

Coding Dojo

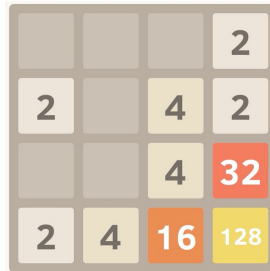
Implementación de 2048

Comunidad Haskell San Simón

Octubre 2016

1. 2048

2048 es un juego de computadora inventado por Gabriel Cirulli.¹ El tablero principal se puede ver en la figura 1. Es un tablero de 4 x 4 casillas, cada una de las cuales o está vacía o contiene un número que es una potencia de 2.² El objetivo del juego es acumular los valores hasta llegar al 2048.



			2
2		4	2
		4	32
2	4	16	128

Figura 1: Tablero de juego 2048

El tablero inicialmente tiene dos números aleatoriamente colocados en las casillas. El resto de las casillas están vacías. Se pueden hacer cuatro movimientos: izquierda, derecha, arriba o abajo. Un movimiento en cualquier sentido permitido provoca que todas las casillas con números se muevan en el sentido marcado. Ver por ejemplo el movimiento hacia abajo desde el tablero de la figura 1 lleva al tablero que se muestra en la figura 2.

Al recorrerse las casillas en la dirección indicada, cuando se encuentran dos números adyacentes iguales, se los suma. Esto explica:

1. El 4 en la casilla de la segunda fila, cuarta columna.

¹Se puede ver más detalles históricos del juego en [https://en.wikipedia.org/wiki/2048_\(video_game\)](https://en.wikipedia.org/wiki/2048_(video_game)).

²Las potencias de 2 desde 2^1 (2) hasta 2^{11} (2048).

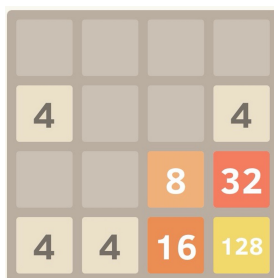


Figura 2: Tablero luego de mover hacia abajo

2. El 8 en la casilla de la tercera fila, tercera columna.
3. El 4 en la casilla de la cuarta fila, primera columna.

El número 4 en la casilla de la segunda fila, primera columna, es el introducido aleatoriamente por el juego para la próxima jugada.

De esta forma se puede ir acumulando valores hasta lograr el objetivo del juego o quedarse sin casillas libres. Si se logra el objetivo del juego se gana. Si se queda sin casillas libres sin llegar al valor 2048 se pierde.

Para familiarizarse con las reglas se puede jugar en línea en el URL <http://2048game.com>.

2. Implementación

El juego en versión consola fue implementado en Haskell por Gregor Ulm el 2014.³ La figura 3 muestra la interacción con el juego en consola. Se usan las teclas WASD para subir, izquierda, bajar y derecha respectivamente.

```

0  0  0  0
0  0  0  2
0  0  0  0
2  4  8  16

```

Figura 3: Juego 2048 en consola

La versión original fue modificada con fines educativos por miembros de la Comunidad Haskell San Simón.

Para la implementación empezamos con un módulo Haskell que implementa la estructura principal del juego (debes tener el archivo `H2048Inicial.hs`). Siguiendo esta guía completaremos el código del juego a medida que reforzamos los conocimientos de nuestro lenguaje favorito.

³Puedes descargarlo de internet en <https://github.com/gregorulm/h2048>.

2.1. Preliminares

Representaremos al tablero en una lista de listas:

```
type Grid = [ [ Int ] ]
```

Trabajaremos en las dos partes importantes del código: preparar el tablero para el juego y ejecutar las jugadas.

2.2. Preparar el tablero del juego

La preparación del tablero del juego se realiza en la función `initialGrid` que se muestra en el segmento a continuación:

```
initialGrid :: IO Grid
initialGrid = do
    grid1 <- addTile grid0
    grid2 <- addTile grid1
    return grid2

grid0 :: Grid
grid0 = replicate 4 [0, 0, 0, 0]

addTile :: Grid -> IO Grid
addTile grid = do
    let candidateTiles = getZeroes grid
    pickedTile <- choose candidateTiles
    value <- choose [2,2,2,2,2,2,2,2,4]
    let newGrid = setTile pickedTile value grid
    return newGrid
```

Se basa en el tablero inicial (función `grid0`). A partir del mismo se añaden dos casillas con números en posiciones aleatorias usando la función `addTile`. Necesitamos completar la función `addTile`. La lógica de la función es: a partir del tablero calculamos las posiciones de casillas vacías y elegimos una posición aleatoriamente (`choose candidateTiles`). Luego elegimos aleatoriamente un número que vamos a poner en la posición elegida (`choose [2,2,2...,4]`). Finalmente usando la función `setTile` agregamos al tablero el valor en la posición elegida.

2.2.1. Calcular las posiciones vacías de un tablero

Para ello primeramente debemos implementar la función `getZeroes`. Esta función necesita devolvernos todas las casillas del tablero que están vacías. Es decir su tipo es:

```
getZeroes :: Grid -> [(Int, Int)]
```

Podemos probar los siguientes casos con la función que implementemos:

```

*H2048> getZeroes [ [0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0] ]
[(0,0),(0,1),(0,2),(0,3),(1,0),(1,1),(1,2),(1,3),(2,0),(2,1),(2,2),
(2,3),(3,0),(3,1),(3,2),(3,3)]

*H2048> getZeroes [ [0,0,2,0], [0,0,0,0], [0,0,0,0], [2,0,0,0] ]
[(0,0),(0,1),(0,3),(1,0),(1,1),(1,2),(1,3),(2,0),(2,1),(2,2),(2,3),
(3,1),(3,2),(3,3)]

*H2048> getZeroes [ [0,0,0,2], [0,0,2,4], [8,16,32,64], [128,256,512,1024] ]
[(0,0),(0,1),(0,2),(1,0),(1,1)]

*H2048> getZeroes [ [2,4,2,4], [4,8,4,8], [2,4,2,4], [16,8,16,8] ]
[]

```

Aunque sabemos que el último caso no debería darse porque sobre ese tablero ya no se puede jugar ya que está lleno.

2.2.2. Agregar un número al tablero

A continuación necesitamos implementar la función `setTile`:

```
setTile :: (Int, Int) -> Int -> Grid -> Grid
```

Es decir, la función debe insertar en la posición dada por el par, el número que viene como segundo parámetro en el tablero. Es decir:

```

*H2048Initial> setTile (0,0) 2 [ [0,0,0,0], [0,0,0,0], [0,0,0,0], [0,0,0,0] ]
[[2,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]

*H2048Initial> setTile (1,2) 4 [ [0,0,0,2], [0,0,0,4], [8,16,32,64],
[128,256,512,1024] ]
[[0,0,0,2],[0,0,4,4],[8,16,32,64],[128,256,512,1024]]

```

No necesitaremos validar los datos de entrada porque sabemos que las posiciones son correctas (son calculadas a partir de las posiciones libres del tablero) y el número es 0 2 o 4 (sale de la lista `[2,2,2,...,4]`).

2.3. El ciclo de juego

El ciclo principal del juego por supuesto se centra en la idea de que o bien se han terminado las casillas libres, en cuyo caso no se ha logrado el objetivo de construir el 2048, o bien hay casillas libres todavía. Si hay casillas libres, hay que fijarse si ya se ha llegado al objetivo en cuyo caso detener el juego indicando que el jugador ha logrado el objetivo. Si no se ha llegado al objetivo, atender el próximo movimiento del jugador (en la función `getNewGrid`) y añadir un nuevo número en una casilla libre aleatoriamente (como ya elaboramos anteriormente se realiza en la función `addTile`).

El código de esta función es:

```

gameLoop :: Grid -> IO ()
gameLoop grid
  | areMovesPossible grid = do
    printGrid grid
    if check2048 grid
    then putStrLn "You won!"
    else do newGrid <- getNewGrid grid
           if grid /= newGrid
           then do new <- addTile newGrid
                  gameLoop new
           else gameLoop grid
  | otherwise = do
    printGrid grid
    putStrLn "Game over"

```

2.3.1. Ganamos el juego?

Como calentamiento para esta fase de la implementación podemos empezar escribiendo el código de la función `check2048`:

```
check2048 :: Grid -> Bool
```

Es decir toma un tablero como entrada y produce un valor booleano que indica si pudimos acumular el valor objetivo 2048.

```

*H2048Initial> check2048 [ [0,0,0,2], [0,0,0,4], [8,16,32,64], [128,256,512,1024] ]
False
*H2048Initial> check2048 [ [0,0,0,2], [0,0,0,4], [8,16,32,64], [128,256,512,2048] ]
True

```

2.3.2. Realizar la movida

Una función que requiere un poco más de análisis es la que usamos para realizar la movida `applyMove`. El código de la misma es:

```

data Move = Up | Down | Left | Right

applyMove :: Move -> Grid -> Grid
applyMove Left  = map merge
applyMove Right = map (reverse . merge . reverse)
applyMove Up    = transpose . applyMove Left  . transpose
applyMove Down  = transpose . applyMove Right . transpose

```

Como se puede ver cada movida es básicamente una variación de la operación `merge`. Por ejemplo para aplicar la movida a derecha, primero invertimos el tablero (horizontalmente) para poder aplicar una movida a izquierda, luego de la misma invertimos el tablero nuevamente para que las casillas queden en su posición final. Lo mismo por ejemplo con una operación de mover arriba o abajo,

primeramente invertimos el tablero verticalmente (calculando la transpuesta) y luego aplicando la operación a izquierda o derecha que corresponda. Una especificación muy ingeniosa y elegante de la operación de mover!

Nos queda como tarea implementar la operación de `merge`:

```
merge :: [Int] -> [Int]
```

Es una función que acumula los valores en una fila, hacia la izquierda:

```
*H2048Initial> merge [0,0,2,4]
[2,4,0,0]
*H2048Initial> merge [0,2,2,4]
[4,4,0,0]
*H2048Initial> merge [2,2,2,4]
[4,2,4,0]
*H2048Initial> merge [2,2,4,4]
[4,8,0,0]
*H2048Initial> merge [2,4,4,4]
[2,8,4,0]
```

2.3.3. Finalizando: hay casillas libres?

```
areMovesPossible :: Grid -> Bool
```

```
*H2048Initial> areMovesPossible grid0
True
*H2048Initial> areMovesPossible [[2,4,2,4],[4,8,4,8],[2,4,2,4],[4,8,4,8]]
False
*H2048Initial> areMovesPossible [[2,4,2,4],[8,4,8,4],[2,4,2,4],[4,8,4,8]]
True
```

3. Puntos extra

. calcular los puntos del juego