Trabajo Práctico 1
Programación Funcional

Paradigmas de Lenguajes de Programación — 2^{do} cuat. 2020

Fecha de entrega: 24 de septiembre de 2020

1. Introducción

Este trabajo consiste en implementar en Haskell varias funciones que permitan generar melodías y combinarlas.

Melodías

En el contexto de este trabajo, una *melodía* está compuesta por sonidos y silencios, dispuestos a lo largo del tiempo.

Los sonidos se identifican por un *tono*, que corresponde a la "nota musical", y se expresa mediante un número natural. Las duraciones se expresan también con un número natural, que corresponde a una cantidad de unidades de tiempo.[1]

Se definirán los siguientes tipos en Haskell:

```
type Tono     = Integer
type Duracion = Integer
type Instante = Integer

data Melodia = Silencio Duracion
      | Nota Tono Duracion
      | Secuencia Melodia Melodia
      | Paralelo [Melodia]
```

En el caso del constructor Secuencia, la segunda melodía comienza al instante siguiente a la finalización de la primera. En el caso de Paralelo, todas las melodías suenan simultáneamente.

Se asume que tanto los tonos como las duraciones no tendrán valores negativos.

[1] Dado que eventualmente se tendrá la posibilidad de generar archivos MIDI, el tono será un valor entre 0 y 255, que estará asociado al valor correspondiente del formato MIDI (donde do=60, re=62, mi=64, fa=65, etc.). De manera similar, una unidad de tiempo será lo que en términos del formato MIDI es un *beat*.

2. Resolver

Ejercicio 1

Definir las siguientes funciones:

a) superponer :: Melodia → Duracion → Melodia → Melodia, que hace sonar la primera melodía y, luego de un intervalo de tiempo desde el inicio de la misma, comienza a tocar la segunda sin detener la anterior.

b) canon :: Duracion → Integer → Melodia → Melodia, cuya melodía resultante consiste en la reproducción de una misma melodía, tantas veces como el número indicado, donde la duración indica el intervalo de tiempo entre el inicio de cada instancia de la melodía y la siguiente. **Sugerencia:** usar foldNat y la función superponer.

Por ejemplo, sea m una melodía, canon 2 3 m será la melodía resultante de reproducir m 3 veces, empezando una reproducción cada 2 *beats*:

```
canon 2 3 (Nota 60 4) ⇝ Paralelo [Nota 60 4,Secuencia (Silencio 2)
                                  (Paralelo [Nota 60 4,Secuencia (Silencio 2) (Nota 60 4)])]
```

(notar que esta solución no es única debido a la falta de orden dentro de Paralelo)

c) secuenciar :: [Melodia] → Melodia que, dada una lista **no vacía** de melodías, las toca en secuencia preservando el orden de la lista. **Sugerencia:** considerar las distintas variantes de fold.

Por ejemplo:

```
secuenciar [Nota 60 1, Nota 60 2, Nota 60 3]
⇝ Secuencia (Secuencia (Nota 60 1) (Nota 60 2)) (Nota 60 3)
```

Ejercicio 2

Definir la función canonInfinito :: Duracion → Melodia → Melodia, similar a canon, pero genera una melodía infinita. **Sugerencia:** utilizar foldr sobre una lista infinita para lograr la repetición infinita. **Nota:** En este ejercicio es importante **no generar una lista infinita** como parámetro de Paralelo ya que luego habría que reproducir infinitas notas y eso no es posible.

Ejercicio 3

Dar el tipo y definir la función foldMelodia, que implemente un esquema de recursión estructural (fold) para el tipo de datos Melodia. Por tratarse de un esquema de recursión, para definir esta función se permite utilizar **recursión explícita**.

Ejercicio 4

Utilizando foldMelodia, definir las siguientes funciones:

a) mapMelodia :: (Tono → Tono) → Melodia → Melodia, que aplica la función a cada nota de la melodía, manteniendo la estructura de la misma.

b) transportar :: Integer → Melodia → Melodia, tal que transportar n m sea la melodía m transportada n semitonos. Es decir, el resultado de transportar n m debe ser equivalente al de sumarle el valor n a cada tono de m.

c) duracionTotal :: Melodia → Duracion que calcula la duración (en *beats*) de una melodía. Asumir que la composición en paralelo de 0 melodías dura 0 *beats*.

d) cambiarVelocidad :: Float → Melodia → Melodia que utilice el factor de tipo Float para acelerar o desacelerar la melodía. Es decir, la duración de cada nota y silencio (y, por lo tanto, de la melodía en sí) debe multiplicarse por el factor indicado. Notar que las duraciones deben quedar como enteros. Por lo tanto será necesario redondear luego de la multiplicación.
Sugerencia: Utilizar las funciones fromIntegral y round para transformar de Integer a Float y para transformar de Float a Integer, respectivamente.

e) invertir :: Melodia → Melodia que invierte una melodía (las notas y silencios se reproducen en el orden inverso).

Ejercicio 5

- Definir la función notasQueSuenan :: Instante → Melodia → [Tono], que indica qué notas de la melodía están sonando en un instante dado. En instantes menores que 0 no suena ninguna nota (devuelve [] para cualquier melodía). Debe funcionar para melodías finitas e infinitas. Las listas devueltas por la función resultante no deben contener notas repetidas. Para este ejercicio pueden utilizar **recursión explícita**.
Sugerencia: usar la función concatMap.

- Intentar definir esta función usando esquema de recursión foldMelodia. Indicar en comentarios del código por qué no es posible lograrlo.

- Intentar también (si no lo hiceron en el punto anterior) con
notasQueSuenan' :: Melodia → (Instante → [Tono]), es decir haciendo que el tipo de lo que devuelve el fold sean funciones. Indicar en comentarios del código por qué no es posible lograrlo.

Ejercicio 6

Nota: este ejercicio es **OPCIONAL** si entregan el trabajo el día de la presentación (15-sept) antes de las 20:40

Definiendo el tipo Evento de la siguiente manera:

```
data Evento = On Instante Tono | Off Instante Tono
```

a) Escribir la función cambios :: [Tono] → [Tono] → Instante → [Evento], que representa las variaciones entre los tonos en un instante determinado, de la siguiente manera: se debe devolver una función que tome un instante i, tal que, para cada tono t que se encuentre en la segunda lista y no en la primera, el resultado debe contener el elemento On i t, y para cada tono t que se encuentre en la primera lista y no en la segunda, el resultado debe contener el elemento Off i t. No se deben repetir eventos (aun si las listas tuvieran tonos repetidos), ni agregar elementos que no sean los ya mencionados.

Por ejemplo:

cambios [1,2,3,4,5] [1,2,7,5,7,4,9] 1 ⇝ [Off 1 3,On 1 7,On 1 9]

b) Definir la función eventosPorNotas :: (Instante → [Tono]) → Duracion → [Evento] tal que, al recibir una función que indica las notas que suenan en un instante dado y una duración, devuelve una lista con todos los eventos que suceden desde el instante 0 hasta la duración indicada.

Cada evento puede ser de la forma On i tono, indicando que hay que empezar a tocar la nota tono en el instante i, o de la forma Off i tono, indicando que hay que dejar de tocar la nota tono en ese instante.

Por ejemplo, sea f una función que indica que en el instante 0 suena la nota 60, en 1 las notas 60 y 64, en 2 ninguna nota y en 3 la nota 67:

```
eventosPorNotas f 2 ⇝ [On 0 60,On 1 64,Off 2 60,Off 2 64]
eventosPorNotas f 3 ⇝ [On 0 60,On 1 64,Off 2 60,Off 2 64,On 3 67,Off 4 67]
```

Observar que se incluyen los instantes entre 0 y duracion inclusive, y que al instante siguiente se apagan todas las notas que aún continúen sonando. La secuencia de eventos debe estar ordenada crecientemente por el valor del instante, aunque el orden dentro de cada instante es irrelevante.

Sugerencia: usar foldl sobre la lista de 0 a la duración.

c) Definir la función eventos :: Melodia → Duracion → [Evento].

La lista eventos m duracion denota la sucesión de eventos que deben realizarse para ejecutar la melodía m con una duración de duracion unidades de tiempo.

Al igual que en el punto anterior, la secuencia de eventos debe estar ordenada crecientemente por el valor del instante, aunque el orden dentro de cada instante es irrelevante.

Debe funcionar para melodías finitas e infinitas.

Sugerencia: Reutilizar las funciones definidas en los ejercicios 5 y 6.

Por ejemplo, considerando las melodías definidas a continuación:

```
acorde::Melodia
acorde = Paralelo [Nota 60 10,
          Secuencia (Silencio 3) (Nota 64 7),
          Secuencia (Silencio 6) (Nota 67 4)]

doremi::Melodia
doremi = secuenciar [_do 3, _re 1, _mi 3, _do 1, _mi 2, _do 2, _mi 4]
```

La secuencia de eventos asociada es la siguiente:

```
eventos acorde 6 ⇝
    [On 0 60,On 3 64,On 6 67,Off 7 60,Off 7 64,Off 7 67]

eventos doremi 5 ⇝
    [On 0 60,Off 3 60,On 3 62,Off 4 62,On 4 64,Off 6 64]
```

Utilizando la función eventos se pueden escuchar las melodías. En el archivo `Generador.hs` se provee una función:

```
generarMidi :: String → [Evento] → Duracion → IO ()
```

que dado un nombre de archivo, una secuencia de eventos, y la duración en *beats*, genera un MIDI. Por ejemplo:

```
generarMidi "prueba.mid" (eventos (Nota 62 20) 10) 10
```

Tests

Parte de la evaluación de este Trabajo Práctico es la realización de tests. Tanto HUnit[2] como HSpec[3] permiten hacerlo con facilidad.

En el archivo de esqueleto que proveemos se encuentran tests básicos utilizando *HUnit*. Para correrlos, ejecutar dentro de *ghci*:

```
> :l tp1.hs
[1 of 4] Compiling Midi.IOGhc       ( Midi/IOGhc.hs, interpreted )
[2 of 4] Compiling Midi.IOMidi      ( Midi/IOMidi.hs, interpreted )
[3 of 4] Compiling Midi.Midi        ( Midi/Midi.hs, interpreted )
[4 of 4] Compiling Main             ( tp1.hs, interpreted )
Ok, four modules loaded.
> tests
```

Para instalar HUnit usar: `> cabal install hunit` o bien `apt install libghc-hunit-dev`.

Para instalar cabal ver: https://wiki.haskell.org/Cabal-Install

Pautas de Entrega

Se debe entregar el código impreso con la implementación de las funciones pedidas. Cada función asociada a los ejercicios debe contar con ejemplos que muestren que exhibe la funcionalidad solicitada. Además, se debe enviar un e-mail conteniendo el código fuente en Haskell a la dirección plp-docentes@dc.uba.ar. Dicho mail debe cumplir con el siguiente formato:

- El título debe ser `[PLP;TP-PF]` seguido inmediatamente del **nombre del grupo**.
- El código Haskell debe acompañar el e-mail y lo debe hacer en forma de archivo adjunto (puede adjuntarse un `.zip` o `.tar.gz`).
- El código entregado **debe** incluir tests que permitan probar las funciones definidas.

El código debe poder ser ejecutado en Haskell2010. No es necesario entregar un informe sobre el trabajo, alcanza con que el código esté **adecuadamente** comentado (son comentarios adecuados los que ayudan a entender lo que no es evidente o explican decisiones tomadas; no son adecuadas las traducciones al castellano del código). Los objetivos a evaluar son:

- Corrección.
- Declaratividad.
- Prolijidad: evitar repetir código iRNecesariamente y usar adecuadamente las funciones previamente definidas (tener en cuenta tanto las funciones definidas en el enunciado como las definidas por ustedes mismos).

[2] https://hackage.haskell.org/package/HUnit
[3] https://hackage.haskell.org/package/hspec

- Uso adecuado de funciones de alto orden, currificación y esquemas de recursión: es necesario para los ejercicios que usen las funciones que vimos en clase y aprovecharlas, por ejemplo, usar zip, map, filter, take, takeWhile, dropWhile, foldr, foldl, listas por comprensión, etc, cuando sea necesario y no volver a implementarlas.

Salvo donde se indique lo contrario, **no se permite utilizar recursión explícita**, dado que la idea del TP es aprender a aprovechar lo ya definido. Se permite utilizar listas por comprensión y esquemas de recursión definidos en el preludio de Haskell y los módulos `Prelude`, `Data.Char`, `Data.Function`, `Data.List`, `Data.Maybe`, `Data.Ord` y `Data.Tuple`. Las sugerencias de los ejercicios pueden ayudar, pero no es obligatorio seguirlas. Pueden escribirse todas las funciones auxiliares que se requieran, pero estas no pueden usar recursión explícita (ni mutua, ni simulada con `fix`).

Importante: se admitirá un único envío, sin excepción alguna. Por favor planifiquen el trabajo para llegar a tiempo con la entrega.

Referencias del lenguaje Haskell

Como principales referencias del lenguaje de programación Haskell, mencionaremos:

- **The Haskell 2010 Language Report:** el reporte oficial de la última versión del lenguaje Haskell a la fecha, disponible online en: http://www.haskell.org/onlinereport/haskell2010.
- **Learn You a Haskell for Great Good!**: libro accesible, para todas las edades, cubriendo todos los aspectos del lenguaje, notoriamente ilustrado, disponible online en: http://learnyouahaskell.com/chapters.
- **Real World Haskell**: libro apuntado a zanjar la brecha de aplicación de Haskell, enfocándose principalmente en la utilización de estructuras de datos funcionales en la "vida real", disponible online en http://book.realworldhaskell.org/read.
- **Hoogle**: buscador que acepta tanto nombres de funciones y módulos, como signaturas y tipos *parciales*, online en http://www.haskell.org/hoogle.
- **Hayoo!**: buscador de módulos no estándar (i.e. aquellos no necesariamente incluídos con la plataforma Haskell, sino a través de **Hackage**), online en http://holumbus.fh-wedel.de/hayoo/hayoo.html.