Programación Funcional

Parcial - Nim

El juego del Nim es un juego en el que dos jugadores se alternan para remover fichas de distintas pilas. En su turno un jugador debe elegir una (y solo una) pila de la cual remover fichas y quitar tantas como desee (teniendo que remover al menos una), lo cual se indica dando el número de la pila de dónde se eliminarán fichas, y la cantidad de fichas a eliminar. El jugador que remueve la última ficha es considerado el ganador.

a) Juego de Nim con una sola pila de 3 fichas.

b) Nim con 3 pilas de distintas fichas cada una.

Figura 1: dos posibles juegos de Nim

El árbol del juego representa todas las posibles elecciones de los jugadores. En el primer nivel se encuentran las posibles elecciones del jugador 1 en su primer turno, en el segundo nivel se encuentran las posibles elecciones del jugador 2 (supeditadas a las jugadas previas) y así sucesivamente. Las hojas representan los estados en los que un jugador removió las últimas fichas y por ende obtuvo la victoria (ver Figura 2 para el árbol de juego de un Nim con 2 pilas de 2 y 1 fichas).

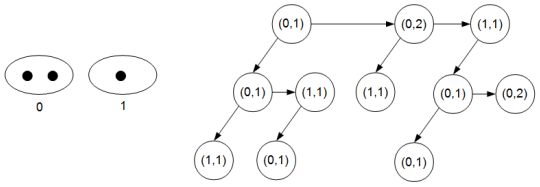

Figura 2: Árbol de juego para un Nim de 2 pilas con 2 y 1 fichas respectivamente.

Considerar la siguiente representación para el juego del Nim y el árbol del juego:

```
data Nim = Empty        -- juego vacío
         | Heap Int Nim -- pila con n fichas (n>0)

type Move = (Int,Int)   -- jugada: de la fila i remover k fichas

data GameTree = Nil                    -- árbol vacío
              | Node (Move, GameTree) -- (jugada, hijos - jugadas del contrincante)
                     GameTree         -- hermanos (otras jugadas propias)
```

Observar que el **`GameTree`** tiene dos casos recursivos, el primero hace referencia al primero de los hijos de la jugada almacenada en el nodo (jugadas del contrincante), mientras que el segundo referencia al primero de los hermanos de la jugada (otras jugadas propias – del jugador actual).

Ejercicios

1. Dar el tipo y la implementación para los esquemas de recursión sobre las estructuras **`Nim`** y **`GameTree`** (**`foldNim`**, **`recNim`**, **`foldGame`** y **`recGame`** respectivamente).
2. Escriba las siguientes funciones sin usar recursión explícita.
   a. **`heaps :: Nim -> Int`**, que indica la cantidad de pilas en un juego de **`Nim`** dado.
   b. **`chips :: Nim -> Int`**, que indica la cantidad de fichas en un juego de **`Nim`** dado.
   c. **`maxHeap :: Nim -> Int`**, que computa la cantidad de fichas en el heap más grande.
   d. **`alongside :: Nim -> Nim -> Nim`**, que une dos juegos de **`Nim`** en uno solo que contiene las pilas de ambos en orden. Por ejemplo, el juego de la Figura 1b) puede obtenerse como merge de los juegos de las Figuras 1a) y 2).
   e. **`gameHeight :: GameTree -> Int`**, que retorna la altura de un **`GameTree`**, considerado como árbol general (por ejemplo, el árbol de la Figura 2 tiene altura de juego 3, porque el juego más largo que se puede jugar contiene 3 jugadas).
   f. **`branches :: GameTree -> [[Move]]`**, que retorna la lista de ramas en un **`GameTree`**, considerado como árbol general. Una rama es el desarrollo de un juego completo de Nim hasta que haya un ganador.
3. Escriba las siguientes funciones (si lo desea puede usar recursión explícita):
   a. **`turn :: Nim -> Move -> Maybe Nim`**, que lleva a cabo una jugada si es posible.
   b. **`moves :: Nim -> [Move]`**, que retorna la lista de jugadas válidas del jugador actual.
   c. **`solve :: Nim -> GameTree`**, que construye el árbol de juego con todas las jugadas válidas.
   AYUDA: definir **`genGT :: Nim -> [Move] -> GameTree`**.
   d. **(BONUS)** **`winning :: Nim -> Bool`**, que indica si el jugador 1 tiene una estrategia ganadora, es decir, si puede ganar de alguna forma, sin importar las jugadas que haga el contrincante.
   PISTA: definir dos funciones mutuamente recursivas **`hasWinningMove :: GameTree -> Bool`** y **`cannotWin :: GameTree -> Bool`**.
4. Considere la estructura de los árboles generales y las siguientes funciones:

```
data AG a = NodeAG a [AG a]

toAG :: GameTree -> [ AG Move ]
toAG Nil                  = []
toAG (Node (move,tC) tP) = NodeAG move (toAG tC) : toAG tP

toGT :: [ AG Move ] -> GameTree
toGT []                  = Nil
toGT (NodeAG move mcs : mps) = Node (move, toGT mcs) (toGT mps)
```

Demostrar que

```
¿ toGT . toAG = id ?
```