Introducción a los Algoritmos | Lic. Ciencias de la Computación | FAMAF

Autor: Kevin Leonel Santillan; Comisión: 4; Profesor: Marcos Gomez

Presentación de la Base

Este informe contiene un extracto de los datos oficiales de los años 2014 y 2015 de la temporada regular de la liga deportiva National Basketball Association, (NBA).
Los datos se obtuvieron del sitio:
https://www.nba.com/stats/leaders/?Season=2014-15&SeasonType=Regular%20Season&StatCategory=MIN&CF=MIN*G*2&PerMode=Totals
La base contiene datos como el nombre del jugador, la cantidad de partidos que disputó, la cantidad de minutos jugados, los puntos realizados, y también la cantidad de tiros de campo encestados e intentados así como también la cantidad de triples hechos e intentados.
A Continuación se presenta la forma de la base con un extracto de cinco 5 registros, y más adelante se especificará el contenido del campo.

NAME	GAMES	MIN	PTS	FGM	FGA	PM3	PA3
Stephen Curry	80	2613	1900	653	1341	286	646
James Harden	81	2981	2271	647	1470	208	555
Luis Scola	81	1661	763	300	642	5	20
Manu Ginobili	70	1587	738	251	589	89	258
Chris Paul	82	2857	1564	568	1170	139	349

Estructura de Tipo de la Base de Datos

A continuación se define un nuevo tipo en Haskell para las tuplas, (registros), que tenemos en la lista, (base de datos).

```
type Player = (String, Int, Int, Int, Int, Int, Int, Int)

-- <NAME, GAMES, MIN, PTS, FGM, FGA, PM3, PA3>

players :: [Player]

players = [(String, Int, Int, Int, Int, Int, Int, Int) … ]
```

Proyecciones Definición y Descripción

A continuación se define cada una de las proyecciones, (Notacion Haskell), para este proyecto. Luego se especifica el tipo de campo que representa y el tipo de dato que retorna tal proyección.

Definición name:

```
name :: Player -> String
name (player, games, minutes, pts, fgm, fga, pm3, pa3) = player
```

Descripción:

name : este campo representa el nombre del jugador, En haskell el tipo de dato es String.

Definición de cant_games:

```
cant_games :: Player -> Int
cant_games (player, games, minutes, pts, fgm, fga, pm3, pa3) = games
```

Descripción:

games : este campo ocupa el tipo de dato Int, representa la cantidad de partidos jugados por el jugador.

Definición cant_min:

```
cant_min :: Player -> Int
cant_min (player, games, minutes, pts, fgm, fga, pm3, pa3) = minutes
```

Descripción:

minutes : este campo representa la cantidad de minutos hechas por el jugador, el tipo de dato en Haskell es Int.

Definición cant_pts:

```
cant_pts :: Player -> Int
cant_pts (player, games, minutes, pts, fgm, fga, pm3, pa3) = pts
```

Descripción:

pts : este campo representa la cantidad de puntaje hecha por el basquetbolista. El tipo de dato en Haskell es Int.

Definición cant_fgm:

```
cant_fgm :: Player -> Int
cant_fgm (player, games, minutes, pts, fgm, fga, pm3, pa3) = fgm
```

Descripción:

fgm : este campo representa la cantidad de tiros de campo encestados por el jugador. En Haskell el tipo de dato es Int.

Definición cant_fga:

```
cant_fga :: Player -> Int
cant_fga (player, games, minutes, pts, fgm, fga, pm3, pa3) = fga
```

Descripción:

fga : este campo representa la cantidad de tiros de campo intentados por el basquetbolista. El tipo de dato en Haskell es Int.

Definición cant_pm3:

```
cant_pm3 :: Player -> Int
cant_pm3 (player, games, minutes, pts, fgm, fga, pm3, pa3) = pm3
```

Descripción:

pm3 : este campo representa la cantidad de triples hechas por el jugador. En Haskell el tipo de dato es Int.

Definición cant_pa3:

```
cant_pa3 :: Player -> Int
cant_pa3 (player, games, minutes, pts, fgm, fga, pm3, pa3) = pa3
```

Descripción:

pa3 : este campo representa la cantidad de triples intentados por el jugador. En Haskell el tipo de dato es Int.

Análisis

Vamos ahora a definir las cinco funciones recursivas, las cuales extraen información sobre la base. Luego se dará la descripción sobre las mismas. Pero primero recordemos algunas definiciones útiles (*) :

- *Función Fold:* dada una lista devuelve un valor resultante de combinar los elementos de la lista.
- *Función Map*: dada una lista devuelve otra lista donde los elementos de la nueva lista fueron transformados a través de una función aplicada a cada uno de los elementos de la lista original.
- *Función Filter:* dada una lista, se devuelve otra lista donde permanecen los elementos que cumplen una condición x.

Vamos ahora con las definiciones y descripciones;

Definición:

```
totalPlayers :: [Player] -> Int
totalPlayers [] = 0
totalPlayers (x:xs) = 1 + totalPlayers xs
```

Descripción: De acuerdo con la definición está función pertenece a la clase fold. Luego la función toma una lista de jugadores y cuenta la cantidad de estos. El tipo de dato que retorna es Int.

Definición:

```
totalPts :: [Player] -> Int
totalPts [] = 0
totalPts (x:xs) = cant_pts x + totalPts (xs)
```

Descripción: Esta función lo que hace es tomar una lista de jugadores y calcular la sumatoria de

todos los puntos de todos los jugadores. De acuerdo con la definición esta función es de tipo fold y el tipo de dato que devuelve es Int.

Definición:

```
leaders :: [Player] -> [Player]
leaders [] = []
leaders (x:xs) | (cant_pts x > 718) = x : leaders xs

               | otherwise = leaders xs

leaders1 :: [Player] -> [Player]

                        (función del coloquio)

leaders1 [] = []

leaders1 (x:xs) | (cant_pts x < 100)= x : leaders1 xs

                | ¬(cant_pts x < 100) = leaders1 xs
```

Descripción: Esta función toma una lista de jugadores y devuelve otra lista de jugadores pero que cumplan la condición de que los jugadores en la nueva lista tengan un total de puntos mayores a la media. (los cuales fueron calculados gracias a una función auxiliar que nos devuelve el Int 718). De acuerdo con la descripción la función es de la clase filter.

Definición:

```
leadersMap :: [Player] -> [String]
leadersMap [] = []
leadersMap (x:xs) | (cant_pts x > 718) = (name x) : leadersMap xs

                  | otherwise = leadersMap xs
```

Descripción: Esta nueva función recursiva según las definiciones **(*)** son una combinación entre las clases map y filter. La función hará que el filtro extraiga a los jugadores con un puntaje superior a la media. Luego map nos lleva al nombre de los jugadores que cumplen tal condición. La función entonces toma una lista de jugadores y devuelve una lista de String.

Definición:

```
king3 :: [Player] -> [Player]
```

```
king3 [x1] = [x1]
king3 (x1:x2:xs)
| (cant_pm3 x1 > cant_pm3 x2) = king3 (x1:xs)

| otherwise = king3 (x2:xs)
```

Descripción: la última función que se presenta es de la clase filter, toma una lista, se le da la condición, (quedarse con el jugador que más triples anotó), entonces la función tendrá un caso base el cual es un primer elemento x1. El caso recursivo va a comparar la cantidad de triples de x1 con x2 y se quedará con el que mayor cantidad de triples tenga. Luego se compara a x1 con el resto de elementos xs. En el caso de que x1 no cumpla el criterio se compara entonces a x2 con xs, (el resto de elementos de la lista).

Implementación

Hemos definido ya algunas funciones recursivas en la sección de **Análisis**.
Ahora nos interesa ejecutar las funciones en Haskell y ver que resultados nos retorna.
Veamos entonces dos ejemplos con sus respectivos resultados. En particular estudiaremos las funciones **`leadersMap y king3.`**

- Implementación de **`leadersMap;`**
  si ejecutamos esta función en Haskell finalmente arroja por resultado:

```
["James Jarden","Luis Scola","Manu Ginobili","Stephen
Curry","Andrew Wiggins","Draymond Green","Kobe
Bryant","Trevor Ariza","Damian Lillard","Jimmy Butler","Jhon
Wall","Kevin Love","Anthony Davis","DeMarcus Cousins","Paul
Millsap","Timofey Mozgov","Russell Westbrook","LeBron
James","Pau Gasol","Terrence Ross","Zach LaVine","Avery
Bradley","Chris Paul","Kawhi Leonard","Brandon
Knight","Dwyane Wade","Brandon Bass","Jeremy Lin","Klay
Thompson"]
```

- Implementación de **`king3;`**
  Finalmente ejecutando esta función sobre Haskell tenemos el siguiente resultado:

```
[("Stephen Curry",80,2618,1900,653,1341,286,646)]
```

Link del código del proyecto: https://replit.com/join/ncdtnhbvwr-kevin-leonelleo