ChurchPiano

Ariel R. D'Alessandro

16 de mayo de 2012

1. Introducción

El programa **ChurchPiano** -también llamado El Piano de Church- tiene por objetivo simular la ejecución de un Piano sobre un Standard de Jazz. A su vez, fue diseñado para complementarse con el Contrabajo de **HasBass**, trabajo final ALPII 2011 de Javier M. Corti.

1.1. Compilación

Para compilar ejecutamos en una consola:

```
ghc Proceso.lhs -iHasBass
```

La bandera -iHasBass agrega el subdirectorio ./HasBass/ (ubicado en el actual directorio) a la lista de búsqueda de módulos. De esta manera podemos importar los módulos de HasBass directamente en nuestro programa.

1.2. Ejecución

El usuario deberá pasar por línea de comandos el nombre del archivo de parámetros de entrada, como en el ejemplo:

```
./Proceso Parametros/InfantEyes.in
```

Un archivo de parámetros de entrada se estructura de la siguiente manera.
Veamos **./Parametros/InfantEyes.in**:

```
Inf = (1,C)
InputLHV = Inputs/InputLHV.in
InputESC = Inputs/InputESC.in
InputFRA = Inputs/InputFRA.in
InputMelodia = Midis/InfantEyes.mid
InputArmonia = Midis/InfantEyes.co
OutputFile = output.mid
```

1.3. Inputs

1.3.1. InputMelodia

Ruta del archivo midi que contiene la melodía del Standard de Jazz. El midi de entrada debe consistir de una pista individual.

1.3.2. InputArmonia

Ruta del archivo que contiene el tempo y la sucesión de acordes del Standard de Jazz. Este archivo es igual al que se utiliza en HasBass como entrada, por lo que a continuación se transcribe parte de su informe.

La sintaxis del archivo de entrada sigue la forma de los leadsheet tradicionales, y está dada por:

```
input = tempo sucesion
tempo = int
sucesion = compas '|' sucesion | ''
compas = acorde compas | ''
```

Los Acordes están dados por :

```
acorde = nota tipo | nota tipo '/' nota
nota = notanat | notanat b | notanat #
notanat = A | B | C | D | E | F | G
```

Los tipos de acordes :

```
"", "maj", "+", "aug", "-", "m", "m+", "m-", "0", "dim",
"sus2, "sus4", "4", "0+", "2", "6",
"7", M7", "Ma7", "9", "M9", "11", "13",
"3+", "-5", "5-", "+5", "5+", "-6", "6-", "7+", "-9", "+9", "+11"
```

Veamos **./Midis/InfantEyes.co**:

```
120

Gm7 | Gm7 | Fm7 | EbM7 | A7-9 | GbM7 | Fsus47 | Ebm7 |
Bbsus47 | Bb7 | EbM7 | Gb7/Eb | EbM7+11 | EM7 | BM7 | Bbsus47 |
Abm7 | Ebsus47 | D7-9 |
Gm7 | Fm7 | EbM7 | A7-9 | GbM7 | Fsus47 | Ebm7 |
Bbsus47 |
```

1.3.3. OutputFile

Ruta del archivo midi de salida.

1.3.4. Inf

Nota cota inferior. Todo lo que toque el piano se encontrará por sobre (más agudo) que esta nota.
Inf está dado por:

```
Inf = '(' Int ',' Nota ')'
Nota = Notanat | Notanat 'b' | Notanat '#'
Notanat = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G'
```

1.3.5. InputLHV

Ruta del archivo que determina cómo se construye la armonía de la mano izquierda.
Veamos **./Inputs/InputLHV.in**:

```
M7, 7+, 6 -> [4,7,9,14] [11,12,16,19] # I chord
M7+11 -> [4,7,9,14] [11,12,16,19] # IV Lydian
7 -> [4,9,10,14] [10,14,16,21] # V chord
m7 -> [3,7,10,14] [10,14,15,19] # II chord
m7-5 -> [3,6,10,14] [10,14,15,18] [0,5,6,10] [6,10,14,17] [5,6,10,14] # half-diminished
7-9 -> [10,13,16,21] [4,9,10,13] #
7+5 -> [10,14,16,20] [4,8,10,14] # or 7b13
7-9+5, 7+5-9 -> [10,13,16,20] [4,8,10,13] # or 7b9b13
7+5+9, 7+9+5 -> [10,15,16,20] [4,8,10,15] # or 7alt
7+11 -> [10,16,18,21] [4,6,9,10] #
7+9 -> [4,7,10,15] [10,15,16,19] #
m7+ -> [3,7,11,14] [11,14,15,19] # minor-major
0+, 07 -> [0,3,6,11] # diminished seventh chord
sus47 -> [0,5,10] [3,7,10,14] # Levine's "sus". Ej: Gsus = Dm7/G
sus4-9 -> [0,1,5,7] #
```

InputLHV está dado por:

```
InputLHV = LHVLinea InputLHV | ''
LHVLinea = LHVIzq '->' LHVDer ( '#' Comentario '\n' | '\n' )
LHVIzq = tipo | tipo ',' LHVIzq
LHVDer = Lista | Lista LHVDer
```

1.3.6. InputESC

Ruta del archivo que determina las escalas utilizadas en la improvisación.

```
InputESC = ESCLinea InputESC | ''
ESCLinea = ESCIzq '->' ESCDer ( '#' Comentario '\n' | '\n' )
ESCIzq = LHVIzq
ESCDer = Lista
```

Veamos **./Inputs/InputESC.in**:

```
M7, 7+, 6 -> [0,2,4,7,9,11] # I Ionian
M7+11 -> [0,2,4,6,7,9,11] # IV Lydian
7 -> [0,2,4,7,9,10] # V Mixolydian
m7 -> [0,2,3,5,7,9,10] # II Dorian
m7-5 -> [0,3,5,6,8,10] # VII Locrian
7-9 -> [0,1,3,4,6,7,9,10] # half step/whole step diminished scale
7+5 -> [0,2,4,6,8,10] # whole-tone scale
7-9+5, 7+5-9 -> [0,1,3,4,6,8,10] # or 7b9b13 (same as 7alt)
7+5+9, 7+9+5 -> [0,1,3,4,6,8,10] # or 7alt VII melodic minor scale
7+11 -> [0,2,4,6,7,9,10] # IV Lydian Dominant melodic minor scale
7+9 -> [0,1,3,4,6,7,9,10] # (same as 7-9)
m7+ -> [0,2,3,5,7,9,11] # or minor-major I Melodic Minor Scale
0+, 07 -> [0,2,3,5,6,8,9,11] # whole step/half step diminished scale
sus47 -> [0,2,4,5,7,9,10] # ej. Gsus = Dm7/G... V Mixolydian2
sus4-9 -> [0,1,3,5,7,8,10] # III Phrygian
```

1.3.7. InputFRA

Ruta del archivo que determina las posibles frases rítmicas utilizadas en la improvisación.

Las figuras rítmicas son representadas de la siguiente manera: b = redonda, w = blanca, h = negra, q = corchea, e = semicorchea, s = fusa, t = semifusa.

```
InputFRA = FRALinea InputFRA | ''
FRALinea = Figura ( 'r' | '' ) FRALinea | Tresillo FRALinea | '\n'
Figura = 'b' | 'w' | 'h' | 'q' | 'e' | 's' | 't'
Tresillo = '(' Figura Figura Figura ')3'
```

Veamos **./Inputs/InputFRA.in**:

```
qr q (q q q)3 q q q q
hr e e e e e e e e e e e
q e e q q q q h
q e e q q q q h
(q q q)3 (q q q)3 (q q q)3 (q q q)3
hr hr hr er e (e e e)3 e e e e e e e e e e e e e e e e
qr e e q q q q q q h q w q
```

1.4. Outputs

El programa procesará las entradas y devolverá escribirá en OutputFile el midi de salida, que constará de tres partes:

1. Melodía + Acompañamiento.
2. Improvisación + Acompañamiento.
3. Melodía + Acompañamiento.

Además, en todo momento el Piano estará acompañado por el Contrabajo producido por HasBass.

2. Librerías

2.1. Haskore

La librería más importante del programa es Haskore, que proporciona tanto los tipos de datos básicos como las funciones que permiten manipular música fácilmente. Haskore propone una especificación algebraica de la música (en general representa "multimedia temporal") cuyas primitivas son la composición secuencial (:+:) y la composición paralela (:=:), dur la duración de un "temporal mediaτ none la ausencia de "temporal media"por una duración. Fue utilizado el paquete haskore-0.2.0.2 .

2.2. System.Random

Esta librería es muy útil para simular improvisación mediante pseudo-aleatoridad. En este programa se genera una secuencia pseudo aleatoria (gen) al inicio, y se la pasa a toda función que necesite un valor aleatorio. Cada una de estas funciones debe ademas, retornar una nueva secuencia de donde extraeran valores las funciones que se ejecuten despues. En este sentido la secuncia aleatoria se trata como un estado en el programa. Fue usado el paquete random-1.0.0.3 .

2.3. System.Environment

Esta libreria proporciona las funciones getArgs y getProgramName que le permiten al programa recibir argumentos de linea de comandos y comunicarse con el SO. Incluida en el paquete base-4.3.1.0 .

2.4. Parsec

Se usa parsec para construir el parser del programa, ya que proporciona buenos mensajes de error. Su uso en este programa no difiere demasiado al de la libreria de combinadores de parsers implementada en clase. Para correr el parser se usa la función parse, que da como resultado un tipo Either, donde Left lleva el mensaje de error, en caso de un error en el parseo (Right contiene el resultado de un parseo exitoso). Se usa el paquete parsec-3.1.1 .

2.5. ReadP

Para usar la implementacion del parser que reconoce los tipos de acordes de Haskore, se usa el parser de tipo ReadP. Cuando este parser falla, se invoca manualmente al fail de parsec. El ReadP esta incluida en el paquete base-4.3.1.0 .

2.6. Control.Monad

Cuando se quiere usar liftM o alguna otra función monadica, se importa esta librería, también incluida en el paquete base.

3. General

El objetivo del módulo **General** es definir ciertos tipos comunes usados a lo largo del programa en los demás módulos, facilitando así la comprensión del código.

```
> module General where
```

3.1. Import

```
> import qualified Haskore.Composition.Chord as Chord
> import qualified Haskore.Basic.Pitch as Pitch
> import qualified Haskore.Music as Music
> import qualified Haskore.Melody as Melody
> import qualified Haskore.Melody.Standard as Standard
> import qualified Haskore.Composition.ChordType as ChordType
```

Importamos el módulo **Tipos** de HasBass.

```
> import qualified Tipos
```

3.2. Tipos generales

Definición de los tipos generales usados en ChurchPiano.

Nota de la escala musical en cifrado americano.

```
> type Note = Pitch.Class
```

RelNote (Nota Relativa) es un intervalo en semitonos, representado con un entero.

```
> type RelNote = Pitch.Relative
```

Nota absoluta.

```
> type AbsNote = Pitch.Absolute
```

Haskore define el tipo Chord.T como una lista de notas relativas.
Llamamos Chord Absoluto a una lista de notas absolutas.

```
> type AbsChord = [AbsNote]
```

En el informe de HasBass dice:

> Para reprensentar los acordes se usan datos correspondientes a su notacion simbólica. La nota tónica, la nota más grave del acorde, y el tipo de acorde, representado con un tipo de Haskore. Por ejemplo : Am/G representa el acorde de tónica A, con bajo en G y tipo de acorde menor.

Y define en Tipos.lhs:

> data Chord = Cons { root :: Note, bassnote :: Note, chordType :: ChordType.T } deriving (Eq,Show)

Entonces, usamos acordes como en HasBass y le agregamos duración.

```
> type Acorde = (Tipos.Chord,Music.Dur)
```

Llamamos Acorde Absoluto a un AbsChord y su duración.

```
> type AbsAcorde = (AbsChord,Music.Dur)
```

Trabajamos música con atributos standard.

```
> type Musica = Melody.T Standard.NoteAttributes
```

Una Escala Relativa a una nota tónica es una lista de Notas Relativas.

```
> type EscalaRel = [RelNote]
```

Una Escala Absoluta es una lista de Notas Absolutas.

```
> type EscalaAbs = [AbsNote]
```

Para representar una Figura Rítmica nos interesa saber su duración y si es un Silencio o una Nota.

```
> data Figura = Silencio Music.Dur | Nota Music.Dur
>    deriving (Show,Eq)
```

Una Frase Rítmica es una lista de Figuras.

```
> type Frase = [Figura]
```

3.3. Tipos de las Inputs

Para el procesado de las entradas del programa definimos tres tipos.

Para la Armonía de la Mano Izquierda definimos un conjunto de correspondencia entre acordes y sus posibles formas de ejecución.

```
> type LHVInput = [([ChordType.T],[Chord.T])]
```

Para las Escalas definimos un conjunto de correspondencia entre acordes y la escala que se le asocia.

```
> type ESCInput = [([ChordType.T],EscalaRel)]
```

Para las Frases Rítmicas definimos un conjunto de las mismas.

```
> type FRAInput = [Frase]
```

4. Parseadores

Parseadores es el módulo que proporciona los parsers para leer las entradas del programa, que son utilizados en el módulo **Main**.

```
> module Parseadores where
```

4.1. Import

```
> import Text.ParserCombinators.Parsec
> import qualified Haskore.Composition.ChordType as ChordType
> import qualified Control.Monad as Monad
> import qualified Haskore.Composition.Chord as Chord
> import qualified Haskore.Basic.Pitch as Pitch
> import qualified Haskore.Music as Music
> import qualified Haskore.Basic.Duration as Duration

> import General

> import qualified Tipos
```

4.2. Parsers generales

Los parsers **sepCho** y **parseChordT** han sido copiados de HasBass y son utilizados para parsear los acordes.

```
> sepCho :: Parser String
> sepCho = (string "," <|> string " ")
>
> parseChordT :: Parser ChordType.T
> parseChordT = do s <- manyTill anyChar (lookAhead sepCho)
>                  if null (Tipos.chordTfromString s)
>                    then fail ("\""++s++"\" : Tipo de acorde inesperado")
>                    else return $ (fst.head) $ Tipos.chordTfromString s
```

Definimos algunos parsers básicos.

```
> espacio :: Parser Char
> espacio = char ' '
>
> espacios :: Parser ()
> espacios = do skipMany espacio
>
> espacios1 :: Parser ()
> espacios1 = do skipMany1 espacio
>
> nuevaLinea :: Parser String
> nuevaLinea = string "\n"
>
> parseInt :: Parser Int
> parseInt = (Monad.liftM read $ many1 digit) <?> "entero"
>
> parseLista :: Parser [Int]
> parseLista = do string "["
>                 xs <- (parseInt `sepBy1` string ",") <?> "lista"
>                 string "]"
>                 return xs
```

4.3. Parsers para InputLHV

Definimos los parsers que consumirán el archivo de InputLHV de acuerdo a la gramática ya especificada en la sección 1.3 .

```
> parseLHVIzq :: Parser [ChordType.T]
> parseLHVIzq = do (parseChordT `sepBy1` string ", ") <?> "parte izquierda"
>
> parseLHVDer :: Parser [Chord.T]
> parseLHVDer = do (parseLista `sepEndBy1` espacio) <?> "parte derecha"
>
> parseLHVLinea :: Parser ([ChordType.T],[Chord.T])
> parseLHVLinea = do i <- parseLHVIzq
>                    string " -> "
>                    d <- parseLHVDer
>                    many espacio
>                    (string "#" >> manyTill anyChar nuevaLinea ) <|> nuevaLinea
>                    return (i,d)
>
> parseLHVInput :: Parser LHVInput
> parseLHVInput = do ls <- many parseLHVLinea
>                    eof
>                    return ls
```

4.4. Parsers para InputESC

Definimos los parsers que consumirán el archivo de InputESC de acuerdo a la gramática ya especificada en la sección 1.3 .

```
> parseESCIzq :: Parser [ChordType.T]
> parseESCIzq = parseLHVIzq
>
> parseESCDer :: Parser EscalaRel
> parseESCDer = parseLista
>
> parseESCLinea :: Parser ([ChordType.T],EscalaRel)
> parseESCLinea = do i <- parseESCIzq
>                    string " -> "
>                    d <- parseESCDer
>                    many espacio
>                    (string "#" >> manyTill anyChar nuevaLinea ) <|> nuevaLinea
>                    return (i,d)
>
> parseESCInput :: Parser ESCInput
> parseESCInput = do ls <- many parseESCLinea
>                    eof
>                    return ls
```

4.5. Parsers para InputFRA

Definimos los parsers que consumirán el archivo de InputFRA de acuerdo a la gramática ya especificada en la sección 1.3 .

```
> char2Dur :: Char -> Music.Dur
> char2Dur c = case c of
>                   'b' -> Duration.bn
>                   'w' -> Duration.wn
```

```
>                   'h' -> Duration.hn
>                   'q' -> Duration.qn
>                   'e' -> Duration.en
>                   's' -> Duration.sn
>                   't' -> Duration.tn
>
> parseFRAFigura :: Parser [Figura]
> parseFRAFigura = do f <- oneOf "bwhqest"
>                     (string "r" >> return [Silencio (char2Dur f)]) <|>
>                        return [Nota (char2Dur f)]
```

Definimos el parser de los tresillos dentro de una frase rítmica. Con estas funciones queda abierta la posibilidad de implementar fácilmente otras figuras de valores irregulares (dosillo, tresillo, quintillo, septillo).

```
> escalar :: Music.Dur -> Figura -> Figura
> escalar k f = case f of
>                   Silencio d -> Silencio (k * d)
>                   Nota d -> Nota (k * d)
>
> parseFRATresillo :: Parser [Figura]
> parseFRATresillo = do char '('
>                       espacios
>                       f1 <- parseFRAFigura
>                       espacios1
>                       f2 <- parseFRAFigura
>                       espacios1
>                       f3 <- parseFRAFigura
>                       espacios
>                       char ')'
>                       char '3'
>                       return (map (escalar (2 Duration.%+ 3)) (f1++f2++f3))
>
> parseFRALinea :: Parser Frase
> parseFRALinea = do ts <- ( parseFRATresillo <|> parseFRAFigura ) `sepEndBy1` espacios
>                    rs <- parseFRALinea
>                    return (concat ts ++ rs)
>                 <|>
>                 do nuevaLinea
>                    return []
>
> parseFRAInput :: Parser FRAInput
> parseFRAInput = do ls <- many parseFRALinea
>                    eof
>                    return ls
```

4.6. Parsers para Parametros Generales

Definimos los parsers que consumirán el archivo de InputLHV de acuerdo a la gramática ya especificada en la sección 1.3 .

```
> parseInf :: Parser Int
> parseInf = do string "Inf"
>               espacios
>               string "="
>               espacios
>               string "("
>               n <- parseInt
>               string ","
```

```
>               c <- Tipos.parseNote
>               string ")"
>               espacios
>               nuevaLinea
>               return (Pitch.toInt (n,c))
>
> parseParametro :: String -> Parser String
> parseParametro s = do string s
>                       espacios
>                       string "="
>                       espacios
>                       r <- manyTill anyChar nuevaLinea
>                       return r
>
> parseParametrosInput :: Parser (Int,[String])
> parseParametrosInput = do inf <- parseInf
>                           lhv <- parseParametro "InputLHV"
>                           esc <- parseParametro "InputESC"
>                           fra <- parseParametro "InputFRA"
>                           mel <- parseParametro "InputMelodia"
>                           arm <- parseParametro "InputArmonia"
>                           out <- parseParametro "OutputFile"
>                           eof
>                           return (inf,[lhv,esc,fra,mel,arm,out])
```

Combinamos los parsers de las inputs que contienen las armonías, escalas y frases rítmicas.

```
> parsearInputs :: String -> String -> String -> Either ParseError (LHVInput, ESCInput, FRAInput)
> parsearInputs inpLHV inpESC inpFRA = let x = parse parseLHVInput "LHV" inpLHV
>                                          y = parse parseESCInput "ESC" inpESC
>                                          z = parse parseFRAInput "FRA" inpFRA
>                                      in parsearInputs' x y z
>
> parsearInputs' :: Either err a -> Either err b -> Either err c -> Either err (a,b,c)
> parsearInputs' (Left err) _ _ = Left err
> parsearInputs' _ (Left err) _ = Left err
> parsearInputs' _ _ (Left err) = Left err
> parsearInputs' (Right a) (Right b) (Right c) = Right (a,b,c)
```

5. Main

Este es el módulo principal desde donde se comienza a ejecutar el programa.

```
> module Main where
```

5.1. Import

```
> import Haskore.Music as Music
> import qualified Sound.MIDI.File.Load as LoadMidi
> import System.Random
> import qualified System.Environment as Environment
> import Text.ParserCombinators.Parsec
>
> import General
> import qualified Parseadores
> import Proceso
>
> import qualified Tipos
```

5.2. Función main

La función principal **Main** lee la entrada, la parsea (indicando los casos de error), inicializa el generador de aleatoriedad y llama a la función **proc** del módulo **Proceso**.

Para parsear la entrada se usan los parsers del módulo **Parseadores** y **parseInput** del módulo **Tipos** de **HasBass**.

```
> main :: IO ()
> main = do xs <- Environment.getArgs
>           case (length xs) of
>              1 -> do let [x] = xs
>                      param <- readFile x
>                      case parse Parseadores.parseParametrosInput "" param of
>                         Left err ->
>                            do putStr "Parametros: error de parseo en "
>                               print err
>                         Right (inf,ls) ->
>                            do let [lhvF,escF,fraF,melF,armF,outF] = ls
>                               lhvInp <- readFile lhvF
>                               escInp <- readFile escF
>                               fraInp <- readFile fraF
>                               case Parseadores.parsearInputs lhvInp escInp fraInp of
>                                  Left err ->
>                                     do putStr "Librerias: error de parseo en "
>                                        print err
>                                  Right (l,e,f) ->
>                                     do m <- LoadMidi.fromFile melF
>                                        a <- readFile armF
>                                        g <- newStdGen
>                                        case (parse Tipos.parseInput "" a) of
>                                           Left err ->
>                                              do putStr "Armonia: error de parseo en "
>                                                 print err
>                                           Right (s,bpm) -> proc g s bpm l e f m inf outF
>              _ -> do putStrLn "ERROR: Debe dar la ruta del archivo de param. generales"
```

6. Proceso

El módulo **Proceso** proporciona la función **proc** utilizada en **Main**, sirviéndole de interfaz con el resto de ChurchPiano, terminando de llevar la entrada a una forma interna.

```
> module Proceso where
```

6.1. Import

```
> import Haskore.Music as Music
> import qualified Haskore.Basic.Duration  as Duration
> import qualified Haskore.Music.GeneralMIDI as MidiMusic
> import qualified Haskore.Interface.MIDI.Render as Render
> import qualified Haskore.Melody as Melody
> import qualified Haskore.Melody.Standard as Standard
>
> import Medium.Controlled.List as List
>           (T(Primitive,Serial,Parallel,Control))
> import qualified Sound.MIDI.File as MidiFile
> import qualified Sound.MIDI.File.Save as SaveMidi
> import qualified Haskore.Interface.MIDI.Read as ReadMidi
> import qualified Haskore.Example.Miscellaneous as Miscellaneous
> import qualified Haskore.Music.Rhythmic as RhyMusic
>
> import System.Random
> import qualified System.Environment as Environment
>
> import General
> import qualified ArmoniaLHV
>
> import qualified Tipos
> import qualified Sonido
```

6.2. Procesando la entrada

La función **hacerMusica** toma directamente el archivo midi y lo transforma en algo del tipo Musica. En este proceso se quitan todos los atributos con **quitarAttr** ya que ChurchPiano trabajo sólo con atributos standard.

```
> type MidiArrange = Miscellaneous.MidiArrange
>
> quitarAttr :: MidiMusic.T -> Musica
> quitarAttr = fmap f
>                 where f (Atom dur x) =
>                           case x of
>                                Nothing -> (Atom dur Nothing)
>                                Just note -> let t = RhyMusic.pitch $ RhyMusic.body note
>                                             in Atom dur (Just(Melody.Note Standard.na t))
>
> hacerMusica :: MidiFile.T -> Musica
> hacerMusica t = quitarAttr $ third (ReadMidi.toGMMusic t :: MidiArrange)
>                    where third (_,_,x) = x
```

6.3. Procesando la salida

Como proceso inverso a **hacerMusica**, tenemos la función **hacerMidi** que toma algo del tipo Musica y lo devuelve en una estructura Midi.

```
> hacerMidi :: Musica -> MidiMusic.T
> hacerMidi m = let m' = Music.transpose (-12) (Music.changeTempo (2) m) in
>                  MidiMusic.fromStdMelody MidiMusic.AcousticGrandPiano m'
```

saveToFile f m renderiza el midi **m** y lo almacena en la ubicación **f**.

```
> saveToFile :: String -> MidiMusic.T -> IO ()
> saveToFile f m = SaveMidi.toFile f (Render.generalMidiDeflt m)
```

6.4. Juntando los procesos

Para combinar con el Contrabajo de HasBass usamos **javierElBajo** que devuelve 3 repeticiones de la línea de Contrabajo concatenadas.

```
> javierElBajo :: RandomGen g => g -> Tipos.Song -> Integer -> MidiMusic.T
> javierElBajo g s i = x +:+ x +:+ x
>                         where x = Sonido.process g s i
```

La función principal del módulo es **proc**. Ésta toma todas las entradas del programa y almacena en disco (con **saveToFile**) el midi con la ejecución en simultáneo del ContraBajo y el Piano, combinando el resultado de **javierElBajo** y la función **principal** del módulo **ArmoniaLHV**.

```
> proc :: RandomGen g => g -> Tipos.Song -> Integer -> LHVInput ->
>         ESCInput -> FRAInput -> MidiFile.T -> Int -> String -> IO ()
> proc gen s bpm lhv esc fra mel inf outF =
>    saveToFile outF $
>       javierElBajo gen s bpm
>       =:=
>       ( hacerMidi $ Music.changeTempo (bpm Duration.%+ 60) aux )
>          where aux = ArmoniaLHV.principal gen lhv esc fra inf (hacerMusica mel) s
```

7. ArmoniaLHV

El módulo **ArmoniaLHV** proporciona la función **principal**, que arma toda la composición del piano en el Piano de Church.

```
> module ArmoniaLHV  where
```

7.1. Import

```
> import Haskore.Composition.ChordType as ChordType
> import Haskore.Music as Music
> import Haskore.Basic.Duration as Duration
>           ((%+),fromRatio,divide)
> import qualified Haskore.Composition.Chord as Chord
> import qualified Haskore.Basic.Pitch as Pitch
> import qualified Haskore.Melody as Melody
> import qualified Haskore.Melody.Standard as Standard
>
> import Medium.Controlled.List as List
>           (T(Primitive,Serial,Parallel,Control))

> import System.Random
>
> import General
> import qualified Improvisacion
>
> import qualified Tipos
```

7.2. Armando la armonía de mano izquierda

En base al Input de la Armonía de la Mano Izquierda y dado un acorde, devolvemos una lista con las posibles formas de ejecución.

```
> sugerirLHV :: LHVInput -> ChordType.T -> [Chord.T]
> sugerirLHV [] ch = error "sugerirLHV: inputLHV no es exhaustivo."
> sugerirLHV (x:xs) ch = let (as,bs) = x
>                        in if null $ Prelude.filter (==ch) as
>                             then sugerirLHV xs ch
>                             else bs
```

Transformamos las posibles formas de ejecución en acordes absolutos.

```
> armarLHV :: LHVInput -> Tipos.Chord -> [AbsChord]
> armarLHV inp (Tipos.Cons n b c) = map ( map (Pitch.classToInt n +) ) $ sugerirLHV inp c
```

La función **acomodarLHV** transporta el AbsChord **xs** tal que: inf <= nota más baja del acorde < inf + 12

```
> acomodarLHV :: AbsNote -> AbsChord -> AbsChord
> acomodarLHV inf xs = let h = head xs
>                          d = (div (h-inf) 12) * 12
>                      in map (\n -> n - d) xs
```

Definimos **mediaTonal** como el promedio simple de todas las notas absolutas del acorde.

```
> mediaTonal :: AbsChord -> Double
> mediaTonal xs = sum (map fromIntegral xs) / fromIntegral (length xs)
```

Vamos a necesitar una función auxiliar que dada una lista, determine cuál es el par ordenado cuya segunda componente sea la menor entre todos los pares.

```
> menor2da :: Ord b => [(a,b)] -> (a,b)
> menor2da = foldr1 (\(a,b) (c,d) -> if b <= d then (a,b) else (c,d))
```

Entonces, con las funciones antes definidas podemos armar las posibles ejecuciones de la armonía.

Con **elegirLHV** hacemos esto y elegimos la ejecución -es decir, el acorde absoluto- cuya media tonal es la más cercana a la dada en **m**.

```
> elegirLHV :: LHVInput -> AbsNote -> Double -> Acorde -> AbsAcorde
> elegirLHV inp inf m (c,dur) = let h = map (acomodarLHV inf) $ armarLHV inp c
>                                   f = (\n->abs(n-m)) . mediaTonal
>                               in (fst $ menor2da $ zip h $ map f h, dur)
```

Finalmente, para que resulte agradable al oído, queremos que la transición entre la armonízación de dos acordes sea lo más "suave"posible. Esto lo modelamos escogiendo la menor diferencia de media tonal.

```
> armonizarLHV :: LHVInput -> AbsNote -> [Acorde] -> [AbsAcorde]
> armonizarLHV inp inf = armonizarLHV' inp inf 0
>
> armonizarLHV' :: LHVInput -> AbsNote -> Double -> [Acorde] -> [AbsAcorde]
> armonizarLHV' inp inf n [] = []
> armonizarLHV' inp inf n (x:xs) = c : armonizarLHV' inp inf m xs
>                                  where (c,m) = (elegirLHV inp inf n x, mediaTonal $ fst c)
```

7.3. Evitando superposiciones

Antes de juntar la armonía de la mano izquierda con la melodía debemos asegurarnos que las mismas no se superpongan. Al momento de ejecutar una acorde la melodía debe encontrarse por sobre el mismo, nunca por debajo, como indica Mark Levine "...just make sure that the melody note is on top, where the melody belongs."Para esto vamos transportar la melodía tantas octavas hacia arriba como sea necesario para lograr nuestro objetivo.

La función **calcularDur** toma una grilla de acordes del tipo Tipos.Song utilizado en HasBass, y la transforma en una lista de acordes con sus respectivas duraciones, es decir, del tipo Acorde.

```
> calcularDur :: Tipos.Song -> [Acorde]
> calcularDur = concat . map ( \x -> ponerDur (2 %+ toInteger (length x)) x )
>                  where ponerDur dur = map (\x -> (x,dur))
```

Dada una lista de acordes, obtenemos un lista con las posiciones (distancia del inicio) de cada acorde.

```
> armoniaPosiciones :: [AbsAcorde] -> [Music.Dur]
> armoniaPosiciones = armoniaPosiciones' 0
>
> armoniaPosiciones' :: Dur -> [AbsAcorde] -> [Dur]
> armoniaPosiciones' n [x] = [n]
> armoniaPosiciones' n ((_,dur):xs) = n:(armoniaPosiciones' (n+dur) xs)
```

La función **queSuena dist m** devuelve una lista con las notas que suenan a una distancia **dist** en **m**.

```
> -- TO DO: optimizarla para que tome [Music.Dur] y devuelva el queSuena en cada Music.Dur
> queSuena :: Music.Dur -> Musica -> [Pitch.Absolute]
> queSuena dist (Primitive (Atom dur x)) =
>       if dur > dist
```

```
>        then case x of
>                  Nothing -> []
>                  Just note -> [Pitch.toInt $ Melody.notePitch_ note]
>        else []
> queSuena dist (Serial xs) =
>       case xs of
>            [] -> []
>            (y:ys) -> let d = Music.dur y in
>                      if d > dist then queSuena dist y else queSuena (dist - d) (Serial ys)
> queSuena dist (Parallel xs) =
>       concat $ map (queSuena dist) xs
> queSuena dist (Control cont x) =
>       case cont of
>            Tempo ratio -> queSuena (dist * ratio) x
>            Transpose relat -> map (relat+) $ queSuena dist x
```

Una vez que obtenemos la lista de notas que suena en cada ejecución de un acorde, buscamos cuál es el menor intervalo existente entre estos. Consideramos los intervalos descendentes como negativos, y por lo tanto menores que los ascendentes.

La función **distMin** devuelve Nothing en caso de que no hubiera ninguna nota de la melodía que suene al momento de ejecutar un acorde; en caso contrario retorna (Just n), siendo n la mínima distancia.

```
> distMin :: [(AbsAcorde,[Pitch.Absolute])] -> Maybe Int
> distMin ls = foldr f Nothing $ map distMin' ls
>                 where f = (\x y -> case (x,y) of
>                                         (Nothing,y) -> y
>                                         (x,Nothing) -> x
>                                         (Just a, Just b) -> Just (min a b) )
>
> distMin' :: (AbsAcorde,[Pitch.Absolute]) -> Maybe Int
> distMin' (a,p) = if p == [] then Nothing else Just (minimum p - maximum (fst a))
```

Juntamos todo en **comparar**, que toma una lista de acordes con duración y una melodía devolviendo cuántas octavas se debe alzar la melodía para quedar disjunta de los acordes.

```
> comparar :: [AbsAcorde] -> Musica -> Int
> comparar as m = case distMin $ zip as $ map (\p -> queSuena p m) $ armoniaPosiciones as of
>                      Nothing -> 0
>                      Just n -> if n <= 0 then -(div n 12) else 0
```

7.4. Algunas funciones útiles y necesarias

Pasaje de un acorde a música de Haskore con atributos standares.

```
> aco2mus :: AbsAcorde -> Musica
> aco2mus (xs,dur)= chord $ map (\t -> Melody.note t dur Standard.na) $ map Pitch.fromInt xs
```

Transportamos todo n octavas.

```
> subirOcts :: Int -> Musica -> Musica
> subirOcts n = Music.transpose (n*12)
```

Modificamos la Dinámica para que al volumen se le aplique un factor de (2/3).

```
> atenuar :: Musica -> Musica
> atenuar m = Music.accent (0) $ Music.loudness1 (2/3) m
```

7.5. Juntando el Piano de Church - Principal

La función **principal** arma la música final del Piano, compuesta por 3 partes: acordes más melodía, acordes más improvisación, acordes más melodía.

Los parámetros son: **g** generador de aleatoriedad; **lhv esc fra** inputs de armonía escalas y fraseos respectivamente; **inf** nota cota inferior; **m** melodía del standard de jazz; **s** armonía del standard de jazz.

Para la parte de la improvisación se llama a la función **imp0** que es la más importante del módulo **Improvisacion**.

```
> principal :: RandomGen g => g -> LHVInput -> ESCInput -> FRAInput ->
>              AbsNote -> Musica -> Tipos.Song -> Musica
> principal g lhv esc fra inf m s = let acor = calcularDur s
>                                       armo = armonizarLHV lhv inf acor
>                                       m' = subirOcts (comparar armo m) m
>                                       imp = Improvisacion.imp0 g esc fra (inf+24) acor
>                                       izq = (atenuar (line $ map aco2mus armo)
>                                   in (izq =:= m') +:+
>                                      (izq =:= imp) +:+
>                                      (izq =:= m')
```

8. Improvisacion

El módulo **Improvisacion** aporta la función **imp0** simulando una pseudo-improvisación de un piano de jazz.

```
> module Improvisacion where
```

8.1. Import

```
> import Haskore.Music as Music
> import qualified Haskore.Composition.ChordType as ChordType
> import qualified Haskore.Melody as Melody
> import qualified Haskore.Composition.Chord as Chord
> import qualified Haskore.Basic.Pitch as Pitch
> import Haskore.Basic.Duration as Duration
>           ((%+),fromRatio,divide)
> import qualified Haskore.Melody.Standard as Standard
>
> import System.Random
>
> import qualified Tipos
> import General
```

8.2. Armando las escalas

En base al Input de Escalas y dado un acorde, devolvemos la Escala Relativa asociada.

```
> escala :: ESCInput -> ChordType.T -> EscalaRel
> escala [] ch = error "escala: inputESC no es exhaustivo"
> escala (x:xs) ch = let (as,b) = x
>                    in if null $ Prelude.filter (==ch) as
>                         then escala xs ch
>                         else b
```

Armamos la escala absoluta partiendo de la tónica del acorde. Además la extendemos sobre la octava inferior y superior, y filtramos las notas inferiores a **inf**.

```
> hacerEscala :: ESCInput -> AbsNote -> Tipos.Chord -> EscalaAbs
> hacerEscala inp inf (Tipos.Cons n b c) = Prelude.filter func $
>                                          map (inf + Pitch.classToInt n +) $
>                                          map (\n->n-12) esc ++ esc ++ map (12+) esc
>                                             where esc = escala inp c
>                                                   func = ( \x -> (x < inf) || (x >= inf ) )
>
> ac2es :: ESCInput -> AbsNote -> Acorde -> (EscalaAbs,Dur)
> ac2es inp inf (c,d) = (hacerEscala inp inf c, d)
```

8.3. Armando las frases

Una función auxiliar para saber la duración de una Figura.

```
> durFi :: Figura -> Dur
> durFi (Silencio d) = d
> durFi (Nota d) = d
```

Armamos una frase combinando aleatoriamente las frases del input.

```
> -- TO DO: evitar el peligro de terminar la improvisacion en medio de la frase
> armarFrase :: RandomGen g => g -> FRAInput -> Frase
> armarFrase g fra = let (f,ng) = oneof g fra
>                    in f ++ armarFrase ng fra
```

ponerAc f e toma las figuras de **f** y las va asociando a la escala correspondiente al momento de su ejecución.

De esta manera arma una lista, que sería equivalente a una gran frase rítmica donde se indica a qué escala pertenece cada figura de la misma.

La función pide que la frase tenga una longitud mayor igual que la armonía sobre la que se va a improvisar.

```
> ponerAc :: Frase -> [(EscalaAbs,Dur)] -> [(Figura, EscalaAbs)]
> ponerAc = ponerAc' 0
>
> ponerAc' :: Dur -> Frase -> [(EscalaAbs,Dur)] -> [(Figura, EscalaAbs)]
> ponerAc' _ [] _ = error "ponerAc: Frase demasiado corta\n"
> ponerAc' _ _ [] = []
> ponerAc' acum (f:fs) ((e,d):as) = let df = durFi f
>                                   in if acum < d
>                                        then (f,e) : ponerAc' (acum + df) fs ((e,d):as)
>                                        else ponerAc' (acum - d) (f:fs) as
```

8.4. Funciones auxiliares

Funciones auxiliares para trabajar sobre listas.

```
> primeras2 :: [a] -> [a]
> primeras2 [] = []
> primeras2 (x:[]) = [x]
> primeras2 (z:y:xs) = [z,y]
>
> ultimas2 :: [a] -> [a]
> ultimas2 = primeras2 $ reverse
```

La función **oneof** de Javier Corti elige un elemento al azar de una lista.

```
> oneof :: (RandomGen g) => g -> [a] -> (a,g)
> oneof g xs = (xs !! ri,rg)
>                  where (ri,rg) = randomR (0,length xs - 1) g
```

La función **cerca** toma una nota y una escala, devolviendo (en el caso de que existan) las 2 notas inmediatas inferiores y 2 inmediatas superiores dentro de la escala.

```
> cerca :: AbsNote -> EscalaAbs -> [Pitch.Absolute]
> cerca n e = let (lo,hi) = (Prelude.filter (<n) e, Prelude.filter(>n) e)
>             in ultimas2 lo ++ primeras2 hi
```

8.5. Pseudo-Improvisando

La función principal del módulo es **imp0**. Comenzaremos por **imp3** e iremos subiendo hasta llegar a **imp0**.

En base a una nota dada, elegimos otra nota cercana dentro de la escala.

```
> imp3 :: RandomGen g => g -> AbsNote -> EscalaAbs -> (AbsNote,g)
> imp3 g n e = oneof g (cerca n e)
>
```

Definamos **imp2 g dur n e = (nn, m, ng)**. Siendo **nn** la nota obtenida hecha música, **m** es la misma nota como nota absoluta, **ng** es el nuevo generador de aleatoriedad.

La nota **nn** se elige en base a **n**, pasada como argumento, sobre la escala **e** ejecutando **imp3**.

```
> imp2 :: RandomGen g => g -> Dur -> AbsNote -> EscalaAbs -> (Musica, AbsNote, g)
> imp2 g dur n e = let (m,ng) = imp3 g n e
>                  in (Melody.note (Pitch.fromInt m) dur Standard.na, m, ng)
>
```

Con **imp1** vamos consumiendo las figuras, ejecutando reiteradas veces imp2, haciendo que la salida de una ejecución sea la entrada de la siguiente.

```
> imp1 :: RandomGen g => g -> AbsNote -> [(Figura, EscalaAbs)] -> Musica
> imp1 g n [x] = case x of
>                      (Silencio d, e) -> rest d
>                      (Nota d, e) -> f
>                            where (f,_,_) = imp2 g d n e
> imp1 g n (x:xs) = case x of
>                      (Silencio d, e) -> rest d +:+ imp1 g n xs
>                      (Nota d, e) -> f +:+ imp1 ng m xs
>                            where (f,m,ng) = imp2 g d n e
```

Finalmente, en base a las inputs **esc** y **fra**, generamos una improvisación (música) armónicamente sobre **acs** cuyas notas son más altas que **inf**.

```
> imp0 :: RandomGen g => g -> ESCInput -> FRAInput -> AbsNote -> [Acorde] -> Musica
> imp0 g esc fra inf acs = imp1 g inf (ponerAc (armarFrase g fra) (map (ac2es esc inf) acs))
```