PROGRAMACIÓN FUNCIONAL

Comenzado el	viernes, 5 de julio de 2019, 19:02
Estado	Finalizado
Finalizado en	viernes, 5 de julio de 2019, 19:47
Tiempo empleado	44 minutos 52 segundos

Finalizado

Puntúa como 1,0

Dado el siguiente tipo de datos:

```
data HijaraPlayer = BluePlayer | YellowPlayer deriving (Eq, Show, Enum)
```

se desea implementar la función `isValidAction`, que dado un estado de juego y un par *jugador-acción* determina si la acción es válida para el estado y jugador dados.

¿Cuál de las siguientes es la mejor implementación? (1p)

Seleccione una:

a. Ninguna de las implementaciones dadas es correcta.

b.

```
isValidAction :: HijaraGame -> (HijaraPlayer, HijaraAction) -> Bool
isValidAction g (p,a) = head $ filter (\(p1, as) -> p1 == p && elem a as) (actions g)
```

c.

```
isValidAction :: HijaraGame -> (HijaraPlayer, HijaraAction) -> Bool
isValidAction g (p,a) = head [elem a as | (p, as) <- actions g]
```

d.

```
isValidAction :: HijaraGame -> (HijaraPlayer, HijaraAction) -> Bool
isValidAction g (p,a) = elem (p, a) (actions g)
```

Finalizado

Puntúa como 1,0

Dado el tipo de datos de Haskell para representar las acciones de los jugadores en el juego *Hijara*:

```
data HijaraAction = HijaraAction Int Int deriving (Eq, Show, Enum) -- row, column
```

se desea implementar la función `showAction`, usando la notación clásica para las coordenadas en un tablero de *Ajedrez*. Las filas se enumeran con los números del 1 al 8, y las columnas con las letras de la A a la H. En el caso del *Hijara*, las filas serían del 1 al 4 y las columnas de la A a la D.

Cada acción se convierte a texto usando dos caracteres: el primero para la columna y el segundo para la fila. Por ejemplo

```
showAction (HijaraAction 1 1) = "A1"
showAction (HijaraAction 2 4) = "D2"
showAction (HijaraAction 3 2) = "B3"
```

¿Cuál de las siguientes es la mejor implementación? (1p)

Seleccione una:

a.

```
showAction :: HijaraAction -> String
showAction (HijaraAction r c) = ("ABCD" !! c):[r]
```

b.

```
import Data.Char
showAction :: HijaraAction -> String
showAction (HijaraAction r c) = (chr ((ord 'A') + c): (show r)
```

```
showAction (MiJarAction r c) = ( ABCD  !! c): (show r)
```

d. Ninguna de las implementaciones dadas es correcta.

Finalizado

Puntúa como 4,0

Adjuntar a la respuesta a esta pregunta una modificación de la entrega del proyecto de curso con las siguientes modificaciones:

- Sumar al *score* el número de cada casilla donde cada jugador tenga una pieza. Es decir, cada pieza en una casilla 1 vale 1 punto, cada pieza en una casilla 2 vale 2 puntos, y así con las casillas 3 y 4.
- Los jugadores tienen 30 piezas (en lugar de las 32 anteriores). Esto quiere decir que la partida termina con 4 casillas vacías.

Escribir en la respuesta un breve resumen sobre los cambios realizados en el código adjunto. (4p)

1) Creé una nueva función recursiva que verifica si la pieza en determinada posición es igual a la pieza recibida por parámetro, de forma muy similar a cómo se calcularon los puntajes anteriores
2) Cambié la función "ActivePlayer", para que en vez de preguntar si hay 64 piezas en el tablero, pregunte por 60, de forma que el tablero siempre tiene 60 fichas al darlo por finalizado

SamuelParaElTe.hs

Finalizado

Puntúa como 1,0

Dadas las definiciones de Haskell realizadas para implementar el juego *Hijara*, se desea implementar la función `playthrough`, que dado un estado de juego y una lista de pares *jugador-acción* calcula la lista de estados de juego resultantes.

Las acciones en la lista se aplican de primera a última, y los estados de juego en la lista resultado deben estar en ese orden. El primer valor en la lista resultado es el estado de juego inicial dado.

¿Cuál de las siguientes es la mejor implementación? (1p)

Seleccione una:

a.

```
playthrough :: HijaraGame -> [(HijaraPlayer, HijaraAction)] ->
[HijaraGame]
playthrough g as = map (next g) as
```

b.

```
playthrough :: HijaraGame -> [(HijaraPlayer, HijaraAction)] ->
[HijaraGame]
playthrough g as = foldl foldFunc [g] as
   where foldFunc gs@(g:_) (p,a) = (next g (p, a)):gs
```

c. Ninguna de las implementaciones dadas es correcta.

d.

```
playthrough :: HijaraGame -> [(HijaraPlayer, HijaraAction)] ->
[HijaraGame]
playthrough g as = foldl foldFunc [g] as
   where foldFunc gs (p,a) = gs ++ [next (last gs) (p, a)]
```

Ir a...

Síguenos