JUEGO DE LA VIDA DE CONWAY

Realizado por:

- Nombre y apellidos: Sergio Velázquez García.
  uvus: servelgar.
  email: sergio.050.v.g@gmail.com
- Silvia Cazalla Bazán.
  uvus: silcazbaz.
  email: silviacazallabazan@gamil.com

Introducción:

La temática de nuestro proyecto sería la resolución de problemas de búsqueda y optimización mediante un juego visual autómata. Este juego es conocido como el juego de la vida de Conway, basado en la evolución (vive o muere) de una célula.
Su evolución es determinada por un estado inicial, en nuestro caso uno aleatorio ya que al ser autómata nos facilita mucho más las distintas evoluciones de las células.

Se trata de un juego donde su evolución está determinada por el estado inicial y no necesita ninguna entrada de datos posterior. El Tablero del juego es una matriz cuadrada en nuestro caso 10x10, formada por cuadrados (las "células") que se extiende en todas las direcciones. Por tanto, cada célula tiene 8 células "vecinas", que son las que están próximas a ella, incluidas las diagonales.

Las células tienen dos estados: están "vivas" o "muertas". El estado de las células evoluciona a lo largo de unidades de tiempo discretas (se podría decir que por turnos). El estado de todas las células se tiene en cuenta para calcular el estado de las mismas al turno siguiente. Todas las células se actualizan simultáneamente en cada turno, siguiendo estas reglas:

- Una célula muerta con exactamente 3 células vecinas vivas "nace" (es decir, al turno siguiente estará viva).
- Una célula viva con 2 o 3 células vecinas vivas sigue viva, en otro caso muere (por "soledad" o "superpoblación").

Estructura del juego:

El juego se estructura en 2 módulos (ReglasDelJuego.hs y ModuloJuego.hs) y un archivo Representacion.hs
A continuación se describen todos los archivos detalladamente:

- ReglasDelJuego.hs

Comenzamos creando el modulo y pasandole la variables y métodos que debe exportar a otros archivos para funcionar correctamente.

```
module ReglasJuego
(
   Punto,
   Celula(..),
   Red,
   alto,
   ancho,
   getVecinos,
   getVecinosCoor,
```

```
   calculaVecinosVivos,
   destinoCelulaMuerta,
   destinoCelulaViva
) where
```

Establecemos las dimensiones de la matriz del juego en nuestro caso (10x10) ya que en el juego de conway es una matriz cuadrada. También definimos cada punto en la matriz para poder colocar cada célula en su lugar correspondiente, a su vez el tipo Celula que será el estado de cada celda de nuestra matriz para mostrar si vive o muere en cada iteración.

```
alto = 10
ancho = alto

type Punto = (Int, Int)

data Celula = VIVA | MUERTA deriving (Eq, Ord, Read, Show)

type Red = [Punto]
```

Cada célula tendrá 8 vecinos, aunque para calcular la siguiente generación sólo se utilizan los vecinos que estén “VIVOS”
El método getVecinosCoor es una definición por comprensión, como getVecinos que además utiliza el tipo abstracto “Matrix”. Mediante estos dos métodos conseguimos las coordenadas de los vecinos y establecer estas junto con su estado (viva o muerta) en la matriz.

```
getVecinosCoor :: Punto -> [Punto]
getVecinosCoor (x,y) =
   [(x+i, y+j) | i <- [-1,0,1], j <- [-1,0,1], (i,j) /= (0,0), x+i <=
ancho-1, y+j <= alto-1, x+i >= 0, y+j >= 0 ]

getVecinos :: [(Punto, Celula)] -> (Int, Int) -> [Celula]
getVecinos matriz (x,y) = [c | (coor,c) <- matriz, coor `elem`
vecinosCoor]
   where
       vecinosCoor = getVecinosCoor (x,y)
```

destinoCelulaVIva y destinoCelulaMuerta dos funciones recursivas que añaden los estados antes expuestos de nuestras células (cada celda de la matriz)

```
destinoCelulaViva :: (Ord a, Num a) => a -> Celula
destinoCelulaViva vecinos
   | vecinos < 2 = MUERTA
   | vecinos == 2 || vecinos == 3 = VIVA
   | otherwise = MUERTA

destinoCelulaMuerta :: (Eq a, Num a) => a -> Celula
destinoCelulaMuerta vecinos
   | vecinos /= 3 = MUERTA
   | otherwise = VIVA
```

Calcula los vecinos que están vivos para poder crear la siguiente generación.

```
calculaVecinosVivos :: Num p => [Celula] -> p
calculaVecinosVivos [] = 0
calculaVecinosVivos cs = sum [1 | c <- cs, c == VIVA]
```

- ModuloJuego.hs

Creación del modulo y funciones que debe exportar, además de las distinta importaciones para que funcionen los métodos.

```
module ModuloJuego(
   red,
   agrupar,
   agruparGeneracion,
   agruparGeneracionSinPunto,
   semillaSistema,
   calculaGeneracion,
) where


import System.Random
import System.IO
import GHC.Exts.Heap (GenClosure(FloatClosure))
import Distribution.Simple (Extension(EnableExtension))
import Data.List
import Data.Ratio
```

```
import Control.Monad.Trans.State
import Control.Monad
import Data.Vector.Internal.Check (doChecks)
import Data.Matrix
import Data.Char (isDigit)
import qualified ReglasJuego as R(Celula(VIVA, MUERTA),
Punto,
   Red,
   alto,
   ancho,
   getVecinos,
   getVecinosCoor,
   calculaVecinosVivos,
   destinoCelulaMuerta,
   destinoCelulaViva)
```

Creamos la red para la matriz, función auxiliar recursiva para agrupar listas en listas de n elementos creando la matriz y utilizando orden superior con foldl

```
red :: R.Red
red  = [(x,y) | x <- [0..R.ancho-1], y <- [0..R.ancho-1]]

agrupar :: Int -> [a] -> [[a]]
agrupar _ [] = []
agrupar n l = take n l : agrupar n (drop n l)

agrupar' :: Foldable t => Int -> p -> t [a] -> [[a]]
agrupar' n l = foldl (\y x -> take n x : drop n y) [[]]

-- Primera matriz del sistema
semillaSistema :: (Ord a, Fractional a) => Int -> [a] -> Matrix
(R.Punto, R.Celula)
semillaSistema tam numR = fromLists $ agrupar tam $ zip red $ replicar
(tam*tam) numR
          where
              replicar t [] = []
              replicar t (r:rs)
                  | r > 0.5 = R.VIVA:replicar (t-1) rs
                  | otherwise = R.MUERTA:replicar (t-1) rs
```

Cálculo de la siguiente generación de células usando el tipo abstracto “Matrix”

Agrupamos por punto y estado de la célula (viva, muerta), y por otra parte sólo el estado vivo o muerto en su posición correspondiente para poder representarlo más fácilmente.

```
calculaGeneracion :: Matrix (R.Punto, R.Celula) -> [R.Celula]
calculaGeneracion m = aux $ concat $ toLists m
   where
       aux red = [if c == R.VIVA then
           R.destinoCelulaViva $ R.calculaVecinosVivos $ R.getVecinos
red coor else
               R.destinoCelulaMuerta $ R.calculaVecinosVivos $
R.getVecinos red coor |(coor,c) <- red]

agruparGeneracion :: Int -> [b] -> Matrix (R.Punto, b)
agruparGeneracion tam lg = fromLists $ agrupar tam $ zip red lg

agruparGeneracionSinPunto tam lg = fromLists $ agrupar tam lg
```

- Representación.hs, donde finalmente representamos nuestro juego gráficamente.

Para ello lo primero importar codeWorld y los dos módulos creados anteriormente que contienen los métodos principales del juego

```
{-# LANGUAGE OverloadedStrings #-}
import CodeWorld
import qualified Data.Array as A
import qualified Data.Set as S
import qualified Data.Text as T
import qualified ModuloJuego as M (
   red,
   agrupar,
   agruparGeneracion,
   agruparGeneracionSinPunto,
   semillaSistema,
   calculaGeneracion)

import qualified ReglasJuego as R(
 Celula(VIVA, MUERTA),
 Punto,
   Red,
   alto,
   ancho,
   getVecinos,
   getVecinosCoor,
```

```
   calculaVecinosVivos,
   destinoCelulaMuerta,
   destinoCelulaViva)

import System.Random
import System.IO
import GHC.Exts.Heap (GenClosure(FloatClosure))
import Distribution.Simple (Extension(EnableExtension))
import Data.List
import Data.Ratio
import Control.Monad
import Data.Vector.Internal.Check (doChecks)
import Data.Matrix as Ma
import Data.Char (isDigit)
import qualified Language.Haskell.TH as R
import qualified Language.Haskell.Exts as R
import ModuloJuego (agruparGeneracion)
```

Representación con colores mediante codeworld de nuestra matriz.

```
minimap :: Matrix R.Celula -> Picture
minimap testMap =
 pictures [cell i j | i <- [1..R.alto], j <- [1..R.alto]]
where
 cell i j = translated (fromIntegral i) (fromIntegral j)
            $ colored (minimapColor (testMap Ma.! (i,j)))
            $ solidRectangle 1 1

minimapColor :: R.Celula -> Color
minimapColor R.MUERTA  = grey
minimapColor R.VIVA  = green

siguienteEstado :: Event -> Matrix R.Celula -> Matrix R.Celula
siguienteEstado evento m =
 case evento of
   KeyPress "Enter" -> nuevaMatriz
   _ -> m
   where
     nuevaMatriz =
       M.agruparGeneracionSinPunto R.alto $ M.calculaGeneracion
        $ agruparGeneracion R.alto $ concat $ toLists m

{- MAIN -}
```

```haskell
main :: IO ()
main =  do
 numR <-  replicateM (R.alto*R.alto) (randomRIO (0, 1 :: Double))
 let semilla = M.semillaSistema R.alto numR
 let semillaSinPunto = M.agruparGeneracionSinPunto R.alto  [c |
((_,_),c) <- toList semilla]

 putStrLn  "La semilla inicial, generada aleatoriamente, es:"
 activityOf semillaSinPunto siguienteEstado minimap
```

Cómo compilar y usar el programa, incluyendo varios ejemplos de uso.

Para compilar el programa necesitamos tener todas las dependencias instaladas, además es necesario que los dos módulos se encuentre en un directorio superior al de Representación.hs.

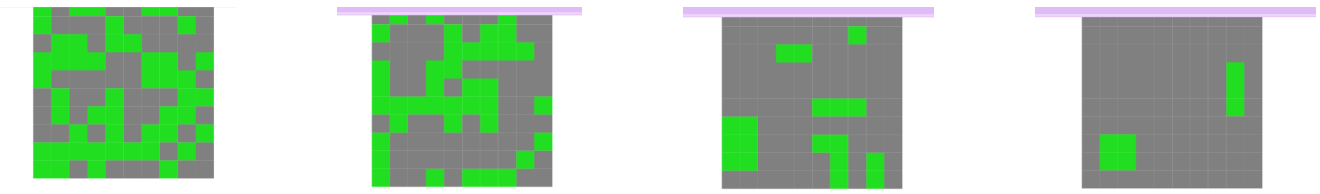
Comenzamos ejecutando cada uno de los módulos para exportar los métodos necesarios, y ejecutamos Representación.hs, esta ejecución nos redirigirá a un enlace de la siguiente manera:

```
PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL

runhaskell "JuegoDeLaVida/Representacion.hs"
silvia@silvia-VirtualBox:~/Escritorio$ runhaskell "JuegoDeLaVida/Representacion.hs"
La semilla inicial, generada aleatoriamente, es:
Open me on http://127.0.0.1:3000/
Program is starting...
Program is starting...

Program is starting...
Program is starting...
```

Haremos Ctrl+Click en el enlace y nos mostrará la matriz de manera gráfica donde las celdas grises son las células muertas y las verdes las vivas, una vez aquí si queremos ver como avanzan las generaciones hasta la solución final solo tenemos que pulsar “enter”



DEPENDENCIAS:

Librería/módulo	Comando
CodeWorld	cabal install codeworld-api
Data.Matrix	cabal install matrix