AutoTrack

Stefan Ratschan

April 30, 2000

1 Introduction

This software has a short term and a long term goal. The short term goal is a tool for creating practicing tracks for musicians. For this it is already usable: You feed it with some chord chart, tell it the style of music, and it outputs some MIDI file with a simple drum and bass track over these chords. The long term goal is a sophisticated high-level composing environment, especially useful for creating demos for bands. You should be able to make instructions like: Give me four bars of mainstream jazz over these chords, then switch to heavy-metal, using this melody and these chords, afterwards a short drum break, and so on.

Under Microsoft Windows there are a lot of different programs for music production systems (Cubase, Band-In-A-Box, Finale). Instead of such WYSIWYG systems, the UNIX world has traditionally used language-based approaches in various application areas (e.g. LaTeXfor typesetting). The advantage of the first approach is that it is easier to learn, the advantage of the second approach is that it is more flexible (and one can always add a WYSIWYG interface afterwards). For this software we follow the second approach.

In the area of music various languages for representing and creating music have been developed, see [6], [5], [3] and [4] for just a few examples. Most of the existing systems provide very general languages with an emphasis on gaining theorical insight, while the system, that is presented here, should be *practical* and *useful*.

For writing the software, the library `Haskore` [2] programmed in the functional programming language `Haskell` (see [1, 7] for further references) proved to be the perfect basis for such a system. Another author, Martin Schwenke [8], is working on a similar system, aimed at a slightly different application area.

This document consists of a (short) user-manual, and the literate source code.

2 User Manual

The program acts as a filter, which takes some chord chart from standard input, and writes the corresponding MIDI file to standard output. This output can be directly piped into some MIDI player. Information about run-time options can be obtained by calling the program with the `-h` option.

The syntax of input files is as follows:

```
chart = { (bar | '%') '|' }

bar = { chord | timeSig | '%' }

timeSig = '(' int '/' int ')'
```

Chords follow the usual syntax (e.g., like in the Real Book). The character % acts as a short-cut for repeating the last bar or chord, respectively. Examples of chord charts come with the program distribution.

3 Main Program

We just extract the options from the command-line, and construct a string-to-string filter from the chord-chart and options.

```
> module Main where
> import Options
> import Styles(playToString)

> main :: IO ()
> main = do (t, s, r, c) <- getOpts
>           interact (\str -> playToString r s t c (read str))
```

4 Chord-Symbol-, Scale- and other Charts

```
> module Charts(EventChart(Chart), ChordChart(ChordChart), getChartBars, chartLength,
>               ChartBar, getBarDur, getBarChords, convertChart, hasChord) where
> import Haskore
> import ChordSym
> import Ratio
> import Char(isSpace, isAlpha)
> import Utils(Transp, transp, gcdDur, divide)
```

A bar consists of a time signature and a list of chord symbols. Bars have the following input syntax:

```
bar = { chord | timeSig | '%' | '_' }
timeSig = '(' int '/' int ')'
```

If no time signature is provided then a default is used (within chord chart the time signature of the bar before, and 4/4 for the first bar). The character '%' is a short-cut for the chords just before. The character '_' denotes a break.

```
> data ChartBar = ChartBar Dur [ Maybe ChordSym ]  deriving Show
>
```

```
> instance Read ChartBar where
>   readsPrec p s = readBar (4%4) Nothing s

> readBar :: Dur -> Maybe ChordSym -> ReadS ChartBar
> readBar oldSig oldChord (c:s) | isSpace c = readBar oldSig oldChord s
> readBar oldSig oldChord s@('(':_) =
>     [ (ChartBar r b, r2) | (r, r1) <- readSig s,
>                            (ChartBar sig b, r2) <- readBar r oldChord r1 ]
> readBar oldSig (Just chord) ('%':r) =
>     [ (ChartBar oldSig (Just chord:b), r1) | (ChartBar sig b, r1) <- readBar oldSig (Just chord) r
> readBar oldSig (Just chord) ('_':r) =
>     [ (ChartBar oldSig (Nothing:b), r1) | (ChartBar sig b, r1) <- readBar oldSig Nothing r ]
> readBar oldSig oldChord s@(d:_) | isAlpha d =
>     [ (ChartBar oldSig (Just c:b), r2) | (c, r1) <- reads s,
>                                          (ChartBar sig b, r2) <- readBar oldSig (Just c) r1 ]
> readBar oldSig _ s = [ (ChartBar oldSig [], s) ]
>
> readSig :: ReadS (Ratio Int)
> readSig s@('(':_) = let readRatio s = [ ((p%q), r1) | (p,'/':r) <- reads s, (q, r1) <- reads r ]
>                     in (readParen True readRatio) s
> readSig s = []
```

Bars can be transposed.

```
> instance Transp ChartBar where
>   transp = transpBar

> transpBar :: Int -> ChartBar -> ChartBar
> transpBar i (ChartBar d l) = let tr b = fmap (transp i) b
>                              in ChartBar d (fmap tr l)
```

Accessor functions for the elements of a bar.

```
> getBarDur :: ChartBar -> Dur
> getBarDur (ChartBar d _) = d

> getBarChords :: ChartBar -> [ Maybe ChordSym ]
> getBarChords (ChartBar _ c) = c
```

Chord charts are lists of bars. They have the following input syntax:

```
chart = { (bar | '%') '|' }
```

The character '%' is a shortcut for the same bar as before. Comments can occur everywhere in the text. They start with "–" and continue till the end of the current line.

```
> data ChordChart = ChordChart [ ChartBar ] deriving Show

> getChartBars :: ChordChart -> [ ChartBar ]
> getChartBars (ChordChart c) = c

> chartLength c = length (getChartBars c)

> instance Read ChordChart where
>   readsPrec _ = readChordChart

> concatChordChart :: ChordChart -> ChordChart -> ChordChart
> concatChordChart (ChordChart x) (ChordChart y) = ChordChart (x++y)

> readChordChart :: ReadS ChordChart
> readChordChart s = readChordChart1 (ChartBar (4%4) []) (filterComment s)

> filterComment :: String -> String
> filterComment ('-':'-':r) = filterComment (tail (snd (break (=='\n') r)))
> filterComment (c:r) = (c:filterComment r)
> filterComment "" = ""

> readChordChart1 :: ChartBar -> ReadS ChordChart
> readChordChart1 lb (c:r) | isSpace c =
>     readChordChart1 lb (dropWhile isSpace r)
> readChordChart1 lb ('%':r) =
>     [ (ChordChart (lb:br), r2) | ('|':r1) <- [ dropWhile isSpace r ],
>                                  (ChordChart br, r2) <- readChordChart1 lb r1 ]
> readChordChart1 (ChartBar sig _) s | length s >= 1 =
>     [ (ChordChart (b:br), r1) | (b, ('|':r)) <- readBar sig Nothing s,
>                                 (ChordChart br, r1) <- readChordChart1 b r ]
> readChordChart1 _ s = [ (ChordChart [], s) ]
```

Chord charts can be transposed.

```
> transpChordChart :: Int -> ChordChart -> ChordChart
> transpChordChart i (ChordChart c) = ChordChart (fmap (transp i) c)

> instance Transp ChordChart where
>   transp = transpChordChart
```

We can extract a Boolean list from a chord chart that tells whether there is a chord at a certain position (hc[i] is true iff d*i has a chord).

```
> hasChord :: ChordChart -> (Dur, [ Bool ] )
> hasChord c = let g = barGCD c
>              in (g, hasChord1 g c)
```

```
> hasChord1 :: Dur -> ChordChart -> [ Bool ]
> hasChord1 bDur (ChordChart c) = foldr1 (++) (map (hasChordBar bDur) c)

> hasChordBar :: Dur -> ChartBar -> [ Bool ]
> hasChordBar bDur (ChartBar d chords) = let times = divide (d/((length chords)%1)) bDur
>             createList (Just c) = replicate times True
>                                             createList (Nothing) = replicate times False
>                                         in foldr1 (++) (map createList chords)

> barGCD :: ChordChart -> Dur
> barGCD (ChordChart c) = let chordDur bar = (getBarDur bar)/((length (getBarChords bar))%1)
>                           in foldr1 gcdDur (map chordDur c)
```

Event charts are currently not used. An event chart represents a list of objects of a certain type and duration (the "events").

```
> data EventChart e = Chart [ (Dur, e) ] deriving Show

> concatChart :: EventChart e -> EventChart e -> EventChart e
> concatChart (Chart x) (Chart y) = Chart (x++y)

> convertChart :: ChordChart -> EventChart ChordSym
> convertChart (ChordChart c) = foldl1 concatChart (map convertBar c)

> convertBar :: ChartBar -> EventChart ChordSym
> convertBar (ChartBar d l) = let f (Just c) = (d/((length l)%1), c)
>                               in Chart (fmap f l)
```

Transpose an event chart by a certain number of semitones

```
> transpChart :: (Transp a ) => Int -> (EventChart a) -> (EventChart a)
> transpChart i  = fmap (transp i)

> instance (Transp a) => Transp (EventChart a) where
>   transp = transpChart
```

Event charts can also act as functors:

```
> instance Functor EventChart where
>   fmap = mapChart

> mapChart :: (a -> b) -> EventChart a -> EventChart b
> mapChart f (Chart v) = Chart (map ( \(d, c) -> (d, (f c)) ) v)
```

A certain type of event chart is a scale chart.

```
type ScaleChart = EventChart Scale
```

Conversion from chord chart to ScaleChart. This needs to be improved into a sophisticated scale analyzer.

```
chordToScaleCh :: (EventChart ChordSym) -> ScaleChart
chordToScaleCh (Chart c) = let f = \(d, ch) -> (d, chordToScale ch)
                           in Chart (map f c)
```

5 Styles

```
> module Styles(Style, playToString, jazz, bossa, takeFive, rock) where
> import Random(randomRs, mkStdGen, randomRIO)
> import Ratio
> import Haskore
> import Utils(Transp, transp)
> import Utils(createNote, repeatN, repeatMN, divisible, divide, line0, filterMusic)
> import Rhythm
> import ChordSym
> import Charts
> import Instruments(bass, bottomRange)
```

A style takes a chord chart and creates some music out of it.

```
> type Style = ChordChart -> Music
```

5.1 Drum Fill

¿ jazzFill :: Dur -¿ Music ¿ jazzFill d — d¿=(2¿ (rhythmPercUnitShuffle en (createRhythm ”...x”) SplashCymbal :=: ¿ rhythmPercUnitShuffle en (createRhythm ”...x”) AcousticBassDrum :=: ¿ rhythmPercUnitShuffle en (createRhythm ”.xx.”) AcousticSnare)

¿ endFill :: [ChartBar] -¿ Music ¿ endFill l = let barPause b = Rest (getBarDur b) ¿ in line0 (fmap barPause (init l)) :+: ¿ jazzFill (getBarDur (last l))

5.2 Bass Lines

First some auxiliary function to play the bass note of a chord.

```
> bassChoose :: (Dur, ChordSym) -> Music
> bassChoose (l, (ChS _ b _)) = bassNote l b
```

```
> bassNote :: Dur -> PitchClass -> Music
> bassNote l b = createNote (l, bottomRange bass b)
```

5.2.1 Chart Bass

This bass line style plays the root of a chord on every chord of a chord chart.

```
> chartBass :: Style
> chartBass c = let (Chart ec) = convertChart c
>               in line0 (fmap bassChoose ec)
```

5.2.2 Quarter Bass

This bass line style plays the root of the current chord on every quarter note. It first creates chords on every beat, then maps bassChoose to it. Problem: Right now only works if all chords are on quarter notes!

```
> splitToDur :: [ ( Dur, e ) ] -> Dur -> [ ( Dur, e ) ]
> splitToDur [] _ = []
> splitToDur ((d, e):r) sd | (divisible d sd) =
>              (replicate (divide d sd) ( sd, e ) ) ++ (splitToDur r sd)

> quarterBass :: Style
> quarterBass c = let (Chart ec) = convertChart c
>                 in line0 (fmap bassChoose (splitToDur ec (1%4)))

> eighthBass :: Style
> eighthBass c = let (Chart ec) = convertChart c
>                 in line0 (fmap bassChoose (splitToDur ec (1%8)))
```

5.2.3 Bossa Bass

A simple bass for Bossas using the bass not and its fifth.

```
> bossaBass :: Style
> bossaBass c = let (Chart ec) = convertChart c
>               in line0 (map bossaBassC ec)

> bossaBassC :: (Dur, ChordSym) -> Music
> bossaBassC (l, ch@(ChS r b _)) =
>   let bossa = bassNote (3%8) r :+: bassNote (1%8) (transp 7 r) :+:
>               bassNote (1%2) (transp 7 r) :+:
>               bossaBassC (l-(1%1), ch)
>   in if (l>=(1%1)) then bossa
>      else if (l>=(1%2)) then bassNote (3%8) r :+: bassNote (1%8) r
```

```
>      else bassChoose (l, ch)
```

5.2.4 Walking Bass Line

Creating a good walking bass is a science in itself. There are numerous books which give various rules for creating good bass lines. The following code is still VERY experimental and just follows these basic rules:

- Create the root on the first quarter note of a chord, and
- create random quarter notes of the appropriate scale for the rest.

We do this by creating a walking bass line for every chord of a chart separately and then concatenating the created bass lines.

```
walking :: Style
walking c = let (Chart ec) = convertChart c
            in foldl1 (:+:) (map walkChord ec)
```

Walking bass line for a single chord of a certain length. Take the root for the first note and random notes for the rest.

```
walkChord :: (Dur, ChordSym) -> Music
walkChord (d, ch) | (divisible d (1%4)) =
             bassChoose ((1%4), ch) :+: walkRandom ((divide d (1%4))-1) ch
```

Create a random walking bass line of n quarter notes using chord ch.

```
walkRandom :: Int -> ChordSym -> Music
walkRandom n ch = let scale = (chordToScale ch)
                      choice = \n -> bassChooseR n (1%4, scale)
                  in  line (map choice (take n (randList (length scale))))

bassChooseR :: Int -> (Dur, Scale) -> Music
bassChooseR n (d, s) = createNote (d, pitch (s!!n))
```

5.3 Full Styles

The jazz style works for 3/4 and 4/4 measure. It currently does not yet use walking bass, but uses the quarter bass style above.

```
> jazzDrum :: Dur -> Music
> jazzDrum d | d==(3%4) = (jazzWaltzRideP RideCymbal2) :=: (jazzWaltzHiHatP PedalHiHat)
> jazzDrum d | d==(4%4) = repeatMN 2 ((jazzRideP RideCymbal2) :=: (backBeatP PedalHiHat))
```

```
> jazz :: Style
> jazz s = let drums = line0 (fmap (jazzDrum . getBarDur) (getChartBars s)) :=:
>                      endFill (getChartBars s)
>              (bd, hc) = hasChord s
>          in (filterMusic bd hc drums) :=: (Instr "bass" (quarterBass s))
>
```

The bossa style just plays the usual bossa clave with the hi-hat on the backbeat and some simple bass.

```
> bossa :: Style
> bossa c = let drums = repeatM claveBossa :=:
>                       repeatM ride :=:
>                       repeatM (backBeatP PedalHiHat)
>               bass = (Instr "bass" (bossaBass c))
>           in cut ((4*(chartLength c))%4) drums :=: bass
```

The Take-Five style works for charts with 5/4 measures only.

```
> takeFiveBass :: ChartBar -> Music
> takeFiveBass b | getBarDur b == (5%4) && (length (getBarChords b)) <= 2 =
>     let c=getBarChords b
>         bass d Nothing = Rest d
>         bass d (Just c) = bassChoose (d, c)
>     in if (length c)==2 then bass (3%4) (c!!0) :+: bass (2%4) (c!!1)
>                         else bass (3%4) (c!!0) :+: bass (2%4) (c!!0)

> takeFive :: Style
> takeFive c@(ChordChart l) =
>     let hiHatR = repeatN (length l) (createRhythm "..x .x")
>         cymbalR = repeatN (length l) (createRhythm "x. xx x. x. xx")
>     in rhythmPercUnit qn hiHatR PedalHiHat :=:
>        rhythmPercUnitShuffle en cymbalR RideCymbal2 :=:
>        endFill l :=:
>        (Instr "bass" (line0 (map takeFiveBass l)))
```

The rock style just plays the usual hi-hat eights, bass drum on downbeat, snare on backbeat.

```
> rock :: Style
> rock c = let drums = repeatM basicBassDrum :=:
>                      repeatM basicSnare :=:
>                      repeatM basicHiHat
>              bass = (Instr "bass" (eighthBass c))
>          in cut ((4*(chartLength c))%4) drums :=: bass
```

This style is not yet finished.

```
> thomasCarib :: Style
> thomasCarib c = (backBeatP PedalHiHat) :=:
>                 basicBassDrum :=:
>                 rhythmPercUnitShuffle en (createRhythm ".. .x .x x.") Claves :=:
>                 (Instr "bass" (chartBass c))
```

5.4 Playing Styles

Playing a chord chart and style into a string.

```
> playToString :: Int -> Style -> Int -> Int -> ChordChart -> String
> playToString trans style tempo chornum chart =
>     let countin = countIn (getBarDur (head (getChartBars chart)))
>         choruses = repeatMN chornum (style (transp trans chart))
>         music = Tempo (tempo%60) (countin :+: choruses)
>     in midiFileToString (testMidi music)
```

6 Chord Symbols

```
> module ChordSym(ChordSym(ChS, root, chordtype)) where
> import Haskore
> import Char(isAlphaNum)
> import Utils(Transp, transp)
```

A chord symbol consists of its root, its bass note, and the description of the type of chord. The chord type description is currently in free (string) form and only used by some very experimental code.

```
> data ChordSym= ChS { root::PitchClass,
>                      bassnote::PitchClass,
>                      chordtype::String } deriving Eq
```

Now we define input and output of chord symbols. Note that we denote sharp and flat root notes by '#' and 'b' respectively, instead of 's' and 'f' as in Haskore.

```
> instance Show ChordSym where
>   showsPrec _ ch = (shows (root ch)) . (shows (bassnote ch)) . (shows (chordtype ch))

> instance Read ChordSym where
>  readsPrec _ = readChord
```

```
> readChord :: ReadS ChordSym
> readChord s = [ ((ChS r b t), s3) |
>                       (r, s1) <- [ extractNote s ] ,
>                       (t, s2) <- [ extractOptions s1 ],
>                       (b, s3) <- [ extractBass s2 r ] ]

> extractNote :: String -> (PitchClass, String)
> extractNote (p:'#':r) = (read (p:"s"), r)
> extractNote (p:'b':r) = (read (p:"f"), r)
> extractNote (p:r) = (read [p], r)

> extractOptions :: String -> (String, String)
> extractOptions s = let isNoChordChar c = not (isAlphaNum c || c=='#' || c=='+')
>                     in break isNoChordChar s

> extractBass :: String -> PitchClass -> (PitchClass, String)
> extractBass "" r = (r, "")
> extractBass ('/':s) _ = extractNote s
> extractBass c r = (r, c)
```

We also can transpose chord symbols.

```
> transpChordSym :: Int -> ChordSym -> ChordSym
> transpChordSym i c = ChS (transp i (root c)) (transp i (bassnote  c)) (chordtype c)

> instance Transp ChordSym where
>   transp= transpChordSym
```

7 Instruments

```
> module Instruments(Instrument, bass, bottomRange, topRange) where
> import Haskore
```

Here we store various information about instruments. Currently the only information is the range of an instrument (its highest possible and lowest possible note).

```
> data Instrument = InstrDef { lowest::Pitch, highest::Pitch }

> bass = InstrDef { lowest=(E, 2), highest=(G, 7) }   -- ???
```

Create the deepest/highest note of a certain pitchclass, that an instrument can create.

```
> bottomRange :: Instrument -> PitchClass -> Pitch
> bottomRange instr cl = let (bcl, boct) = lowest instr
>                        in if (cl > bcl) then (cl, boct)
>                               else (cl, boct+1)

> topRange :: Instrument -> PitchClass -> Pitch
> topRange instr cl = let (bcl, boct)= highest instr
>                     in if (cl < bcl) then (cl, boct)
>                                      else (cl, boct-1)
```

8 Rhythm

```
> module Rhythm where
> import Haskore
> import Ratio
> import List(sort)
> import Utils(notIsSpace, divisible, divide, succComb, map2)
```

There are many different possibilities for dealing with the notion of rhythm. Some of them are:

- Modeling it as a succession of notes and rests of equal length
- Allowing notes and rests to be of different (integer or rational) lengths
- Dealing with rhythm on the level of the `Music` data type, without any special data type for modeling rhythm

We will use the first possibility here. The third possibility has been used in Martin Schwenke's `DrumMachine` module.

As explained above we think of rhythm as a succession of notes and rests of equal length. For this we use lists of booleans, where `True` means that a note is played, and `False` means that no note is played (i.e. a rest).

```
> type Rhythm = [ Bool ]
```

By default the basic rhythmical unit is one sixteenth note. The `Rhythm` data-type does not depend on this, it only comes into the game when we convert rhythms to music.

```
> unit :: Dur
> unit = 1%16
```

We provide two ways of creating rhythms:

- From strings, where an 'x' means that some note is played at this place, and any other character means that no note is played, while white spaces are ignored.

- From ordered lists of integers, where every integer means that at the place with this number we have a note (the first place is zero). On all the other places we have rests.

```
> createRhythm :: String -> Rhythm
> createRhythm s = let conv = \c -> if (c=='x') then True else False
>                  in map conv (filter notIsSpace s)

> createRhythmN :: [ Int ] -> Rhythm
> createRhythmN l | sort l == l = let hitAfter = \x -> (replicate x False) ++ [ True ]
>                                     subtr = \x -> \y -> y-x-1
>                                 in foldl1 (++) (map hitAfter (succComb subtr ((-1):l)))
```

Now we want to convert rhythms to music. We do this using two data types, which one can immediately convert to music via function application.

```
> type RhythmM = Music -> Music
> type RhythmP = PercussionSound -> Music

> rhythmMusic :: Rhythm -> RhythmM
> rhythmMusic r m = let play = \b -> if b then m else (Rest (dur m))
>                   in Instr "drums" (line (map play r))

> rhythmPerc :: Rhythm -> RhythmP
> rhythmPerc = rhythmPercUnit unit
```

Sometimes we also want to specify a basic rhythmical unit which is different from the default one.

```
> rhythmPercUnit :: Dur -> Rhythm -> RhythmP
> rhythmPercUnit d r p = rhythmMusic r (perc p d [])
```

Finally one can also create shuffled music from rhythms.

```
> rhythmPercShuffle :: Rhythm -> RhythmP
> rhythmPercShuffle = rhythmPercUnitShuffle unit

> rhythmPercUnitShuffle :: Dur -> Rhythm -> RhythmP
> rhythmPercUnitShuffle d r p =
>             let stretch = 1%3
>                 dstr = d+d*stretch
>                 dcompr = d-d*stretch
>                 play1 = \b -> if b then (perc p dstr []) else (Rest dstr)
>                 play2 = \b -> if b then (perc p dcompr []) else (Rest dcompr)
>             in Instr "drums" (line (map2 play1 play2 r))
```

Some basic rhythms:

```
> tickR = createRhythm "x"

> downBeatR = createRhythm "x."
> backBeatR = createRhythm ".x"

> claveR = createRhythm ("x..x..x.  ..x.x...")

> claveRumbaR = createRhythm ("x..x...x  ..x.x...")

> claveBossaR = createRhythm ("x..x..x.  ..x..x..")

> clave5 = createRhythm "..x.x"

> clave7 = createRhythm ".x.x..x"

> jazzRideR = createRhythm "x.xx"

> jazzWaltzRideR = createRhythm "x.xxx."
> jazzWaltzHiHatR = createRhythm "..x"

> countInR d | d==(4%4) = createRhythm "x.x.xxxx"
> countInR d | d==(5%4) = createRhythm "x..x.xxxxx"
> countInR d | divisible d (1%4) =
>                let b = divide d (1%4)
>                in [True] ++ (replicate (b-1) False) ++ (replicate b True)
```

In one more step in the conversion to music we fix the basic rhythmical unit and shuffle/straight.

```
> tickP = rhythmPercUnit en tickR
> claveP = rhythmPercUnit en claveR
> claveRumbaP = rhythmPercUnit en claveRumbaR
> claveBossaP = rhythmPercUnit en claveBossaR

> jazzRideP = rhythmPercUnitShuffle en jazzRideR

> jazzWaltzRideP = rhythmPercUnitShuffle en jazzWaltzRideR
> jazzWaltzHiHatP = rhythmPercUnit qn jazzWaltzHiHatR

> downBeatP = rhythmPercUnit qn downBeatR
> backBeatP = rhythmPercUnit qn backBeatR
```

And now we assign these rhythms to instruments.

```
> click = repeatM (tickP Claves)
```

```
> clave = claveP Claves
> claveRumba = claveRumbaP Claves
> claveBossa = claveBossaP Claves

> metro5 = rhythmPercUnit qn (cycle clave5) Claves
> metro7 = rhythmPercUnit qn (cycle clave7) Claves

> basicBassDrum = downBeatP AcousticBassDrum
> basicSnare = backBeatP AcousticSnare
> basicHiHat = tickP ClosedHiHat
> ride = tickP RideCymbal2

> countIn m = rhythmPercUnit qn (countInR m) Claves
```

8.1 Utilities

```
> module Utils(Transp, transp, notIsSpace, divisible, divide, succComb, map2, randList,
>              createNote, repeatMN, repeatN, line0, filterMusic, gcdDur) where
> import Haskore
> import System(system)
> import Random(randomRs, mkStdGen, randomRIO)
> import List
> import Char
> import Ratio
```

8.2 Various general utilities

A map function which applies two functions in turn to a list (so we have: `map2 f f = map f`):

```
> map2 :: (a->b) -> (a->b) -> [ a ] -> [ b ]
> map2 _ _ [] = []
> map2 f1 f2 (f:r) = (f1 f):(map2 f2 f1 r)
```

This function combines every pair of following elements in a list with a certain function. Is there any way to define this by library functions?

```
> succComb :: (a->a->a) -> [ a ] -> [ a ]
> succComb f (a:b:[]) = [ f a b ]
> succComb f (a:b:r) = (f a b):(succComb f (b:r))
```

Append n copies of the same list.

```
> repeatN :: Int -> [ a ] -> [ a ]
> repeatN n l = foldl1 (++) (replicate n l)
```

Convert a mapping (i.e. list of pairs) to a function, and use this for a translation function, which translates every character in a by replacing it by looking it up in l2 and replacing it with the according character in l2.

```
> mapToFunction :: (Eq a) => [ (a, a) ] -> (a -> a)
> mapToFunction m a = maybe a id (lookup a m)

> translate :: (Eq a) => [ a ] -> [ a ] -> [ a ] -> [ a ]
> translate l1 l2 a | (length l1)==(length l2) = map (mapToFunction (zip l1 l2)) a
```

Negation of the check for white space:

```
> notIsSpace c = not (isSpace c)
```

A random list of integers between 0 and n. Adapt this to Hugs98

```
> randList :: Int -> [ Int ]
> randList n = map fromInteger (randomRs ((toInteger 0), (toInteger n)) (mkStdGen 0))
```

Is one rational divisible by another one (i.e., is it a integer multiple of it)?

```
> divisible :: Integral a => Ratio a -> Ratio a -> Bool
> divisible r1 r2 = ((numerator r1)*(denominator r2)) `mod`
>                               ((numerator r2)*(denominator r1)) == 0
```

Do the division.

```
> divide :: Integral a => Ratio a -> Ratio a -> a
> divide r1 r2 | divisible r1 r2 = ((numerator r1)*(denominator r2)) `div`
>                                             ((numerator r2)*(denominator r1))

> modulus :: Integral a => Ratio a -> Ratio a -> a
> modulus r1 r2  = ((numerator r1)*(denominator r2)) `mod`
>                               ((numerator r2)*(denominator r1))
```

Also the GCD can be generalized to ratios:

```
> gcdDur :: Dur -> Dur -> Dur
> gcdDur x1 x2 = let a = numerator x1
>                    b = denominator x1
>     c = numerator x2
>                    d = denominator x2
>                in ((gcd a c)%(lcm b d))
```

8.3 Haskore Additions

Here we turn to some stuff that really belongs into the Haskore core. First transposition of pitch classes:

```
> class Transp a where
>   transp :: Int -> a -> a

> transpPitchClass :: Int -> PitchClass -> PitchClass
> transpPitchClass i pc = fst (pitch ((pitchClass pc)+i))

> instance Transp PitchClass where
>   transp = transpPitchClass
```

Creating one note:

```
> createNote :: (Dur, Pitch) -> Music
> createNote (t, p) = (Note p t [])
```

Repeat some piece of music n times.

```
> repeatMN :: Int -> Music -> Music
> repeatMN 1 m =  m
> repeatMN n m =  m :+: repeatMN (n-1) m
```

Make line function work on empty list also.

```
> line0 :: [ Music ] -> Music
> line0 [] = (Rest (0%4))
> line0 l = line l
```

Filtering certain parts from music, in order to introduce rests *after* the creation of some music. The needed information can be encoded in several ways:

(Dur, Dur) : Place of rest, length of rest, sorted

Dur : Place to switch from rest to music, or other way round

1. Dur [Bool]: Some basic duration and then True implies music, False implies Rest

We use the third possibility here, but use a helper function with a more general interface, which additionally specifies the length of the first list member (different from the basic duration).

```
> filterMusic :: Dur -> [ Bool ] -> Music -> Music
> filterMusic = fm (0%4)

> fm :: Dur -> Dur -> [ Bool ] -> Music -> Music
> fm fDur bDur plc (Note p dur att) = (Note p (min dur (musicDur fDur bDur plc)) att)
> fm fDur bDur plc (Rest dur) = (Rest dur)
> fm fDur bDur plc (m1 :=: m2) = (fm fDur bDur plc m1) :=: (fm fDur bDur plc m2)
> fm fDur bDur plc (m1 :+: m2) = let m1' = fm fDur bDur plc m1
>                                    (rFDur, rPlc) = remLen bDur plc ((dur m1)-fDur)
>                                    m2' = fm rFDur bDur rPlc m2
>                                in m1' :+: m2'
> fm fDur bDur plc (Trans  a m)      = Trans a  (fm fDur bDur plc m)
> fm fDur bDur plc (Instr  a m)      = Instr a  (fm fDur bDur plc m)
> fm fDur bDur plc (Player a m)      = Player a (fm fDur bDur plc m)
> fm fDur bDur plc (Phrase a m)      = Phrase a (fm fDur bDur plc m)
> fm fDur bDur plc (Tempo  a m)      = Tempo a  (fm (fDur*a) (bDur*a) plc m)


> remLen :: Dur -> [ Bool ] -> Dur -> (Dur, [ Bool ])
> remLen bDur plc len | bDur>len = (bDur-len, plc)
> remLen bDur (f:plc) len = (len-bDur, plc)

> musicDur fDur bDir [] = 0
> musicDur fDur bDir (False:plc) = 0
> musicDur fDur bDir (True:plc) = fDur + (musicDur bDir bDir plc)
```

For extracting the options from the command line we use the `GetOpt` package proviced by `ghc`. This is currently a little bit of a mess.

```
> module Options(Opts, getOpts) where
> import System
> import GetOpt
> import IO
> import Styles(playToString, Style, jazz, bossa, takeFive, rock)
> import List(find)
> import Charts

> data Opts = OError String | OTempo Int | OStyle Style | OTrans Int | OChoruses Int | OHelp

> isHelp OHelp = True
> isHelp _ = False

> isError (OError _) = True
> isError _ = False

> options :: [ OptDescr Opts ]
> options = [ Option [ 't' ] [ "tempo" ] (ReqArg tempoOption "TEMPO") "TEMPO of track",
>             Option [ 'r' ] [ "transpose" ] (ReqArg transposeOption "TRANSPOSE") "TRANSPOSE track",
```

```
>             Option [ 's' ] [ "style" ] (ReqArg styleOption "STYLE") "music STYLE",
>             Option [ 'c' ] [ "choruses" ] (ReqArg chorusesOption "CHORUSES") "number of CHORUSES",
>             Option [ 'h' ] [ "help" ] (NoArg OHelp) "display usage" ]

> tempoOption s = OTempo (read s)
> transposeOption s = OTrans (read s)

> styles = [("jazz", jazz), ("bossa", bossa), ("takeFive", takeFive), ("rock", rock)]

> styleOption s = maybe (OError "Style not found\n") (\s -> OStyle s) (lookup s styles)

> chorusesOption s = OChoruses (read s)

> usage = usageInfo "\nUsage: track [OPTION...] <infile >outfile\n" options ++
>        "\nAvailable styles: " ++ show (fst (unzip styles)) ++ "\n\n" ++
>        "This program is free software; you can redistribute it and/or\n" ++
>        "modify it under the terms of the GNU General Public License\n" ++
>        "as published by the Free Software Foundation; either version 2\n" ++
>        "of the License, or (at your option) any later version.\n\n" ++
>        "This program is distributed in the hope that it will be useful,\n" ++
>        "but WITHOUT ANY WARRANTY; without even the implied warranty of\n" ++
>        "MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the\n" ++
>        "GNU General Public License for more details.\n"

> processOpts :: [String] -> IO [Opts]
> processOpts argv =
>         case (getOpt Permute options argv) of
>              (o,_,[]  ) -> return o
>              (_,_,errs) -> fail (concat errs ++ usage)

> maybeBool :: (a -> Maybe b) -> (a -> Bool)
> maybeBool m x  = maybe False (\x -> True) (m x)

-- can this be done in a more elegant way?

> getDefault :: (a -> Maybe b) -> [ a ] -> b -> b
> getDefault b (h:t) def | ((maybeBool b) h) = (\(Just t) -> t) (b h)
> getDefault b (h:t) def | not ((maybeBool b) h) = getDefault b t def
> getDefault b [] def = def

> getTempo (OTempo t) = Just t
> getTempo _ = Nothing

> getTrans (OTrans t) = Just t
> getTrans _ = Nothing

> getStyle (OStyle s) = Just s
> getStyle _ = Nothing

> getChoruses (OChoruses c) = Just c
```

```
> getChoruses _ = Nothing

> type OptTuple = (Int, Style, Int, Int)

> flatOpts :: [ Opts ] -> OptTuple
> flatOpts l = (getDefault getTempo l 120,
>               getDefault getStyle l jazz,
>               getDefault getTrans l 0,
>               getDefault getChoruses l 5)

> exit :: String -> Bool -> IO a
> exit m c = do putStr (m ++ usage)
>               if c then exitWith ExitSuccess else exitWith (ExitFailure 1)

> getError :: [ Opts ] -> Maybe String
> getError l = let getMess (OError m) = m
>              in fmap getMess (find isError l)

> getOpts :: IO OptTuple
> getOpts = do argv <- getArgs
>              opts <- processOpts argv
>              if any isHelp opts then exit "" True
>                                 else maybe (return (flatOpts opts)) (\m -> exit m False) (getError
```

9 Todo

- rock style: electric bass
- humanize drums (tempo, single notes)
- modularize styles, make style creation simpler
- walking bass
- recording music / reading in MIDI files
- intros, codas, turnarounds etc.
- breaks (e.g. night in tunesia), rhythmic accents
- different styles within one theme (e.g., on green dolphin street)
- error messages on wrong chord charts (for example takeFive not in 5/4 measure) (prelude function "error") via Monads!!!
- more structured approach to parsing chord charts (either parsing tool/library, or via ReadS, or: treat EBNF rules as function definitions, EBNF operators as combinators)
- various degrees of shuffle

References

[1] Haskell - a purely functional language. `http://www.haskell.org`.

[2] The Haskore computer music system. `http://www.haskell.org/haskore`.

[3] D. P. Anderson and R. J. Kuivila. Formula: A programming language for expressive computer music. *IEEE Computer*, 24(7):12–21, 1991.

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

[5] D. J. Collinge. MOXIE: A language for computer music performance. In *Proc. International Computer Music Conference*, pages 217–20. Computer Music Association, 1984.

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

[7] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation - an algebra of music. *Journal of Functional Programming*, 6(3), June 1996.

[8] M. Schwenke. Ideas for a MIDI sequencer. `http://www.meltin.net/people/martin/sequencer.html`.