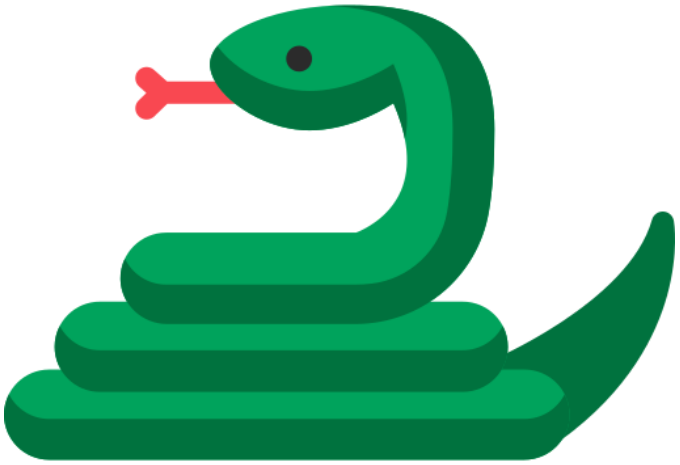
Memoria mini-proyecto.



Nombre de los participantes: Antonio Saborido Campos, Alejandro de la Flor García.
Uvus de los participantes: antsabcam, aledegar1
Correos: antonio.saborido01@gmail.com, alejandrodelaflor01@gmail.com

Índice.

Índice. **2**
Temática elegida y problema a resolver. **3**
Estructura del código. **4**
Funciones básicas de prelude y Data.List. 4
Funciones recursivas. 4
Funciones por patrones. 5
Uso de guardas. 5
Uso listas de comprensión. 6
Usos de funciones de orden superior. 6
Creación de módulo. 7
Creación de tipos de datos y uso de ellos. 7
Uso de tipos de datos vistos en clase. 8
Cómo compilar y usar el programa. **8**

Temática elegida y problema a resolver.

La temática elegida es una recreación del clásico juego de la serpiente que tiene que comer manzanas evitando chocarse tanto con los bordes como con los obstáculos intermedios. Para ello, hemos decidido generar un tablero con diferentes estados definidos, cada vez que se pulsa una tecla de movimiento se genera un nuevo estado. Para ello hemos usado un función que maneja cada evento posible y verifica si en el nuevo estado se ha tocado alguno de los bloques prohibidos, es decir, los bloques del borde o los bloques que hacen de obstáculo, o bien si se avanza a una casilla permitida o se come la manzana.

El juego consta de dos dificultades, la dificultad “fácil” es aquella que se activa al usar el juego con las teclas “←”, “↑”, “→” y “↓”, en este modo los bloques intermedios permanecen en una posición fija desde el inicio hasta que se come la manzana, sin embargo el modo “difícil” se activa al usar el juego con las teclas “1”, “5”, “3” y “2”, la diferencia con el modo “fácil” es que en cada movimiento los bloques intermedios se generan en una nueva posición aleatoria.

En cuanto se toca o bien un bloque que forma el borde o bien un bloque intermedio la serpiente queda encerrada en una cárcel que representa que se ha perdido, para salir de la cárcel es necesario recargar la página y volver a empezar el juego. Cuando se acumulan 10 manzanas sin perder la serpiente es encerrada en una cárcel de color dorado que representa haber ganado el juego.

Estructura del código.

El proyecto está estructurado en dos archivos, el archivo principal es “JuegoMain.hs”, es donde se sitúan todos los parámetros iniciales del juego así como la funciones Maneja Evento y los textos que aparecen alrededor de la zona jugable.

El segundo archivo es “PintaFunciones.hs” dónde se sitúan aquellas funciones con las que se pinta el tablero.

Funciones básicas de prelude y Data.List.

Uso de fromIntegral:

```
pintaNegro :: Int -> Int -> Picture
pintaNegro n m = translated (x) (y) (solidRectangle x y)
  where x = fromIntegral m
        y = fromIntegral n
```

La función fromIntegral la utilizamos cuando necesitamos convertir un número entero a otro tipo numérico, como convertir un entero a un número de punto flotante.

Uso de verificadores y operadores or:

```
manejaEvento :: Event -> Estado -> Estado
manejaEvento (KeyPress "Right") (n,m,i,j,((a1, a2),(b1, b2), (c1, c2),(d1, d2)),((e1, e2),(f1, f2), (g1, g2), (
  | (a1+1, a2) == (e1,e2) || (a1+1, a2) == (f1,f2) || (a1+1, a2) == (g1,g2) || (a1+1, a2) == (h1,h2) || (a1+1,
  | (a1+1, a2) == (l1,l2) || (a1+1, a2) == (l3,l4) || (a1+1, a2) == (l13,l14) || (a1+1, a2) == (l15,l16) = pier
  | (a1+1, a2) == (r1,r2) || (a1+1, a2) == (r3,r4) || (a1+1, a2) == (r5,r6) || (a1+1, a2) == (r7,r8) || (a1+1,
  | (a1+1, a2) == (i,j) = (n,m,(unsafePerformIO((randomRIO(-4, 4))::IO Int)),(unsafePerformIO((randomRIO(0, -4)
  | otherwise = (n,m,i,j,((a1+1, a2),(a1, a2), (b1, b2), (c1, c2)),((e1, e2),(f1, f2), (g1, g2), (h1, h2), (i1,
  where
    pierdes = (n,m,0,0,((-10, 0),(-10, 0), (-10, 0), (-10, 0)),((0, 0),(0, 0), (0, 0), (0, 0), (0, 0)), [(r1,r2
```

Funciones recursivas.

Ejemplo 1: Función recursiva:

```
pintaBloqueBorde :: [Punto] -> Picture
pintaBloqueBorde [] = blank
pintaBloqueBorde ((f,c):ps) = (translated x y (colored black (solidCircle (0.1))) & pintaCuadradoAzul x y)   & pintaBloqueBorde ps
  where x = fromIntegral f
        y = fromIntegral c

pintaCuadradoAzul :: Double -> Double -> Picture
pintaCuadradoAzul x y = translated x y (colored blue (solidRectangle 0.9 0.9))
```

Ejemplo 2: Función recursiva:

```
repiteNumero :: Int -> Int -> [Int]
repiteNumero x 0 = []
repiteNumero x n = [x] ++ repiteNumero x (n-1)
```

Esta función es utilizada como función auxiliar para generar una lista que contiene el elemento que se le indique en el primer parámetro tantas veces como se le indique en el segundo parámetro

Funciones por patrones.

Ejemplo 1: Función por patrones:

```
pintaMarcador :: [Punto] -> Int -> Picture
pintaMarcador _ 0 = blank
pintaMarcador ((f,c):ps) n = (translated x y (colored red (solidCircle (0.1))) & pintaRojo x y)   & pintaMarcador ps (n-1)
  where x = fromIntegral f
        y = fromIntegral c
```

En esta función hacemos uso de patrones para definir el caso base que se da cuando el segundo parámetro vale 0. Esta función es usada para generar un marcador que cuenta el número de victorias seguidas

Ejemplo 2: Función por patrones:

```
manejaEvento (KeyPress "Right") (n,m,i,j,[(a1, a2),(b1, b2), (c1, c2)
```

```
manejaEvento (KeyPress "Left") (n,m,i,j,[(a1, a2),(b1, b2), (c1, c2)
```

En todas las funciones manejaEvento puede apreciarse el uso de patrones para detectar qué tecla se ha pulsado.

Uso de guardas.

Ejemplos: Uso de guardas:

```
manejaEvento (KeyPress "Right") (n,m,i,j,[(a1, a2),(b1, b2), (c1, c2),(d1, d2)],((e1, e2),(f1, f2), (g1, g2), (h1, h2), (i1, i2)), [(r1,r2
  | (a1+1, a2) == (e1,e2) || (a1+1, a2) == (f1,f2) || (a1+1, a2) == (g1,g2) || (a1+1, a2) == (h1,h2) || (a1+1, a2) == (i1,i2) = pierdes
  | (a1+1, a2) == (l1,l2) || (a1+1, a2) == (l3,l4) || (a1+1, a2) == (l13,l14) || (a1+1, a2) == (l15,l16) = pierdes
  | (a1+1, a2) == (r1,r2) || (a1+1, a2) == (r3,r4) || (a1+1, a2) == (r5,r6) || (a1+1, a2) == (r7,r8) || (a1+1, a2) == (r9,r10) || (a1+1, a
  | (a1+1, a2) == (w1,w2) || (a1+1, a2) == (w3,w4) || (a1+1, a2) == (w13,w14) || (a1+1, a2) == (w15,w16) = prisionDeOro
  | ((a1+1, a2) == (i,j)) && (contador == 10) = prisionDeOro
  | ((a1+1, a2) == (i,j)) && (contador < 10) = (n,m,(unsafePerformIO((randomRIO(-4, 4))::IO Int)),(unsafePerformIO((randomRIO(0, -4))::IO
  | otherwise = (n,m,i,j,[(a1+1, a2),(a1, a2), (b1, b2), (c1, c2)],((e1, e2),(f1, f2), (g1, g2), (h1, h2), (i1, i2)), [(r1,r2),(r3,r4),(r5
  where
    pierdes = (n,m,0,0,[(-10, 0),(-10, 0), (-10, 0), (-10, 0)],((0, 0),(0, 0), (0, 0), (0, 0), (0, 0)), [(r1,r2),(r3,r4),(r5,r6),(r7,r8),(
    prisionDeOro = (n,m,0,0,[(11, 0),(11, 0), (11, 0), (11, 0)],((0, 0),(0, 0), (0, 0), (0, 0), (0, 0)), [(r1,r2),(r3,r4),(r5,r6),(r7,r8),
```

Todas las funciones manejaEvento están definidas haciendo uso de las guardas para detectar con qué objeto se está chocando la serpiente y llevar a cabo la acción correspondiente en cada caso.

Estructura de las guardas:

1. Pierdes porque la serpiente ha tocado algún bloque intermedio.
2. Has perdido e intentas salir de la cárcel.
3. Pierdes porque la serpiente ha chocado con los bloques que delimitan el tablero.
4. Intentas escapar de la cárcel de oro.
5. Has ganado 10 veces y te mueve a la cárcel de oro.
6. Te comes la manzana.
7. No hay ninguna acción, simplemente se mueve la serpiente.

Uso listas de comprensión.

Ejemplo 1: Uso listas de comprensión:

```
murosPuntos = [(i,j) | (i,j) <- zip (repiteNumero (-5) (length cs)) cs] ++ [(i,j) | (i,j) <- zip (repiteNumero 6 (length cs)) cs] ++ [(i,j) | (i,j)
  where cs = [-5..6]
        zs = [6..(-5)]
repiteNumero :: Int -> Int -> [Int]
repiteNumero x 0 = []
repiteNumero x n = [x] ++ repiteNumero x (n-1)
```

murosPuntos es una lista que contiene las coordenadas donde deben colocarse los bloques que delimitan el tablero. Utiliza una función auxiliar que genera una lista con un mismo número tan larga como se le pida

Ejemplo 2: Uso listas de comprensión:

```
95      | or [(a1, a2+1) == (i,j) | (i,j) <- cs] = prisionDeOro
```

En esta guarda utilizamos una lista por compresión para verificar que al pulsar la tecla “Up” te chocas con una pared de la prisión de oro.

Usos de funciones de orden superior.

Ejemplo 1: Uso función orden superior:

```
carcelGanadora :: [(Int, Int)]
carcelGanadora = filter (/= (11, 0)) $ concatMap (\f -> map f pares) [id, incrementar, decrementar]
  where
    pares = [(12, 0), (10, 0), (11, 0)]
    incrementar (x, y) = (x, y + 1)
    decrementar (x, y) = (x, y - 1)
```

Ejemplo 2: Uso función orden superior:

```
pintaSerpiente :: [(Int, Int)] -> Picture
pintaSerpiente = foldr (\(f, c) acc -> translated (x f) (y c) (colored yellow (solidCircle 0.1)) & pintaCuadrado (x f) (y c) & acc) blank
  where
    x = fromIntegral
    y = fromIntegral
```

Creación de módulo.

```
module PintaFunciones
    (pintaBloqueBorde,
    pintaCuadradoAzul,
    pintaCarcel3,
    pintaManzana,
    pintaSerpiente,
    pintaCuadrado,
    pintaBloqueInter,
    pintaNegro,
    pintaMarcador,
    pintaCarcelGanador
    ) where
```

```
{-# LANGUAGE OverloadedStrings #-}
import CodeWorld
import System.Random
import System.IO.Unsafe
import PintaFunciones
```

Hemos creado el módulo PintaFunciones que como su nombre indica, pinta todas las funciones en el tablero, la serpiente, muros, bloques intermedios…

Creación de tipos de datos y uso de ellos.

```
type Punto = (Int, Int)
type BloqueInter = (Punto, Punto, Punto, Punto, Punto)

type Estado = (Int, Int, Int, Int, [Punto], BloqueInter, [Punto], [Punto],[Punto], Int, [Punto])
```

Hemos creado 3 tipos para nuestro juego.

Punto: En este tipo simplemente creamos una tupla formada por dos enteros que lo iremos usando para los tipos del tablero, el resto de tipos estará formado por tipos punto.

Bloque Intermedio: Este tipo estará formado por una tupla de 5 puntos, esto define los bloques intermedios que no se podrán tocar en el tablero.

Estado: Este tipo será una tupla de todos los elementos de nuestro tablero y que se irá actualizando con cada movimiento posible. Los elementos serán los siguientes:

- *1 y 2*: Definirán el tamaño de nuestro tablero, por lo que serán 2 enteros.
- *3 y 4*: Serán dos enteros generados aleatoriamente y definirán la posición en la que estará la manzana.

- *5*: Será una lista de puntos (en concreto 4 puntos) que definirá dónde está colocada la serpiente en cada momento.
- *6*: Será un tipo Bloque Intermedio
- *7*: Una lista de puntos que define los muros que no se pueden traspasar, es decir, una lista con los bordes de la zona jugable.
- *8*: Es la lista de puntos de los bloques que te encierran si pierdes.
- *9*: Es la lista de puntos con las coordenadas donde va situado el marcador.
- *10*: Es un contador que acumula las veces que has ganado.
- *11*: Es la lista de puntos de los bloques que te encierran si ganas.

Uso de tipos de datos vistos en clase.

Ejemplo 1:

```
numRandom1 = (unsafePerformIO((randomRIO(-4, 4))::IO Int))
numRandom2 = (unsafePerformIO((randomRIO(0, -4))::IO Int))
main = juego 20 20 i j
  where i = numRandom1
        j = numRandom2
```

Esta función hace uso de la librería Random para generar números aleatorios que utilizamos para marcar las posiciones en donde se generan los bloques intermedios.

Ejemplo 2:

```
carcelGanadora :: [(Int, Int)]
carcelGanadora = concatMap (\f -> map f pares) [id, incrementar, decrementar]
  where
    pares = [(12, 0), (10, 0), (11, 0)]
    incrementar (x, y) = (x, y + 1)
    decrementar (x, y) = (x, y - 1)
```

En esta función que usamos para generar las posiciones de los muros de la cárcel ganadora puede verse el uso de map.

Cómo compilar y usar el programa.

Para compilar el programa dentro de Visual Studio Code carga los módulos, primero abriremos el terminal y escribiremos “ghci” y posteriormente “:l JuegoMain.hs” y una vez hecho esto escribir en la consola “main” aparecerá una IP que si la abrimos en el navegador ya se podrá jugar al juego. Una vez en el navegador simplemente mover la serpiente con las dos formas de controlarla que hemos definido.