Escuela Técnica Superior de
Ingeniería Informática

Mini-proyecto: Juego de puzzle Full House

Realizado por

Jialuo Chen
jiache9 (jiache9@alum.us.es)

Hugo Villanueva Duque
hugvilduq (hugvilduq@alum.us.es)

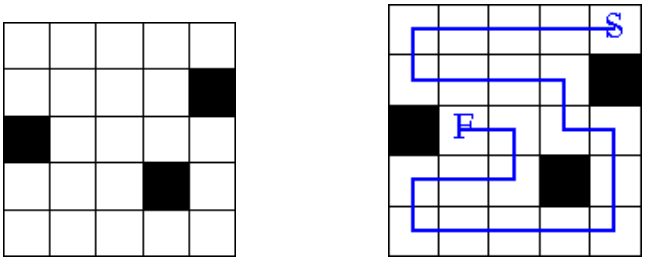
Profesor
David Solís Martín

Realizado para la asignatura
Programación Declarativa

Convocatoria de Julio curso 2022/23

Planteamiento del problema

El puzzle Full House, también conocido como Full Grid o laberinto Hamiltoniano, fue creado por el matemático Erich Friedman. El puzzle consiste en recorrer todos los cuadrados vacíos de una cuadrícula con un solo camino continuo, y este camino solo puede realizar giros ortogonales y no puede pasar por encima de sí mismo. Hay múltiples variaciones de este puzzle, pero nosotros implementaremos la versión primitiva de este, dejando la posibilidad de crear más variedades en el futuro.

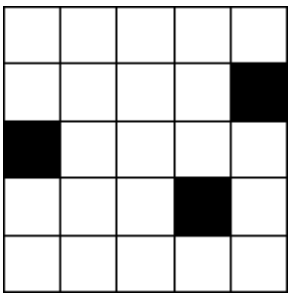


Ejemplo de un puzzle Full House - y una posible solución

En esta memoria se detalla el proceso de creación de un proyecto usando íntegramente Haskell 8.6 donde se implementa un juego ejecutable en la terminal de Haskell donde el usuario podrá indicar al sistema direcciones (arriba-derecha-abajo-izquierda) para que vaya pintando la línea de la solución a medida que avanza sobre la cuadrícula.

El juego se codifica de la siguiente manera: Se utiliza una lista de lista de caracteres para representar la cuadrícula, donde cada lista representa una fila de la cuadrícula. Las paredes o casillas no visitables se codifican con la almohadilla '#', y las casillas visitables con el punto '.' .

```
puzzle1 = ["#######",
           "#.....#",
           "#....##",
           "##....#",
           "#...#.#",
           "#.....#",
           "#######"]
```



El jugador se representa con el carácter del dólar '$', cada casilla recorrida se sustituye con la letra 'o'. El objetivo será que el jugador sustituya todos los . por o moviendo el personaje por la cuadrícula usando los movimientos disponibles.

```
puzzle1 = ["#######",
           "#o...#",
           "#oo$.##",
           "##....#",
           "#...#.#",
           "#.....#",
           "#######"]
```

Para moverse, al jugador se le pedirá que realice uno de los cuatro movimientos (escribiéndolo en la terminal) para actualizar la posición del cursor. La lista de movimientos tiene este aspecto:

```
moves :: [String]
moves = ["AR", "AB", "IZ", "DE"]
```

Estructura del código

El código está separado en 2 archivos, un archivo principal (FullHousePuzzle) con la interfaz y los niveles necesarios para ejecutar el programa, y un archivo adicional (FullHouse_funciones) que se incluye como módulo con funciones auxiliares para el correcto funcionamiento del programa principal.
Las funciones en el archivo ejecutable principal son:

level_select: Muestra una lista de niveles de laberinto disponibles y pide al usuario que seleccione uno. Luego, se ejecuta la función positionMaze.

positionMaze: Se encarga de preguntar al usuario dónde quiere comenzar en el laberinto y muestra el laberinto con el símbolo del jugador en la posición seleccionada.

playMaze: Es la función principal que se encarga de gestionar el juego en sí. Si no hay movimientos disponibles, la función verifica si el jugador ha alcanzado la condición de victoria. Si es así, se muestra un mensaje de felicitación y se vuelve a la pantalla de selección de nivel. Si no, se muestra un mensaje de error y se vuelve a la pantalla de selección de nivel. Si hay movimientos disponibles, la función muestra el laberinto actualizado con el símbolo del jugador en la nueva posición y solicita al usuario la dirección en la que desea moverse. Si el movimiento es válido, la función actualiza el laberinto con la nueva posición del jugador y llama a la función playMaze con la nueva posición. Si el movimiento no es válido, se muestra un mensaje de error y se llama a la función playMaze con la posición actual del jugador.

Para ejecutar estas funciones, se requiere del otro archivo de código que se incluye como módulo.
Las funciones en el archivo FullHouse_funciones son las siguientes:

posicion: dada una lista de listas, y una tupla (f,c) que representa la fila y columna en la que se encuentra un elemento en la lista, esta función devuelve el elemento correspondiente en dicha posición.

vecinos: Dada una lista de listas y una tupla (i,j) que representa una posición, esta función devuelve los caracteres adyacentes en la dirección arriba, abajo, izquierda y derecha.

vecinos2mov: dada una cadena de caracteres que representa los vecinos de una posición, devuelve una lista de movimientos válidos que se pueden realizar en esa posición. Los caracteres que representan movimientos válidos son '.', mientras que los caracteres que representan obstáculos son 'o' y '#'.

movableTo: dado un carácter que representa un tipo de celda en un laberinto, esta función devuelve un valor booleano que indica si la celda es transitable o no.

pos2mov: dada una lista de listas y una tupla (x,y) que representa una posición, esta función devuelve una lista de movimientos válidos que se pueden realizar desde esa posición. Los movimientos válidos son aquellos que llevan a una posición transitable en el laberinto.

pzl2moves: Dada una lista de listas que representa un laberinto, esta función devuelve una lista de movimientos posibles por cada casilla del laberinto. Los movimientos posibles se expresan en forma de una lista de movimientos válidos para cada posición.

siguientePosicion: Recibe una cadena de caracteres que representa un movimiento y una posición

en forma de coordenadas. Devuelve la siguiente posición en base al movimiento proporcionado. Si se introduce un movimiento que no se reconoce, la función devuelve un mensaje de error.

replaceMaze: Recibe una tupla con las coordenadas de la posición del laberinto y un carácter. Reemplaza el valor del laberinto en las coordenadas especificadas por el carácter suministrado. Se utiliza para marcar los lugares en los que el usuario ha estado en el laberinto. La función devuelve una nueva lista de cadenas de caracteres que representa el laberinto.

mueve: Recibe una lista de cadenas de caracteres que representa el laberinto, una cadena de caracteres que representa un movimiento y una posición en forma de coordenadas. La función devuelve una tupla que consta de una lista de cadenas de caracteres que representa el laberinto con el carácter de posición cambiado, una lista de cadenas de caracteres que representa los movimientos que se han realizado para llegar a la nueva posición y una nueva posición en forma de coordenadas.

buildTree: Utiliza un tipo de datos de árbol y una lista de tuplas de coordenadas. La función crea un árbol que representa todos los posibles caminos que se pueden tomar desde la posición de inicio hasta quedarse sin celdas. La función toma una lista de cadenas de caracteres que representa el laberinto, una tupla de coordenadas que representa la posición inicial y una lista de tuplas de coordenadas que representa las posiciones visitadas. La función devuelve un árbol.

matriz: Crea una matriz que representa el laberinto, con el valor 0 representando un muro y -1 representando una celda visitada.

isWin: Función booleana que comprueba si un puzzle está completado de manera victoriosa. Utiliza la librería Pila.hs

Conceptos requeridos para el proyecto utilizados en las funciones

Concepto requerido	Función que lo implementa	Fragmento de código
Función básica de Prelude (1): Zip	vecinos2mov	`vecinos2mov cs = [ moves!!(i) | (c,i)<-zip cs [0,1..], movableTo c]`
Función básica de Prelude (2): !!	posicion2	`posicion2 pzl (f,c) = pzl !! f !! c`

Función recursiva (1)	formatoIsWin	`formatoIsWin p`
		`  | esVacia p = vacia`
		`  | otherwise = foldr apila (formatoIsWin dp) cp`
		`  where`
		`  cp = cima p`
		`  dp = desapila p`

Función recursiva (2)	buildTree	`buildTree :: [String] -> (Int, Int) -> [(Int, Int)] -> Arbol`
		`buildTree maze pos visited`
		`  |validMoves == [] = H pos`
		`  |otherwise = N pos (map (\p -> buildTree maze p (pos : visited)) validMoves)`

Función por patrones (1)	siguientePosicion	`siguientePosicion :: (Int, Int) -> [Char] -> (Int, Int)`
		`siguientePosicion (x, y) "AR" = (x-1, y)`
		`siguientePosicion (x, y) "AB" = (x+1, y)`
		`siguientePosicion (x, y) "IZ" = (x, y-1)`
		`siguientePosicion (x, y) "DE" = (x, y+1)`
		`siguientePosicion _ _ = error "Invalid movement direction"`

Función por patrones (2)	movableTo	`movableTo cell = case cell of`
		`  'o' -> False`
		`  '#' -> False`
		`  '.' -> True`
		`  _ -> error "Invalid cell type"`

Uso de guardas	pos2mov	`pos2mov pzl (x,y)`

(1)		`| posicion pzl (x,y) == '#' = []` `| otherwise = vecinos2mov ls_vecinos`

Uso de guardas (2)	formatoIsWin	`formatoIsWin p`
		`| esVacia p = vacia`
		`| otherwise = foldr apila (formatoIsWin dp) cp`

Uso de case of (1)	movableTo	`movableTo cell = case cell of`
		`'o' -> False`
		`'#' -> False`
		`'.' -> True`
		`_ -> error "Invalid cell type"`

Uso de case of (2)	siguientePosicion	`siguientePosicion mov (x,y) = case mov of`
		`AR" -> (x-1, y)`
		`AB" -> (x+1, y)`
		`IZ" -> (x, y-1)`
		`DE" -> (x, y+1)`
		`-> error "Invalid movement direction"`

Lista por comprensión (1)	vecinos2mov	`vecinos2mov cs = [ moves!!(i) | (c,i)<-zip cs [0,1..], movableTo c]`
Lista por comprensión (2)	pzl2moves	`pzl2moves css = [ [ pos2mov css (i,j) | (c,j)<-zip cs [0..length cs], j>0 && j<tamanyo_j]| (cs,i) <- zip css [0,1..], i>0 && i<tamanyo_i]`
Orden superior (1): Foldr	vecinos	`vecinos pzl (i,j) = foldr (\c acc -> (posicion pzl c):acc) [] coords`

Orden superior (2): Map	builldTree	`buildTree maze pos visited`
		`|validMoves == [] = H pos`
		`|otherwise = N pos (map (\p -> buildTree maze p (pos : visited)) validMoves)`

Tipo algebraico (1)	Pos	`type Pos = (Int, Int)`
Tipo algebraico (2)	Arbol	`data Arbol = N Pos [Arbol] | H Pos deriving (Show)`
Tipo de dato abstracto o librería (1):	matriz	`type Matriz = Array (Int,Int) Int` `matriz pzl = array ((1,1),(m,n)) [ ((i,j),`

Matriz		`condicion (i-1) (j-1)) | i<-[1..m], j<-[1..n]]`

Tipo de dato abstracto o librería (2): Pila	isWin	`isWin pzl = (res 0 p1) <= 1`
		`  where`
		`  cf = conversorFormato pzl`
		`  p1 = formatoIsWin cf`
		`  res contador p`
		`  | esVacia p = contador`
		`  | cp == '.' = res (contador+1) dp`
		`  | otherwise = res contador dp`
		`  where cp = cima p`
		`  dp = desapila p`

Compilación y ejemplo de uso

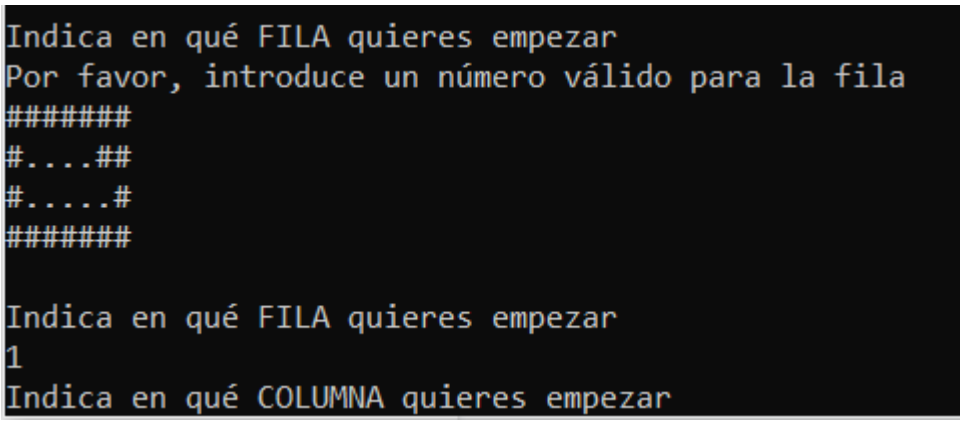
Para crear el ejecutable, el usuario situarse en la carpeta src con los 3 archivos de código, abrir una terminal y ejecutar el siguiente comando:

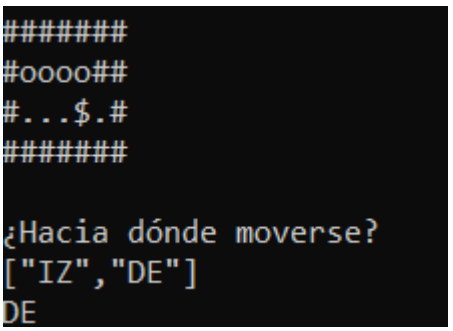
```
ghc --make FullHousePuzzle.hs
```

El archivo ejecutable se llamará FullHousePuzzle y aparecerá en la carpeta actual.

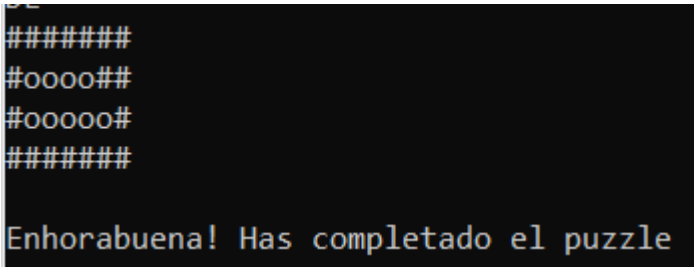
Al abrir el archivo, se presentará al usuario con la pantalla de selección de niveles:

Tras introducir “1” en la terminal para jugar el primer nivel, se nos presenta el puzle, y se nos pregunta por la casilla donde se desea empezar. Se introduce primero un número para la fila, y después otro para la columna





El resto del nivel consiste en mover el jugador, representado con el dólar $, introduciendo una de las direcciones de movimiento ofrecidas en la terminal.



Si el jugador consigue completar el puzzle, se le indicará con un mensaje de reconocimiento. Si vuelve a pulsar intro, se le devuelve a la pantalla de selección de nivel. Lo mismo ocurre si se queda atrapado sin movimientos.

Fuentes y bibliografía

Friedman, E. (n.d.). Puzzles originales de Erich Friedman. Retrieved from https://erich-friedman.github.io/puzzle/full/

Documentación de Haskell: https://www.haskell.org/documentation/

Inspirado en el examen de convocatoria de la asignatura Programación Declarativa por el juego “laberinto”