Propuesta de ejercicio obligatorio para la aprobación del curso de Programación Funcional. Año 2016

UdelaR/FCien/CMat

Noviembre de 2013

1 Objetivo general

La batalla naval es un juego en el que compiten dos jugadores. Cada jugador posee un tablero cuyo contenido se revela paulatinamente, a medida que la partida transcurre.

Se incicializa fijando las dimensiones de los tableros (cantidad de casillas de largo y ancho) y la cantidad y longitud –medida en casillas– de los barcos de los que dispondrá cada jugador. Luego, cada jugador los dispone sobre su tablero sin que el otro jugador sepa su ubicación. La única condición que se pedirá es que los barcos estén completamente dentro del tablero y que no se solapen.

Alternativamente, cada jugador elije una casilla –dada por sus coordenadas– en el tablero del oponente. Este deberá informarle si ha dado en un barco y, en caso de haberlo hecho, si ese barco aún tiene casillas por atacar (es decir, si aún flota o ha sido hundido).

Gana el primer jugador que logra atacar la totalidad de las casillas ocupadas por los barcos del oponente (en otros términos, que ha "hundido" toda la flota del oponente).

Variantes más complejas de estas reglas son:

1. Que los barcos no se puedan disponer en casillas contiguas (que quede siempre "agua" rodéandolos).

2. Que una vez que un jugador dió en un barco, pueda seguir jugando mientras siga dando en algún barco.

Se ha optado por esta versión simplificada, por tratarse de un ejercicio que de por sí evalúa razonablemente el conjunto de las habilidades y conocimientos desarrollados durante el curso.

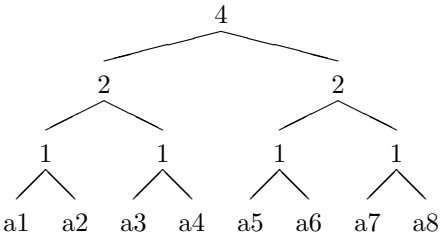
2 El módulo `rows.hs`

En una lista recursiva, como las vistas en el curso, se accede al primer elemento eliminando el constructor más exterior, al segundo eliminando el más exterior y el siguiente y así sucesivamente. En otros términos, se accede al elemento en posición n de una lista recursiva en un tiempo proporcional a n.

Por otra parte, los *tableros de datos* consisten en una lista de datos a cuya información se accede en tiempo constante.

Para implementar la batalla naval en Haskell, el tablero del juego se definirá como un producto de tableros de datos.

Los tableros de datos los implementaremos como árboles binarios equilibrados, con etiquetas de tipo `Int` en los nodos y sus datos a nivel de las hojas. Las etiquetas de los nodos indicarán la cantidad de hojas en cada subárbol debajo de ese nodo.



representa al tablero $\{\ a1 \mid a2 \mid a3 \mid a4 \mid a5 \mid a6 \mid a7 \mid a8\ \}$

1. Definir el tipo **data** `Row a = Leaf a | Thread (Row a) Int (Row a)`.
2. Definir una función `accessRow :: Int -> Row a -> a` que, dados `n::Int` y `xr::Row a` devuelve la entrada de `xr` en la posición `n`. Sugerencia: usar las etiquetas de los nodos.
3. Definir `writeRow :: Int -> a -> Row a -> Row a` tal que, dados `n::Int`, `x::a`, `xr::Row a`; `writeRow n x xr` devuelve el tablero `xr` redefinido para que tenga `x` en su entrada `n`.
4. Definir `initRow :: Int -> a -> Row a` tal que `initRow n x` produce un tablero de largo `n` con todas sus entradas iguales a `x`.
5. Justificar en el código, como comentario, porqué las operaciones de acceso (`accessRow`) y modificación (`writeRow`) toman tiempo constante $t(n)$ que depende de la longitud n del tablero. Hallar $t(n)$.

3 El módulo `board.hs`

Este módulo sirve a describir el contenido del tablero. Cada casilla contiene toda la información necesaria para mostrar el tablero en pantalla y para aplicar una jugada, devolviendo el tablero modificado en concordancia con la jugada –aplicar la *función de transición*–.

3.1 Ships

Cada barco está representado por la lista de todas las casillas que ocupa. Tiene también información sobre la cantidad de casillas "vivas" que aún le quedan. Definir

```
type  Site   =  [(Int, Int)]
type  Lives  =  Int
type  Ship   =  (Site, Lives)
```

3.2 Boards

El tablero de juego está implementado como un tablero (`Row`) de filas. Las filas son tableros y estos tienen en sus entradas casillas. Cada casilla contiene dos datos:

- Un barco (dato de tipo `Ship`). Si no hay barco en la casilla, el barco asociado a la casilla viene representado por la lista vacía.
- Un boleano que indica si ha caído una bomba en esa casilla.

Un tablero tiene además un entero `fleet::Int` que indica la cantidad de barcos no hundidos en el tablero. Definir:

```
type  Bomb   =  Bool
type  Box    =  (Ship, Bomb)
type  Fleet  =  Int
type  Board  =  (Fleet, Row (Row Box))
```

Se definirán funciones auxiliares `site`, `lives`, `attacked` definidas en el tipo `Box` y que calculan respectivamente la lista de casillas ocupadas por el barco asociado a la casilla, la cantidad de casillas vivas que le restan a dicho barco y si la casilla fué o no atacada.

3.3 Funciones de juego

Definiremos una función `shoot` tal que `shoot (c1, c2) xb` modificará el tablero `xb` según un disparo en `(c1,c2)`. Una vez modificado lo devolverá como resultado.

Cada vez que se da en un barco, se debe descontar una vida en todas sus casillas.

1. Definir el sinónimo **type** `Move = (Int, Int)`.
2. Definir `decreaseLive :: Move -> Board -> Board` tal que `decreaseLive (m,n) xb` descuenta una vida al barco en la casilla en posición `(m,n)`[1]. Usarla para definir recursivamente

```
decreaseLives :: Site -> Board -> Board
```

[1] Se actualizará la flota `f::Fleet` en caso de que las vidas del barco se acaben.

tal que `decreaseLives st xb` descuente una vida en todas las casillas cuyas coordenadas figuran en `st`[2].

3. Definir `bombFlag :: Move -> Board -> Board` tal que `bombFlag (m,n) xb` modifique `xb` indicando que cayó una bomba en la posición `(m,n)`.

4. Definir la función `shoot :: Move -> Board -> Board` [3].

5. Definir `finished :: Board -> Bool` que decide si el tablero `Board` tiene todavía barcos a flote.

4 El módulo `St.hs`

El juego se implementará mediante una *mónada de estados*. El *estado* es la información de las posiciones de ambos jugadores en un instante dado. Los tableros de los dos jugadores describen totalmente el estado, aunque les agregaremos las dimensiones para acceder más fácilmente a la información. La estructura general del juego es:

1. Lectura de una movida del jugador 1.
2. Actualización del estado.
3. Lectura de una movida del jugador 2.
4. Actualización del estado.
5. volver a ejecutar estos pasos.

En cada lectura se debe verificar que la movida sea lícita, esto es, que sea adecuada a las dimensiones del tablero y que no se haya jugado antes. Luego de cada actualización, se debe verificar que el jugador al que le corresponde jugar tiene todavía barcos a flote. En caso contrario, el otro jugador gana la partida.

4.1 Mónada de estados

1. Definir los sinónimos

```
type Table  =  (Board, Board)
  type Dim  =  (Int, Int)
type State  =  (Table,Dim)
type Player =  Bool
```

[2]Se supondrá que el barco, al aplicar la función, aún está a flote. Asímismo se supondrá que todas las casillas del barco están en el tablero. Toda la gestión de excepciones queda para la implementación del juego.

[3]Se supondrá que el disparo cae en el tablero y que la casilla no ha sido aún bombardeada al momento de aplicar la función.

Definir funciones `fstBd, sndBd, dimBd` que permitan, dado un estado, calcular respectivamente su primer tablero, segundo tablero y las dimensiones de los mismos.

2. Definir una función `isLegalMove:: Player -> Move -> State -> Bool` tal que `isLegalMove player bd mv st` verifique si `mv` corresponde a una posición en el tablero del jugador `player` (usando las dimensiones de los tableros) y si la casilla no fué aún atacada.

3. Definir la clase **class** `(Monad m)=>StMonad m` caracterizada por una función `tick :: Player -> Move -> m()`[4].

4. Definir **newtype** `St a = MkSt (State -> (a, State))`.
   Definir `apply :: St a -> State -> IO(a,State)` que, dado un elemento de `(MkSt f)::St a` y un estado `s`, aplique `f` a `s`.

5. Declarar `St` como instancia de la clase `Monad`. Luego, declararlo como instancia de `StMonad`. Tener en cuenta que para cada `player::Player`, la función `tick player` debe actualizar el tablero correspondiente a `player`. Se sugiere usar la función `shoot` definida anteriormente.

4.2 Desarrollo de la partida

1. Definir la mónada de excepción:

   ```
   data  Exc a      =  Raise Exception | Return a
   type  Exception  =  String
   ```

   Declarar `Exc` como instancias de la clase `Monad` y `Show` (ver teórico).

2. Definir `stringToIntAux::Int->String->Exc Int` que, usando su primer argumento como parámetro de acumulación:

   - Convierta el argumento de tipo `String` en un entero, si la cadena representa un entero.
   - Levante un mensaje de error –en lo posible indicando el error– sinó.

   Definir `switchWithExc :: Exc a -> IO a -> IO a` tal que `switchWithExc val g` satisface:

   - Si `val` es una excepción, muestra el mensaje correspondiente y luego ejecuta `g`
   - Si `val` es un valor regular (i.e.: `Return x` con `x::a`), devuelve dicho valor.

[4]La diferencia con la mónada de estado vista en clase estriba en que acá la función de actualización está parametrizada en un booleano (que indica cual jugador actúa) y una movida, mientras que la vista en clase es única

3. Con ayuda de `getLine::IO(String)` defina `getInt::IO(Int)` que lea una cadena de caracteres y proceda según:

   - La analice para ver si corresponde a un entero.
   - En caso afirmativo, la convierta en entero y lo devuelva como valor.
   - En caso contrario, muestre un mensaje de error y solicite al usuario que lo reingrese.

4. Definir `isWinner::Player -> State -> Bool` tal que `isWinner player st` decida si el jugador `player` ha ganado la partida (i.e.: si en el estado `st`, el tablero del otro jugador tiene el valor de `fleet::Fleet` igual a cero).

5. Hemos definido las funciones necesarias a la gestión de la interacción durante el juego. Algunas requieren de la mónada `IO` porque realizan una interacción con el usuario. La única que debe ser definida en la mónada de estado es `tick` puesto que es la que actualiza los tableros luego de una jugada. Para poder definir la acción que comanda todo el juego, trabajaremos en la mónada con más estructura, esto es, en `St`.

   Definir `ioToSt:: IO a -> St a` tal que `ioToSt p` calcule el mismo valor de tipo `a` que `p` y que no modifique el estado.

   Recordar que, además, un dato de tipo `a` siempre se puede "exportar" a la instancia `St a` de `St` mediante `return`.

Hasta acá, se puede jugar la partida si se fija el estado inicial, esto es, el par de tableros del comienzo con sus barcos (`Table`) y sus dimensiones (`Dim`). Ahora deberemos ocuparnos de la inicialización del juego, esto es, de fijar las dimensiones de los tableros, las longitudes de los barcos y de preguntarle a cada jugador las posiciones de sus barcos.

4.3 Inicialización del juego

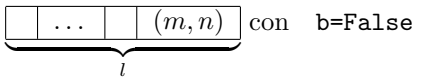
Para inicializar la partida es necesario:

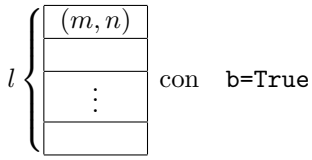
1. Las dimensiones de ambos tableros (Se supondrá que ambos tableros son iguales. Se podría permitir que fueran de diferentes tamaños, pero no parece razonable del punto de vista del juego).

2. La lista de enteros que representa las longitudes de los barcos de la flota de cada jugador. Así la lista `[1,1,3,2,5]` indica que la flota de cada jugador consiste en dos barcos de largo 1, uno de largo 3, otro de largo 2 y otro de largo 5.

3. Una vez fijadas las dimensiones y la flota se debe preguntar las posiciones de los barcos. Los barcos están determinados por tres valores de enteros y un booleano: La cuaterna `(m,n,l,b)::(Int, Int, Int, Bool)` determina al barco cuya casilla más arriba y más a la derecha tiene coordenadas

`(m,n)`, tiene largo `l` y su orientación es horizontal si `b=Flase` o vertical si `b=True`.





La función `getInt::IO Int` será de utilidad aquíPara leer la orientación del barco, definir `getChar::IO Char` y usarla para codificarla mediante `'h'` y `'v'`, por "horizontal" y "vertical" respectivamente.

1. Definir `getBoardSpecs::IO(Dim,Fleet,[Int])` que lea del teclado las dimensiones del tablero, la cantidad de barcos de la flota y la lista de las longitudes de los barcos (sug.: preguntar primero el tamaño `fleet` de la flota y luego ir preguntando la longitud del i-ésimo barco, con i entre 1 y el valor de `fleet`).

2. Definir `initEmptyBoard :: Dim -> Board` que produzca un tablero sin barcos (i.e: todas las `Box` tienen un barco sin casillas ni vidas) y de las dimensiones que indique su argumento.

3. Definir `isCorrectShip :: Dim -> Board -> Ship -> Bool` tal que `isCorrectShip d bd sp` vale `True` si el barco `sp` cabe en el tablero y no se solapa con ningún otro y `False` en caso contrario.

4. Definir `getShip::Int->Dim->Board->IO(Ship)` tal que `getShip l d bd` solicite al usuario ingresar un barco de longitud `l`, preguntando las coordenadas de su casilla más arriba y más a la derecha y su orientación[5]. Luego se procederá según:

   - Verifica si lo puede agregar al tablero `bd`.
   - En caso afirmativo, retorna el barco; representado como un dato del tipo `Ship`.
   - En caso contrario, muestra un mensaje de error –explicando el problema– y solicita que se vuelva a ingresar.

5. Definir `addShip :: Ship -> Board -> Board` tal que `addShip sp bd` agregue el barco `sp` al tablero `bd`[6].

[5] ver en 3.3.3 la representación de los barcos mediante una cuaterna `(Int, Int, Int, Bool)`
[6] Se supondrá que el barco se puede agregar al tablero

6. Definir `initBoard :: Dim -> [Int] -> IO Board` tal que `initBoard d ls` construya –preguntando y leyendo del teclado– un tablero de dimensiones `d` y cuya flota tiene lista de longitudes `ls`.

7. Definir `initState :: Dim -> [Int]->IO State` que usando `initBoard` devuelva el estado incial del juego.

4.4 Visualización del juego (el módulo Main.hs)

Se importará el módulo `Graphics.UI.WX` En el repositorio se encuentra código que debería funcionar cuando tengan instalada correctamente la biblioteca. Pueden tomarlo como base para modificarlo e implementar lo que falte para que el juego funcione. En el ejemplo se crean dos paneles (ventanas) con eventos "onclick" sobre ellas, que representan a los tableros.

Se define el siguiente tipo de datos:

```
data Grid = Grid { hGrid :: [Point]
                 , vGrid :: [Point] }
```

El tipo `Point` se define en `Graphics.UI.WX` , y consiste en un par de enteros que representan las coordenadas en un panel. `Grid` representa los puntos en los bordes (superior e izquierdo) desde donde se dibujan las lineas del tablero. La función `initGrid :: Int -> Int -> Grid` genera una grilla para un tablero de la dimensión de los parámetros.
La función `drawHLine :: Int -> DC a -> Point2 Int -> IO ()` se define tal que `drawHLine l dc point` dibuja una linea horizontal de largo (en pixeles) igual a `l`, desde el punto `point`. Análogamente se define `drawVLine` para lineas verticales. Investigar en la documentación de WXHaskell la función `line` y funciones similares. El tipo `DC a` (Device Context) contiene la información de con qué parámetros vamos a dibujar objetos (colores, texturas, etc).

1. Investigar la documentación de WXHaskell, y entender cómo funciona el ejemplo que se provee.

2. Definir una función `setup :: IO State` que combine `getBoardSpecs` e `initState`, para configurar el estado inicial del juego en la terminal. Los jugadores ingresan los datos por turnos.
   Usar la función `clearScreen :: IO ()` para ocultar los datos luego de que un jugador termina de ingresarlos. (Investigar en la documentación en que módulo se encuentra).

3. Definir `drawBox :: Int -> Int -> Box -> DC a -> IO ()` tal que `drawBox m n box dc` dibuja el contenido de la casilla box en la posición (m, n) del tablero. Notar que estas coordenadas del tablero son distintas a las coordenadas donde dibujar (y las que retorna el input de ratón del punto 5), (aunque se relacionan). Para dibujar una casilla, investigue en la documentación qué funciones puede usar (`rectangle`, por ejemplo). Lo que se dibujará debe reflejar el contenido de `Box`. Por ejemplo se pueden asignar colores distintos dependiendo de si hay agua o un barco.

4. Definir una función `drawBoard :: DC a -> Board -> IO ()` que dado un tablero, lo dibuje.

5. Definir:

   ```
   changeState
   :: Var Player -> Point2 Int
   -> Var State -> Player -> Panel () -> IO ()
   ```

   tal que `changeState turn clickcoords state player p)` cambie el estado del juego, sabiendo que es el turno de `turn` (se debe tomar en cuenta para que no se pueda jugar en un panel cuando no es el turno del jugador l que le corresponde), el click fue en el punto `clickcoords`, en el panel que le corresponde al jugador `player`, y `state` la variable de estado. Además se pasa el panel `p` como parámetro (para redibujarlo, ver ejemplo en el repositorio).

6. Modificar (usando como base el ejemplo) las funciones `main` y `gameFrame` para que computen el juego.

7. Agregar un nuevo widget (por ejemplo otra ventana -o Frame, en términos de WX-) que muestre información sobre la partida (de quien es el turno, si la partida terminó, etc). Agregar un botón para cerrar el juego (el programa finaliza).