> Pch
> absToPch ap = float (ap `quot` 12) + (float (ap `mod` 12) / 100.0)
```

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:

```
> writeScore :: Score -> IO ()
> writeScore s = do putStr "\nName your score file "
>                   putStr "(.sco extension will be added): "
>                   name   <- getLine
>                   h      <- openFile (name ++ ".sco") WriteMode
>                   printScore h s
>                   hClose h
```

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 `printScore`, 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.

```
> printScore :: Handle -> Score -> IO ()
> printScore h []       = hPutStr h "e\n"        -- end of score
> printScore h (s : ss) = do printStatement h s
>                            printScore h ss
```

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 `printStatement` function:

```
> printStatement :: Handle -> ScoreStmt -> IO ()
> printStatement h (CSTempo t) =
>   hPutStr h ("t 0 " ++ show t ++ "\n")
> printStatement h (CSNote i st d p v pfs) =
>   hPutStr h
>     ("i " ++ show i ++ " " ++ show st ++ " " ++ show d ++ " " ++
>              show p ++ " " ++ show v ++ listToString pfs ++ "\n")
> printStatement h (CSTable t ct s n gr)   =
>   hPutStr h
>     ("f " ++ show t ++ " " ++ show ct ++ " " ++ show s ++
>       (if n then " " else " -") ++ showGenRoutine gr ++ "\n")
>   where showGenRoutine (SoundFile nm st cn) =
>                        "1 " ++ nm ++ " " ++ show st ++ " 0 " ++ show cn
>         showGenRoutine (GenRoutine gn gas) =
>                        show gn ++ listToString gas
```

12.2 The Orchestra File

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 Orchestra = (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, OrcExp)
```

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.

12.2.1 Orchestra Expressions

The data type `OrcExp` 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 `OrcExp`. 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 `OrcExp` data type declaration is shown in Figure 10. In what follows, we describe each of the various constructors in turn.

```
> data OrcExp = Const Float
>             | Pfield Int
>
>             | Plus    OrcExp OrcExp
>             | Minus   OrcExp OrcExp
>             | Times   OrcExp OrcExp
>             | Divide  OrcExp OrcExp
>             | Power   OrcExp OrcExp
>             | Modulo  OrcExp OrcExp
>
>             | Int     OrcExp
>             | Frac    OrcExp
>             | Neg     OrcExp
>             | Abs     OrcExp
>             | Sqrt    OrcExp
>             | Sin     OrcExp
>             | Cos     OrcExp
>             | Exp     OrcExp
>             | Log     OrcExp
>
>             | AmpToDb OrcExp
>             | DbToAmp OrcExp
>             | PchToHz OrcExp
>             | HzToPch OrcExp
>
>             | GreaterThan   OrcExp OrcExp OrcExp OrcExp
>             | LessThan      OrcExp OrcExp OrcExp OrcExp
>             | GreaterOrEqTo OrcExp OrcExp OrcExp OrcExp
>             | LessOrEqTo    OrcExp OrcExp OrcExp OrcExp
>             | Equals        OrcExp OrcExp OrcExp OrcExp
>             | NotEquals     OrcExp OrcExp OrcExp OrcExp
>
>             | MonoOut       OrcExp
>             | LeftOut       OrcExp
>             | RightOut      OrcExp
>             | StereoOut     OrcExp OrcExp
>             | FrontLeftOut  OrcExp
>             | FrontRightOut OrcExp
>             | RearRightOut  OrcExp
>             | RearLeftOut   OrcExp
>             | QuadOut       OrcExp OrcExp OrcExp OrcExp
>
>             | 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 LoopTime
>      deriving (Show, Eq)
```


Figure 10: The OrcExp Data Type

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 :: OrcExp
> noteDur = Pfield 3
> notePit = Pfield 4
> noteVol = Pfield 5
```

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 :: OrcExp
> p1 = Pfield 1
> p2 = Pfield 2
> p3 = Pfield 3
> p4 = Pfield 4
> p5 = Pfield 5
> p6 = Pfield 6
> p7 = Pfield 7
> p8 = Pfield 8
> p9 = Pfield 9
```

Arithmetic and Transcendental Functions Arithmetic expressions are represented by the constructors `Plus`, `Minus`, `Times`, `Divide`, `Power`, and `Modulo`, each taking two `OrcExps` 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 `OrcExp` an instance of certain numeric type classes, thus providing more conventional names for the various operations.

```
> instance Num OrcExp where
```

```
>   (+)    = Plus
>   (-)    = Minus
>   (*)    = Times
>   negate = Neg
>   abs    = Abs
>   fromInteger i = Const (fromInteger i)
>   -- omitted: signum

> instance Fractional OrcExp where
>   (/) = Divide
>   fromRational x = Const (fromRational x)

> instance Floating OrcExp where
>  exp   = Exp
>  log   = Log
>  sqrt  = Sqrt
>  (**)  = Power
>  sin   = Sin
>  cos   = Cos
>  pi    = Const pi
>  tan x = Sin x / Cos x
>  -- omitted: asin, acos, atan    :: a -> a
>  --          sinh, cosh, tanh    :: a -> a
>  --          asinh, acosh, atanh :: a -> a
```

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 `CSNote` 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 `OrcExp` also includes comparison constructors `GreaterThan`, `LessThan`, `GreaterOrEqTo`, `LessOrEqTo`, `Equals`, and `NotEquals`. Each takes four `OrcExp` 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.[4]

[4]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

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 `OrcExp` arguments, `QuadOut` takes four, and the rest take one.

The top-level of an instrument's `OrcExp` (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 `OrcExp` 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.

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

```
> type Start       = OrcExp
> type Durn        = OrcExp
> type Finish      = OrcExp
> type Sig         = OrcExp
> type RTime       = OrcExp
> type DTime       = OrcExp
> type RShape      = OrcExp
> type SAttn       = OrcExp
> type DAttn       = OrcExp
> type Steep       = OrcExp
```

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.

```
> type Freq        = OrcExp
> type InitPhase   = OrcExp
> type Index       = OrcExp
> type Table       = OrcExp
> type IndexMode   = OrcExp
> type Amp         = OrcExp
> type CarFreq     = OrcExp
> type ModFreq     = OrcExp
> type ModIndex    = OrcExp
> type HoldHz      = OrcExp
> type NumHarms    = OrcExp
> type LoHarm      = OrcExp
> type Multiplier  = OrcExp
> type DecayMethod = OrcExp
> type DecArg1     = OrcExp
> type DecArg2     = OrcExp
> type MaxDel      = OrcExp
> type AudioSig    = OrcExp
> type TapTime     = OrcExp
> type DelLine     = OrcExp
> type RevTime     = OrcExp
> type LoopTime    = OrcExp
```

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.[5] `rtime` and `dtime` are the rise time and decay time, respectively (in seconds), and `durn` is the overall duration. `rshape` is the identifier integer

[5] 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.

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 :: OrcExp
> 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:

```
osc = let index = Phasor AR (PchToHz notePit) 0.0
      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 `OrcExp` coupled with an instrument ID number (say, 1) produces a complete instrument block:

```
i1 = (1, MonoOut osc)
```

   Adding a suitable `Header` would then give us a complete, though somewhat sparse, `Orchestra` 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:

```
osc' = OscI AR 1 (PchToHz notePit) table
```

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 12.1.1). `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 :: OrcExp
> 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.

12.2.2 Converting Orchestra Values to Orchestra Files

We must now convert the `OrcExp` 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 11, and creates an entry in the list for every signal generating, modifying, or outputting constructor. It uses the following auxiliary functions:

```
> mkListAll :: [OrcExp] -> [(EvalRate, OrcExp)]
> mkListAll = foldr (++) [] . map mkList
>
> addNames :: [(EvalRate,OrcExp)] -> [(Name,OrcExp)]
> addNames ls = zipWith counter ls [1..]
>                 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 `OrcExp` into a list of proper name / `OrcExp` 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.)

```
> processOrcExp :: OrcExp -> [(Name, OrcExp)]
> processOrcExp = 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, OrcExp)] -> [(EvalRate, OrcExp)]
> procDelay []                              = []
> procDelay (x@(AR, d@(Delay _ _)) : xs) = [x] ++ procTaps d xs ++ procDelay xs
> procDelay (x : xs)                        = x : procDelay xs
>
> procTaps :: OrcExp -> [(EvalRate, OrcExp)] -> [(EvalRate, OrcExp)]
> procTaps d@(Delay t sig) []           = [(AR, DelayW sig)]
```

```
> mkList :: OrcExp -> [(EvalRate, OrcExp)]
> mkList (Const _)                  = []
> mkList (Pfield _)                 = []
> mkList (x1 `Plus` x2)             = mkListAll [x1,x2]
> mkList (x1 `Minus` x2)            = mkListAll [x1,x2]
> mkList (x1 `Times` x2)            = mkListAll [x1,x2]
> mkList (x1 `Divide` x2)           = mkListAll [x1,x2]
> mkList (x1 `Power` x2)            = mkListAll [x1,x2]
> mkList (x1 `Modulo` x2)           = mkListAll [x1,x2]
> mkList (Int x)                    = mkList x
> mkList (Frac x)                   = mkList x
> mkList (Abs x)                    = mkList x
> mkList (Neg x)                    = mkList x
> mkList (Sqrt x)                   = mkList x
> mkList (Sin x)                    = mkList x
> mkList (Cos x)                    = mkList x
> mkList (Exp x)                    = mkList x
> mkList (Log x)                    = mkList x
> mkList (AmpToDb x)                = mkList x
> mkList (DbToAmp x)                = mkList x
> mkList (PchToHz x)                = mkList x
> mkList (HzToPch x)                = mkList x
> mkList (GreaterThan   x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
> mkList (LessThan      x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
> mkList (GreaterOrEqTo x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
> mkList (LessOrEqTo    x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
> mkList (Equals        x1 x2 x3 x4) = mkListAll [x1,x2,x3,x4]
> mkList (NotEquals     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 x1 x2)           = (AR,oe) : mkList x2
> mkList oe@(DelTapI x1 x2)          = (AR,oe) : mkList x2
> mkList oe@(DelayW x)               = 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 x3)        = (AR,oe) : mkListAll [x1,x2,x3]
```

Figure 11: The `mkList` Function

```
> 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 (x : xs)                   = procTaps d xs
```

The functions that follow are used to write the orchestra file. `writeOrchestra` is similar to `writeScore`: it asks the user for a file name, opens the file, writes the given orchestra value to the file, and then closes the file.

```
> writeOrchestra :: Orchestra -> IO ()
> writeOrchestra orch = do putStr "\nName your orchestra file "
>                          putStr "(.orc extension will be added): "
>                          name <- getLine
>                          h    <- openFile (name ++ ".orc") WriteMode
>                          writeOrc h orch
>                          hClose h
```

`writeOrc` splits the task of writing the orchestra into two parts: writing the header and writing the instrument blocks

```
> writeOrc :: Handle -> Orchestra -> IO ()
> writeOrc h (hdr,ibs) = do writeHeader  h hdr
>                           writeIBlocks h ibs
```

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

```
> writeHeader :: Handle -> Header -> IO ()
> writeHeader h (a,k,nc) =
>   hPutStrLn h (  "sr     = " ++ show a ++
>                  "\nkr     = " ++ show k ++
>                  "\nksmps  = " ++ show (float a / float k) ++
>                  "\nnchnls = " ++ show nc)
```

`writeIBlocks` writes each instrument block using the function `writeIBlock`:

```
> writeIBlocks :: Handle -> [InstBlock] -> IO ()
> writeIBlocks h = mapM_ (writeIBlock h)
```

`writeIBlock` writes a single instrument block.

```
> writeIBlock :: Handle -> InstBlock -> IO ()
> writeIBlock h (num,ox) =
>   do hPutStrLn    h ("\ninstr " ++ show num)
>      writeOrcExps h (processOrcExp ox ++ [("",ox)])
>      hPutStrLn    h "endin"
```

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

12.3 An Orchestra Example

Figure 14 shows a typical CSound orchestra file. Figure 15 shows how this same functionality would be achieved in Haskore using an `Orchestra` value. Finally, Figure 16 shows the result of applying `writeOrchestra` to `orc1` shown in Figure 15. Figures 14 and 16 should be compared: you will note that except for name changes, they are the same, as they should be.

13 Related and Future Research

Many proposals have been put forth for programming languages targeted for computer music composition [Dan89, Sch83, Col84, AK92, DFV92, HS92, CR84, OFLB94], so many in fact that it would be difficult to describe them all here. None of them (perhaps surprisingly) are based on a *pure* functional language, with one exception: the recent work done by Orlarey et al. at GRAME [OFLB94], which uses a pure lambda calculus approach to music description, and bears some resemblance to our effort. There are some other related approaches based on variants of Lisp, most notably Dannenberg's *Fugue* language [DFV92], in which operators similar to ours can be found but where the emphasis is more on instrument synthesis rather than note-oriented composition. Fugue also highlights the utility of lazy evaluation in certain contexts, but extra effort is needed to make this work in Lisp, whereas in a non-strict language such as Haskell it essentially comes "for free." Other efforts based on Lisp utilize Lisp primarily as a convenient vehicle for "embedded language design," and the applicative nature of Lisp is not exploited

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

Figure 12: The Function `writeOrcExp`

```
> showExp :: [(Name, OrcExp)] -> OrcExp -> String
> showExp _ (Const x)         = show x
> showExp _ (Pfield p)        = "p" ++ show p
> showExp xs (x1 `Plus` x2)   = showBin xs " + " x1 x2
> showExp xs (x1 `Minus` x2)  = showBin xs " - " x1 x2
> showExp xs (x1 `Times` x2)  = showBin xs " * " x1 x2
> showExp xs (x1 `Divide` x2) = showBin xs " / " x1 x2
> showExp xs (x1 `Power` x2)  = showBin xs " ^ " x1 x2
> showExp xs (x1 `Modulo` x2) = showBin xs " % " x1 x2
> showExp xs (Int x)          = "int("  ++ showExp xs x ++ ")"
> showExp xs (Frac x)         = "frac(" ++ showExp xs x ++ ")"
> showExp xs (Abs x)          = "abs("  ++ showExp xs x ++ ")"
> showExp xs (Neg x)          = "-("    ++ showExp xs x ++ ")"
> showExp xs (Sqrt x)         = "sqrt(" ++ showExp xs x ++ ")"
> showExp xs (Sin x)          = "sin("  ++ showExp xs x ++ ")"
> showExp xs (Cos x)          = "cos("  ++ showExp xs x ++ ")"
> showExp xs (Exp x)          = "exp("  ++ showExp xs x ++ ")"
> showExp xs (Log x)          = "log("  ++ showExp xs x ++ ")"
> showExp xs (AmpToDb x)      = "dbamp("  ++ showExp xs x ++ ")"
> showExp xs (DbToAmp x)      = "ampdb("  ++ showExp xs x ++ ")"
> showExp xs (PchToHz x)      = "cpspch(" ++ showExp xs x ++ ")"
> showExp xs (HzToPch x)      = "pchoct (octcps(" ++ showExp xs x ++ "))"
> showExp xs (GreaterThan x1 x2 x3 x4)   = showComp xs " > "  x1 x2 x3 x4
> showExp xs (LessThan    x1 x2 x3 x4)   = showComp xs " < "  x1 x2 x3 x4
> showExp xs (GreaterOrEqTo x1 x2 x3 x4) = showComp xs " >= " x1 x2 x3 x4
> showExp xs (LessOrEqTo x1 x2 x3 x4)    = showComp xs " <= " x1 x2 x3 x4
> showExp xs (Equals x1 x2 x3 x4)        = showComp xs " == " x1 x2 x3 x4
> showExp xs (NotEquals x1 x2 x3 x4)     = showComp xs " != " x1 x2 x3 x4
> showExp xs ox = case find (\(nm,oexp) -> ox==oexp) xs of
>                   Just (nm,_) -> nm
>                   Nothing     -> error ("showExp: constructor not found\n")
>
> showBin xs s x1 x2 =
>     "(" ++ showExp xs x1 ++ s ++ showExp xs x2 ++ ")"
> showComp xs s x1 x2 x3 x4 =
>     "(" ++ showExp xs x1 ++ s ++ showExp xs x2 ++ " ? " ++
>             showExp xs x3 ++ " : " ++ showExp xs x4 ++ ")"
```

Figure 13: The Function showExp

```
sr = 48000
kr = 24000
ksmps = 2
nchnls = 2

instr 4

inote = cpspch(p5)

k1 envlpx ampdb(p4), .001, p3, .05, 6, -.1, .01
k2 envlpx ampdb(p4), .0005, .1, .1, 6, -.05, .01
k3 envlpx ampdb(p4), .001, p3, p3, 6, -.3, .01

a1 oscili k1, inote, 1
a2 oscili k1, inote * 1.004, 1
a3 oscili k2, inote * 16, 1
a4 oscili k3, inote, 5
a5 oscili k3, inote * 1.004, 5

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

endin
```

Figure 14: Sample CSound Orchestra File

```
> orc1 :: Orchestra
> 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 = processOrcExp (snd (head (snd orc1)))
```

Figure 15: Haskore Orchestra Definition

```
sr     = 48000
kr     = 24000
ksmps  = 2.0
nchnls = 2

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

Figure 16: Result of `writeOrchestra orc1`

well (for example, in Common Music the user will find a large number of macros which are difficult if not impossible to use in a functional style).

We are not aware of any computer music language that has been shown to exhibit the kinds of algebraic properties that we have demonstrated for Haskore. Indeed, none of the languages that we have investigated make a useful distinction between music and performance, a property that we find especially attractive about the Haskore design. On the other hand, Balaban describes an abstract notion (apparently not yet a programming language) of "music structure," and provides various operators that look similar to ours [Bal92]. In addition, she describes an operation called *flatten* that resembles our literal interpretation `perform`. It would be interesting to translate her ideas into Haskell; the match would likely be good.

Perhaps surprisingly, the work that we find most closely related to ours is not about music at all: it is Henderson's *functional geometry*, a functional language approach to generating computer graphics [Hen82]. There we find a structure that is in spirit very similar to ours: most importantly, a clear distinction between object *description* and *interpretation* (which in this paper we have been calling musical objects and their performance). A similar structure can be found in Arya's *functional animation* work [Ary94].

There are many interesting avenues to pursue with this research. On the theoretical side, we need a deeper investigation of the algebraic structure of music, and would like to express certain modern theories of music in Haskore. The possibility of expressing other scale types instead of the thus far unstated assumption of standard equal temperament scales is another area of investigation. On the practical side, the potential of a graphical interface to Haskore is appealing. We are also interested in extending the methodology to sound synthesis. Our primary goal currently, however, is to continue using Haskore as a vehicle for interesting algorithmic composition (for example, see [HB95]).

A Convenient Functions for Getting Started With Haskore

```
> module TestHaskore where
> import Basics
> import Performance
> import HaskToMidi
> import System( system )
> import GeneralMidi
> import OutputMidi
>
> ------------------------------------------------------------------------
> -- Given a PMap, Context, UserPatchMap, and file name, we can
> -- write a Music value into a midi file:
> ------------------------------------------------------------------------
> mToMF :: PMap -> Context -> UserPatchMap -> String -> Music -> IO ()
> mToMF pmap c upm fn m =
>       let pf = perform pmap c m
>           mf = performToMidi pf upm
```

```
>       in outputMidiFile fn mf
>
> -------------------------------------------------------------------------
> -- Convenient default values and test routines
> -------------------------------------------------------------------------
> -- a default UserPatchMap
> -- Note: the PC sound card I'm using is limited to 9 instruments
> defUpm :: UserPatchMap
> defUpm = [("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
>
> -- a default PMap that makes everything into a fancyPlayer
> defPMap :: String -> Player
> defPMap pname =
>   MkPlayer pname nf pf sf
>   where MkPlayer _ nf pf sf = fancyPlayer
>
> -- a default Context
> defCon :: Context
> defCon = Context { cTime   = 0,
>      cPlayer = fancyPlayer,
>      cInst   = "piano",
>      cDur    = metro 120 qn,
>      cKey    = 0,
>      cVol    = 127 }
>
> -- Using the defaults above, from a Music object, we can:
> -- a) generate a performance
> testPerf  :: Music -> Performance
> testPerf m = perform defPMap defCon m
> testPerfDur  :: Music -> (Performance, DurT)
> testPerfDur m = perf defPMap defCon m
>
> -- b) generate a midifile datatype
> testMidi :: Music -> MidiFile
> testMidi m = performToMidi (testPerf m) defUpm
>
> -- c) generate a midifile
> test     :: Music -> IO ()
> test     m = outputMidiFile "test.mid" (testMidi m)
>
> -- d) generate and play a midifile on Windows 95, Windows NT, or Linux
> testWin95, testNT, testLinux :: Music -> IO ()
```

```
> testWin95 m = do
>                 test m
>                 system "mplayer test.mid"
>                 return ()
> testNT    m = do
>                 test m
>                 system "mplay32 test.mid"
>                 return ()
> testLinux m = do
>                 test m
>                 system "playmidi -rf test.mid"
>                 return ()

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

> play = do
>          system "mplay32 test.mid"
>          return ()

A more general function in the tradition of testMidi, makeMidi also
takes a Context and a UserPatchMap.

> makeMidi :: (Music, Context, UserPatchMap) -> MidiFile
> makeMidi (m,c,upm) = performToMidi (perform defPMap c m) upm

> ------------------------------------------------------------------------
> -- Some General Midi test functions (use 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
> gmUpm = map (\(gmn,n) -> (gmn, gmn, mod n 16 + 1)) genMidiMap
>
> -- 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) genMidiMap)
>                 mu  = line (map simple gMM)
>                 simple (inm,_) = Instr inm cMajArp
>             in  mToMF defPMap defCon gmUpm "test.mid" mu
```

B Examples of Haskore in Action

```
> module HaskoreExamples (module HaskoreExamples, module Haskore, module IO,
```

```
>                          module ChildSong6, module SelfSim)
>        where
>
> import Haskore
> import IO
> import ChildSong6
> import SelfSim
> import Ssf
```

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 "childSong6" imported from module ChildSong. For example:

```
> t0 = test (Instr "piano" childSong6)
```

C Major scale for use in examples below:

```
> cMajScale = Tempo 2
>             (line [c 4 en [], d 4 en [], e 4 en [], f 4 en [],
>                    g 4 en [], a 4 en [], b 4 en [], c 5 en []])
>
> cms' = line [c 4 en [], d 4 en [], e 4 en [], f 4 en [],
>              g 4 en [], a 4 en [], b 4 en [], c 5 en []]
>
> cms = cMajScale
```

Test of various articulations and dynamics:

```
> t1 = test (Instr "percussion"
>        (Phrase [Art (Staccato 0.1)] cms :+:
>         cms                             :+:
>         Phrase [Art (Legato   1.1)] cms    ))
>
> temp = Instr "piano" (Phrase [Dyn (Crescendo 4.0)] (c 4 en []))
>
> mu2 = Instr "vibes"
>        (Phrase [Dyn (Diminuendo 0.75)] cms :+:
>         Phrase [Dyn (Crescendo 4.0), Dyn (Loudness 25)] cms)
> t2 = test mu2
>
> t3 = test (Instr "flute"
>        (Phrase [Dyn (Accelerando 0.3)] cms :+:
>         Phrase [Dyn (Ritardando  0.6)] cms    ))
```

```
-------------------------------------------------------------------------

A function to recursively apply transformations f (to elements in a
sequence) and g (to accumulated phrases):

> rep :: (Music -> Music) -> (Music -> Music) -> Int -> Music -> Music
> rep f g 0 m = 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       = rep (Trans 5) (delay tn) 8 (c 4 tn [])
> cascade   = rep (Trans 4) (delay en) 8 run
> cascades  = rep  id       (delay sn) 2 cascade
> t4' x     = test (Instr "piano" x)
> t4        = test (Instr "piano"
>               (cascades :+: revM cascades))

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

> run'      = rep (delay tn) (Trans 5) 4 (c 4 tn [])
> cascade'  = rep (delay en) (Trans 4) 6 run'
> cascades' = rep (delay sn)  id       2 cascade'
> t5        = test (Instr "piano" cascades')

-------------------------------------------------------------------------

Example from the SelfSim module.

> t10s   = test (rep (delay durss) (Trans 4) 2 ss)

-------------------------------------------------------------------------

Example from the ChildSong6 module.

> cs6 = test childSong6

-------------------------------------------------------------------------

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

> ssf0 = test ssf

-------------------------------------------------------------------------

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

> drums a b = Instr "drums"
>                   (line (map (\p-> Note (pitch p) sn []) [a..b]))
```

```
> t11 a b = test (drums a b)

-------------------------------------------------------------------------------

Test of cut and shorten.

> t12 = test (cut 4 childSong6)
> t12a = test (cms /=: childSong6)

-------------------------------------------------------------------------------

Tests of the trill functions.

> t13note = (Note (C,5) qn [])
> t13 =  test (trill   1 sn t13note)
> t13a = test (trill'  2 dqn t13note)
> t13b = test (trilln  1 5 t13note)
> t13c = test (trilln' 3 7 t13note)
> t13d = test (roll tn t13note)
> t13e = test (Tempo (2/3) (Trans 2 (Instr "piano" (trilln' 2 7 t13note))))

-------------------------------------------------------------------------------

Tests of drum.

> t14 = test (Instr "Drums" (perc AcousticSnare qn []))

> -- a "funk groove"
> t14b = let p1 = perc LowTom        qn []
>            p2 = perc AcousticSnare en []
>        in test (Tempo 3 (Instr "Drums" (cut 8 (repeatM
>                  ((p1 :+: qnr :+: p2 :+: qnr :+: p2 :+:
>                    p1 :+: p1 :+: qnr :+: p2 :+: enr)
>                   :=: roll en (perc ClosedHiHat 2 [])))))))

> -- a "jazz groove"
> t14c = let p1 = perc CrashCymbal2  qn []
>            p2 = perc AcousticSnare en []
>            p3 = perc LowTom        qn []
>        in test (Tempo 3 (Instr "Drums" (cut 4 (repeatM
>                  ((p1 :+: Tempo (3%2) (p2 :+: enr :+: p2))
>                   :=: (p3 :+: qnr)) ))))

> t14d = let p1 = perc LowTom        en []
>            p2 = perc AcousticSnare hn []
>        in test (Instr "Drums"
>                   (  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 into a MIDI file.

```
> tab m = do
>           outputMidiFile "test.mid" $ makeMidi (m, defCon, defUpm)
```

Music to a MidiFile datatype and back to Music.

```
> tad m = readMidi (testMidi m)
```

A MIDI file to a MidiFile datatype and back to a MIDI file.

```
> tcb file = do
>              x <- loadMidiFile file
>              outputMidiFile "test.mid" x
```

MIDI file to MidiFile datatype.

```
> tc file = do
>             x <- loadMidiFile file
>             print x
```

MIDI file to Music, a UserPatchMap, and a Context.

```
> tcd file = do
>              x <- loadMidiFile file
>              print $ fst3 $ readMidi x
>              print $ snd3 $ readMidi x
>              print $ thd3 $ readMidi x
```

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

```
> tcdab file = do
>              x <- loadMidiFile file
>              outputMidiFile "test.mid" $ makeMidi $ readMidi x

> getTracks (MidiFile _ _ trks) = trks
>
> fst3 (a,b,c) = a
> snd3 (a,b,c) = b
> thd3 (a,b,c) = c
```

C Partial Encoding of Chick Corea's "Children's Song No. 6"

```
> module ChildSong6 where
> import Haskore
>
> -- note updaters for mappings
> fd d n = n d v
> vol  n = n   v
> v      = [Volume 80]
> lmap f l = line (map f l)
>
> -- repeat something n times
> times  1    m = m
> times (n+1) m = m :+: (times n m)
>
> -- Baseline:
> b1 = lmap (fd dqn) [b  3, fs 4, g  4, fs 4]
> b2 = lmap (fd dqn) [b  3, es 4, fs 4, es 4]
> b3 = lmap (fd dqn) [as 3, fs 4, g  4, fs 4]
>
> bassLine = times 3 b1 :+: times 2 b2 :+: times 4 b3 :+: times 5 b1
>
> -- Main Voice:
> 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 vol    [cs 5 tn, d 5 (qn-tn), cs 5 en, b 4 en]
>
> v2  = v2a :+: v2b :+: v2c :+: v2d :+: v2e :+: v2f
> v2a = lmap vol [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 vol [cs 6 (hn+en), d 6 en, cs 6 en, e 5 en] :+: enr :+:
>       lmap vol [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 vol [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 vol [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 = Tempo (3/2) (lmap vol [cs 5 en, d 5 en, cs 5 en]) :+: b 4 (3*dhn+hn) v
>
> mainVoice = times 3 v1 :+: v2
>
> -- Putting it all together:
> childSong6 = Instr "piano" (Tempo 3 (Phrase [Dyn SF] bassLine :=: mainVoice))
```

D Example of Simple Self-Similar (Fractal) Music

```
> module SelfSim where
>
> import Haskore

An example of self-similar, or fractal, music.

> data Cluster = Cl SNote [Cluster]  -- this is called a Rose tree
> type Pat     = [SNote]
> type SNote   = [(AbsPitch,Dur)]    -- i.e. a chord
>
> sim :: Pat -> [Cluster]
> sim pat = map mkCluster pat
>     where mkCluster notes = Cl notes (map (mkCluster . addmult notes) pat)
>
> addmult pds iss = zipWith addmult' pds iss
>                   where addmult' (p,d) (i,s) = (p+i,d*s)
>
> simFringe n pat = fringe n (Cl [(0,0)] (sim pat))
>
> fringe 0 (Cl note cls) = [note]
> fringe n (Cl note cls) = concat (map (fringe (n-1)) cls)
>
> -- this just converts the result to Haskore:
> simToHask s = let mkNote (p,d) = Note (pitch p) d []
>               in line (map (chord . map mkNote) s)
>
> -- and here are some examples of it being applied:
>
> sim1 n = Instr "bass"
>            (Trans 36
>               (Tempo 4 (simToHask (simFringe n pat1))))
> t6 = test (sim1 4)
>
> sim2 n = Instr "piano"
>            (Trans 53
>               (Tempo 4 (simToHask (simFringe n pat2))))
> t7 = test (sim2 4)
>
> sim12 n = sim1 n :=: sim2 n
> t8 = test (sim12 4)
>
> sim3 n = Instr "vibes"
>            (Trans 48
>               (Tempo 4 (simToHask (simFringe n pat3))))
> t9 = test (sim3 3)
>
> sim4 n  = (Trans 60
```

```
>               (Tempo 2 (simToHask (simFringe n pat4')))
>
> sim4s n = let s = sim4 n
>               l1 = Instr "flute" s
>               l2 = Instr "bass"  (Trans (-36) (revM s))
>           in l1 :=: l2
>
> ss     = sim4s 3
> durss  = dur ss
>
> t10    = test 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)]]
```

E General Midi

```
> module GeneralMidi where
>
> type GenMidiName = String
> type GenMidiTable = [(GenMidiName,Int)]
>
> genMidiMap :: GenMidiTable
> genMidiMap =[
>  ("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)]
```

References

[AK92] D.P. Anderson and R. Kuivila. Formula: A programming language for expressive computer music. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[Ary94] K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.

[Bal92] M. Balaban. Music structures: Interleaving the temporal and hierarchical aspects of music. In M. Balaban, K. Ebcioglu, and O. Laske, editors, *Understanding Music With AI*, pages 110–139. AAAI Press, 1992.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.

[Col84] D. Collinge. Moxie: A languge for computer music performance. In *Proc. Int'l Computer Music Conference*, pages 217–220. Computer Music Association, 1984.

[CR84] P. Cointe and X. Rodet. Formes: an object and time oriented system for music composition and synthesis. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programmming*, pages 85–95. ACM, 1984.

[Dan89] R.B. Dannenberg. The Canon score language. *Computer Music Journal*, 13(1):47–56, 1989.

[DFV92] R.B. Dannenberg, C.L. Fraley, and P. Velikonja. A functional language for sound synthesis with behavioral abstraction and lazy evaluation. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[For73] A. Forte. *The Structure of Atonal Music*. Yale University Press, New Haven, CT, 1973.

[HB95] P. Hudak and J. Berger. A model of performance, interaction, and improvisation. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1995.

[Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programmming*. ACM, 1982.

[HF92] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMGW96] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3), June 1996. available via `ftp://nebula.systemsz.cs.yale.edu/pub/yale-fp/papers/haskore/hmn-lhs.ps`.

[HS92] G. Haus and A. Sametti. Scoresynth: A system for the synthesis of music scores based on petri nets and a music algebra. In Denis Baggi, editor, *Computer Generated Music*. IEEE Computer Society Press, 1992.

[IMA90] Midi 1.0 detailed specification: Document version 4.1.1, February 1990.

[JB91] D. Jaffe and L. Boynton. An overview of the sound and music kits for the NeXT computer. In S.T. Pope, editor, *The Well-Tempered Object*, pages 107–118. MIT Press, 1991.

[OFLB94] O. Orlarey, D. Fober, S. Letz, and M. Bilton. Lambda calculus and music calculi. In *Proceedings of International Computer Music Conference*. Int'l Computer Music Association, 1994.

[Sch83] B. Schottstaedt. Pla: A composer's idea of a language. *Computer Music Journal*, 7(1):11–20, 1983.

[Ver86] B. Vercoe. Csound: A manual for the audio processing system and supporting programs. Technical report, MIT Media Lab, 1986.