MEMORIA PECL2 APA

Sanz Sacristán, Sergio
Zamorano Ortega, Álvaro

Índice

FUNCIONES

El objetivo principal de esta práctica es realizar un programa de forma funcional en el lenguaje de programación Scala. Por lo tanto, para llevar a cabo nuestro programa, hemos precisado de una gran cantidad de funciones, las cuales son llamadas en numerosas ocasiones por otras funciones para así conseguir la condición de funcionalidad. A continuación, nos disponemos a tratar de explicar de forma concisa y breve el conjunto de las funciones utilizadas en nuestro programa.

main

```
def main(args: Array[String]) {
  println("------------------BIENVENIDO A 16384------------------")
  val vidas = 3
  val nivel = pedirNivel()
  val modo = pedirModo()
  val tablero = generarNivel(nivel)
  val puntaje = 0
  val puntMax = cargarPuntMax()
  println("·SCORE MAXIMO: " + puntMax + "\n")
  jugar(tablero, nivel, vidas, puntaje, modo, true, puntMax)
}
```

La función main() es la que inicia el juego. En ella lo que hacemos es inicializarlo correctamente, introduciéndole el número de vidas inicial, el nivel y el modo en el que se va a jugar, iniciamos el puntaje y la matriz inicial, y cargamos la puntuación máxima. Todo esto llamando a funciones auxiliares que nos devuelven los datos deseados. Finalmente llamamos a la función jugar a la que le pasamos los datos mencionados anteriormente y los necesarios para que la función se ejecute correctamente.

guardarPuntMax

```
def guardarPuntMax(score: Int) {
  val file = new File("./scoreMaximo.txt")
  val bw = new BufferedWriter(new FileWriter(file))
  bw.write("" + score)
  bw.close()
}
```

A esta función le pasamos como parámetro un entero score que es la puntuación máxima alcanzada en cualquier momento del juego. El cometido de esta función es guardar esta puntuación en un archivo txt llamado scoreMaximo.txt.

cargarPuntMax

```
def cargarPuntMax(): Int = {
  if (scala.reflect.io.File("./scoreMaximo.txt").exists) {
    val rd = Source.fromFile("./scoreMaximo.txt").getLines.mkString
    val score = rd.toInt
    score
  } else {
    0
  }
}
```

El cometido de esta función es devolver un dato de tipo entero, el cual coge del archivo scoreMaximo.txt, donde se encuentra la puntuación máxima obtenida en el juego.

pedirNivel

```
def pedirNivel(): Int = {
  print("-Introduzca el nivel en el que desea jugar (1, 2, 3, 4): ")
  val input = readLine.toLowerCase.trim
  input match {
      case _ => try {
        val num = input.toInt
        if(num!=1 && num!=2 && num!=3 && num!=4) pedirNivel
        else num
      } catch {
        case _: NumberFormatException =>
          println("Incorrecto\n")
          pedirNivel()
      }
  }
}
```

Esta función devuelve como dato un entero el cual es leído por teclado. Se le solicita al usuario que introduzca el nivel de dificultad con el que desea jugar y lee lo que el usuario solicita.

pedirModo

```
def pedirModo(): Int = {
  print("-Introduzca el modo en el que desea jugar (1-Manual, 2-Automatico): ")
  val input = readLine.toLowerCase.trim
  input match {
      case _ => try {
        val num = input.toInt
        if(num!=1 && num!=2) pedirModo
        else num
      } catch {
        case _: NumberFormatException =>
          println("Incorrecto\n")
          pedirModo()
      }
  }
}
```

Esta función devuelve como dato un entero el cual es leído por teclado. Se le solicita al usuario que introduzca el modo en el que desea jugar y lee lo que el usuario solicita.

generarNivel

```
def generarNivel(nivel: Int): List[Int] = {
  if (nivel == 1) generarTablero(16)
  else if (nivel == 2) generarTablero(81)
  else if (nivel == 3) generarTablero(196)
  else if (nivel == 4) generarTablero(289)
  else {
    println("VALOR INCORRECTO")
    generarNivel(pedirNivel())
  }
}
```

A esta función le metemos como parámetro un entero “nivel” que será el nivel de dificultad seleccionado por el usuario. Dependiendo del nivel seleccionado llamamos a la función generarTablero con los datos correspondientes a cada nivel. Esta función devolverá el tablero (List[Int]) correspondiente.

jugar

```
def jugar(tablero: List[Int], nivel: Int, vidas: Int, puntos: Int, modo: Int, primero: Boolean, puntMaximo: Int) {
  val inicial = rellenar(primero, nivel, tablero)
  imprimirMatriz(inicial)
  printMejorMovimiento(inicial, puntos)
  val mov = movManualAutomatico(modo, inicial)
  val juega = hacerMov(inicial, mov)
  val puntaje = calcPuntos(inicial, mov, puntos)
  if (finNiv(nivel, juega)) {
    println("\n\n**VIDA AGOTADA**\n\n\n·Número de vidas: " + (vidas - 1) + "\n·Puntos: " + puntos + "\n·Puntuacion Maxima: " + puntMaximo)
    imprimirMatriz(juega)
    if (muerto(vidas - 1)) println("\n\nGAME OVER")
    else jugar(generarNivel(nivel), nivel, vidas - 1, puntaje, modo, true, puntMaximo)
  } else {
    println("·Número de vidas: " + vidas + "\n·Puntos: " + puntos + "\n·Puntuacion Maxima: " + puntMaximo + "\n")
    if(puntaje > puntMaximo) {
      guardarPuntMax(puntaje)
      jugar(juega, nivel, vidas, puntaje, modo, false, puntaje)
    }else{
      jugar(juega, nivel, vidas, puntaje, modo, false, puntMaximo)
    }
  }
}
```

Esta es una de las funciones más importantes del programa. A esta función pasamos como parámetros los datos recogidos en el main y llamamos a diversas funciones para llevar a cabo de forma correcta el desarrollo del juego. Primero rellenamos el tablero dependiendo de si es la primera vez que lo rellenamos o ya hemos realizado algún movimiento previamente. Luego mostramos el tablero y cuál será el mejor movimiento a realizar. Si está en modo automático, selecciona el movimiento a realizar, si no, pide al usuario que movimiento desea realizar, guardando este movimiento en el val mov. Hace el movimiento dependiendo del mov elegido y almacena el tablero tras realizar el movimiento en la val juega. Calcula los puntos conseguidos en el movimiento y lo almacena en la val puntaje. Tras realizar el movimiento, si tiene vidas y puede seguir realizando movimientos, entonces vuelve a llamar a jugar con los parámetros correspondientes e imprime los puntos actuales y cuál es la puntuación máxima. Si aún tiene vidas pero no puede realizar más movimientos en esa matriz, entonces pierde una vida y llama a jugar con los parámetros correspondientes restándole 1 al número de vidas, e imprimiendo el número de puntos y cuál es la puntuación máxima alcanzada. Y por último, si no puede realizar más movimientos en ese tablero y está sin vidas, se finaliza el juego y se muestra por pantalla “GAME OVER”.

movManualAutomatico

```
def movManualAutomatico(modo: Int, lista: List[Int]): Int = {
  if (modo == 1) {
    val mov = pedirMov()
    if (mov != 6 && mov != 8 && mov != 4 && mov != 2) {
      println("MOVIMIENTO INCORRECTO")
      movManualAutomatico(modo, lista)
    }
    mov
  } else movAutomatico(lista)
}
```

En esta función metemos como parámetros un entero que será el modo que ha seleccionado el usuario para jugar y una lista que será el tablero de juego. Si el modo es 1 entonces está en modo manual. Por lo tanto, llama a la función pedirMov para pedir al usuario por pantalla un movimiento a realizar. Si está en modo automático entonces llama a la función movAutomatico que devolverá un movimiento de forma aleatoria (mejor movimiento). Esta función entonces devuelve un entero que será el movimiento a realizar posteriormente.

pedirMov

```
def pedirMov(): Int = {
  print("\n-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): ")
  val input = readLine.toLowerCase.trim
  input match {
      case _ => try {
        val num = input.toInt
        if(num!=2 && num!=4 && num!=8 && num!=6) pedirMov()
        else num
      } catch {
        case _: NumberFormatException =>
          println("Incorrecto\n")
          pedirMov()
      }
  }
}
```

Esta función devuelve un entero, el cual corresponderá con el que el usuario introduzca por teclado para realizar el movimiento que deseé.

movAutomatico

```
def movAutomatico(lista: List[Int]): Int = {
  val pos = mejorMovimiento(lista, 0)
  if(pos==2) generarAleatorioSemilla(List(6,4))
  else generarAleatorioSemilla(List(8,2))
}
```

A esta función la pasamos como parámetro una lista que será el tablero de juego. El movimiento a realizar será el mejor en cuanto a puntuación. En el caso de que el mejor movimiento sea hacia la izquierda, se generará aleatoriamente un número que represente el movimiento hacia la derecha o hacia la izquierda (los dos generan la misma puntuación). En caso contrario, se generará un número aleatorio que represente el movimiento hacia arriba o hacia abajo.

rellenar

```
def rellenar(primero: Boolean, nivel: Int, tablero: List[Int]): List[Int] = {
  if (primero) rellenarInicial(nivel, tablero)
  else rellenarTablero(nivel, tablero)
}
```

El objetivo de esta función es devolver una lista rellena con los valores correspondientes. Dependiendo de si el tablero se acaba de generar o de si ya van varios movimientos realizados, entonces el tablero se rellena de una forma u otra, por ello aquí lo que hacemos es dirigir el juego dependiendo del caso en el que nos encontremos hacia la forma correcta de ejecución.

muerto

```
def muerto(vidas: Int): Boolean = vidas match {
  case 0     => true
  case vidas => false
}
```

En esta función pasamos como parámetro el entero vidas. Si el número de vidas es 0 entonces devolvemos un booleano true, y si es distinto de cero es que aún le quedan vidas, por lo tanto, devolvemos el booleano false.

finNiv

```
def finNiv(nivel: Int, tablero: List[Int]): Boolean = nivel match {
  case 1 => lleno(1, tablero)
  case 2 => lleno(3, tablero)
  case 3 => lleno(5, tablero)
  case 4 => lleno(6, tablero)
}
```

En esta función comprobamos si el tablero está lleno. Dependiendo del nivel en el que se esté realizando el juego, la condición de que el tablero esté lleno será una u otra. Por lo tanto, lo que hacemos es llamar a la función lleno de la forma correcta dependiendo del nivel en que se esté jugando. Finalmente devolvemos el booleano que nos de la función lleno.

lleno

```
def lleno(numValores: Int, tablero: List[Int]): Boolean = {
  if (numValores > (numCeros(tablero, 0))) true
  else false
}
```

A esta función le pasamos un entero numValores y una lista tablero. Si el numValores es mayor que el número de ceros en el tablero entonces el tablero estará lleno, por lo tanto, devolvemos el booleano true. Si esto no se cumple es que no está lleno el tablero y devolvemos el booleano false.

numCeros

```
def numCeros(tablero: List[Int], cont: Int): Int = {
  val valor = tablero.head
  if (valor == 0) {
    if (tablero.length == 1) cont + 1
    else numCeros(tablero.tail, cont + 1)
  } else {
    if (tablero.length == 1) cont
    else numCeros(tablero.tail, cont)
  }
}
```

A esta función le pasamos un entero cont y una lista tablero. El objetivo de esta función es devolver un entero que corresponda con el número de ceros que hay en el tablero, por ello, añadimos el contador y vamos llamando recursivamente a la función. Finalmente, nos devolverá el número de ceros totales en el tablero.

hacerMov

```
def hacerMov(tablero: List[Int], mov: Int): List[Int] = mov match {
  case 8 => irArriba(tablero)
  case 4 => irIzquierda(tablero)
  case 2 => irAbajo(tablero)
  case 6 => irDerecha(tablero)
}
```

A esta función la pasamos la lista tablero y un entero mov. Su función es, dependiendo del movimiento que se le pasa como parámetro, llamar a la función que le corresponde cada movimiento. Finalmente, devolvemos la lista con el movimiento ya realizado tras llamar a la función correspondiente.

calcPuntos

```
def calcPuntos(tablero: List[Int], mov: Int, puntos: Int): Int = mov match {
  case 8 => puntArriba(tablero, puntos)
  case 4 => puntIzquierda(tablero, puntos)
  case 2 => puntAbajo(tablero, puntos)
  case 6 => puntDerecha(tablero, puntos)
}
```

Esta función es similar a la anterior, pero en vez de llamar a la función que realiza los movimientos, llamamos a la función que calcula la puntuación dependiendo del movimiento que se haya realizado. Finalmente, devolvemos un entero que serán los puntos obtenidos tras ese movimiento.

rellenarTablero

```
def rellenarTablero(nivel: Int, tablero: List[Int]): List[Int] = nivel match {
  case 1 => rellenarTablero(tablero, 1, 1)
  case 2 => rellenarTablero(tablero, 3, 2)
  case 3 => rellenarTablero(tablero, 5, 3)
  case 4 => rellenarTablero(tablero, 6, 4)
}
```

En el juego, dependiendo del nivel en el que te encuentres, se rellena el tablero de una forma u otra. Esta función, dependiendo del nivel en el que se esté jugando, llama a la función de rellenar el tablero con los parámetros correspondientes a cada nivel. Finalmente nos devolverá la lista ya rellena.

rellenarInicial

```
def rellenarInicial(nivel: Int, tablero: List[Int]): List[Int] = nivel match {
  case 1 => rellenarTablero(tablero, 2, 1)
  case 2 => rellenarTablero(tablero, 4, 2)
  case 3 => rellenarTablero(tablero, 6, 3)
  case 4 => rellenarTablero(tablero, 6, 4)
}
```

Esta función realiza el mismo trabajo que la anterior pero con los datos correspondientes para rellenar el tablero por primera vez, ya que, según el enunciado, la primera vez que hay que rellenar el tablero es distinta al resto de veces.

valorCas

```
def valorCas(matriz: List[Int], casilla: Int): Int = {
  if (casilla == 1) matriz.head
  else valorCas(matriz.tail, casilla - 1)
}
```

A esta función le pasamos una lista matriz y un entero casilla. El objetivo es devolver un entero con el valor del tablero en la posición correspondiente a la casilla pedida.

generarAleatorioSemilla

```
def generarAleatorioSemilla(lista: List[Int]): Int = {
  val random = util.Random
  val pos = random.nextInt(lista.length) + 1
  val numero = valorCas(lista, pos)
  return numero
}
```

A esta función le metemos como parámetro una lista. Esta función toma una posición aleatoria de la lista y devuelve un entero con el valor correspondiente a esa posición de la lista.

generarAleatorioTablero

```
def generarAleatorioTablero(tam: Int): Int = {
  val random = util.Random
  val pos = random.nextInt(tam) + 1
  return pos
}
```

A esta función le pasamos como parámetro un entero tam y lo que hace es devolvernos un entero aleatorio entre el 1 y el tam introducido como parámetro.

generarTablero

```
def generarTablero(tam: Int): List[Int] = {
  if (tam == 0) Nil
  else 0 :: generarTablero(tam - 1)
}
```

A esta función le pasamos como parámetro un entero tam y nos devuelve una lista de ceros del tamaño introducido como parámetro.

ponerTablero

```
def ponerTablero(casilla: Int, valor: Int, matriz: List[Int]): List[Int] = {
  if (matriz.isEmpty) Nil
  else if (casilla == 1) valor :: matriz.tail
  else matriz.head :: ponerTablero(casilla - 1, valor, matriz.tail)
}
```

A esta función le pasamos como parámetros dos enteros, casilla y valor, y una lista matriz. El objetivo de esta función es introducir en la posición casilla de matriz el valor dado, devolviéndonos la lista con el cambio introducido.

listaAleatoriosDificultad

```
def listaAleatoriosDificultad(dificultad: Int): List[Int] = dificultad match {
  case 1 => List(2)
  case 2 => List(2, 4)
  case _ => List(2, 4, 8)
}
```

A esta función le pasamos como parámetro un entero dificultad. Dependiendo del nivel de dificultad en el que se juegue, al rellenar el tablero se puede rellenar con distintos valores. Por ejemplo, en el nivel uno solo se va a rellenar con doses, pero en el nivel dos con doses y cuatros. Por ello, aquí lo que hacemos es devolver una lista con los valores posibles con los que se puede rellenar el tablero dependiendo de la dificultad en la que se esté jugando.

rellenarTablero

```
def rellenarTablero(matriz: List[Int], numCasillas: Int, dificultad: Int): List[Int] = {
  if (numCasillas == 0) matriz
  else if (numCasillas > matriz.length) matriz
  else {
    val casilla = generarAleatorioTablero(matriz.length)
    val valor = generarAleatorioSemilla(listaAleatoriosDificultad(dificultad))
    val valor_cas = valorCas(matriz, casilla)
    if (valor_cas == 0) {
      rellenarTablero(ponerTablero(casilla, valor, matriz), numCasillas - 1, dificultad)
    } else {
      rellenarTablero(matriz, numCasillas, dificultad)
    }
  }
}
```

A esta función le pasamos como parámetros una lista matriz, y dos enteros, numCasillas y dificultad. Aquí lo que hacemos primero es coger una posición aleatoria de la matriz; si esa posición es cero entonces la rellenaremos con alguno de los valores correspondientes al nivel en el que estamos jugando, y llamamos a la función de forma recursiva restándole uno a el numCasillas que hay que rellenar hasta que esté sea cero. Si la posición seleccionada aleatoriamente no es cero entonces volveremos a llamar a la función de forma recursiva con los mismos valores.

sumarDerecha_coincidente

```
def sumarDerecha_coincidente(matriz: List[Int], valor_casilla: Int): List[Int] = {
  if (valor_casilla == matriz.head) {
    (valor_casilla * 2) :: matriz.tail
  } else {
    matriz.head :: sumarDerecha_coincidente(matriz.tail, valor_casilla)
  }
}
```

A esta función le pasamos la lista matriz y un entero valor_casilla. Esta función lo que hace es realizar la suma entre dos posiciones de la matriz. Lo que hace es llamarse recursivamente recorriendo la lista hasta que encuentra una posición en la que son iguales el valor de esa posición y el valor_casilla, entonces realiza la suma devolviendo la lista con la suma ya realizada.

paresIguales_D

```
def paresIguales_D(numColumnas: Int, matriz: List[Int], cont: Int, valorElemento: Int): Int = {
  val valor_ev = matriz.head
  if (valorElemento == valor_ev) {
    if (limite(matriz, numColumnas)) cont + 1
    else paresIguales_D(numColumnas, matriz.tail, cont + 1, valorElemento)
  } else if (valor_ev != 0) {
    cont
  } else {
    if (limite(matriz, numColumnas)) cont
    else paresIguales_D(numColumnas, matriz.tail, cont, valorElemento)
  }
}
```

En esta función contamos cuantos valores iguales que la posición en la que nos encontramos hay a la derecha antes de que haya otro valor distinto que no sea cero. Para ello recorremos la fila en la que nos encontramos de forma recursiva aumentando el contador en el caso de encontrar posiciones a la derecha que contengan el mismo valor. En el caso de que nos encontremos un valor cero seguiremos llamando a la función pero sin incrementar el contador.

Si el valor de una posición de la derecha es distinto de cero y del valor buscado entonces devolvemos el contador y habrá finalizado nuestra búsqueda.

contCeros

```
def contCeros(numColumnas: Int, matriz: List[Int], cont: Int): Int = {
  val valor_ev = matriz.head
  if (valor_ev == 0) {
    if (limite(matriz, numColumnas)) cont + 1
    else contCeros(numColumnas, matriz.tail, cont + 1)
  } else {
    if (limite(matriz, numColumnas)) cont
    else contCeros(numColumnas, matriz.tail, cont)
  }
}
```

A esta función le pasamos el número de columnas del tablero, el tablero y un contador. El objetivo de esta función es devolver un entero con el número de ceros que hay a la derecha de la casilla en la que nos encontramos en esa fila. Así sabemos cuántas posiciones tendremos que mover dicha casilla para realizar el movimiento.

sumarDerecha

```
def sumarDerecha(matrizOriginal: List[Int], matrizAuxiliar: List[Int], numColumnas: Int): List[Int] = {
  if (matrizOriginal.length == 0) Nil
  else if (limite(matrizOriginal, numColumnas)) matrizAuxiliar.head :: sumarDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
  else {
    val valor_cas = matrizOriginal.head
    val valor_cas_s = (matrizOriginal.tail).head
    if (matrizOriginal.head == 0 || (valor_cas_s != 0 && valor_cas_s != valor_cas))
      matrizAuxiliar.head :: sumarDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
    else {
      val num_coincidentes = paresIguales_D(numColumnas, matrizOriginal.tail, 1, matrizOriginal.head)
      if (num_coincidentes % 2 != 0) matrizAuxiliar.head :: sumarDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
      else {
        0 :: sumarDerecha(matrizOriginal.tail, sumarDerecha_coincidente(matrizAuxiliar.tail, valor_cas), numColumnas)
      }
    }
  }
}
```

El objetivo de esta función es realizar la suma de las casillas al realizar un movimiento. Para ello vamos a utilizar dos matrices, una que no cambiará y otra auxiliar que será sobre la que se realicen los cambios y las sumas. Inicialmente, la matriz auxiliar será igual a la matriz original. Vamos recorriendo la matriz de inicio a final llamando de forma recursiva a la función sumarDerecha.

Si nos encontramos en el límite del tablero entonces llamamos de nuevo a la función concatenando el valor head de la matriz auxiliar.

Si no nos encontramos en el límite pero el head de la matriz original es cero o el valor de la derecha no es ni cero ni es igual al head, entonces llamamos de la misma forma que anteriormente a la función sumarDerecha.

Si ninguno de estos casos se da, comprobamos si se puede dar el caso de que haya alguna suma que realizar. Para que se dé el caso tenemos que comprobar con la función paresIguales_D que haya alguna casilla a la derecha con el mismo valor que la casilla en la que nos encontramos. Si esto sucede y el número devuelto es par, llamamos a la función sumarDerecha metiéndole como matriz original el tail, como auxiliar la lista devuelta por la función sumarDerecha_coincidente y el número de columnas y concatenando previamente un 0. Así, la casilla en la que nos encontrábamos pasa a cero y la suma aparecerá en la matriz auxiliar.

Finalmente, le devolvemos la matriz auxiliar en la cual ya se han realizado las sumas oportunas.

limite

```
def limite(matriz: List[Int], numColumnas: Int): Boolean = {
  if ((matriz.length - 1) % numColumnas == 0) true
  else false
}
```

A esta función le pasamos como parámetros una lista matriz y un entero numColumnas. Lo que hace es devolver un booleano que te calcula si la primera casilla es el límite del tablero o no. Si está en el límite te devuelve un true, si no, un false.

moverDerecha

```
def moverDerecha(matrizOriginal: List[Int], matrizAuxiliar: List[Int], numColumnas: Int): List[Int] = {
  if (matrizOriginal.length == 0) Nil
  else if (limite(matrizOriginal, numColumnas)) {
    if (matrizOriginal.head != 0) matrizOriginal.head :: moverDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
    else matrizAuxiliar.head :: moverDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
  } else {
    val valor_cas = matrizOriginal.head
    if (valor_cas == 0) matrizAuxiliar.head :: moverDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
    else {
      val num_coincidentes = contCeros(numColumnas, matrizOriginal.tail, 0)
      if (num_coincidentes == 0) {
        matrizOriginal.head :: moverDerecha(matrizOriginal.tail, matrizAuxiliar.tail, numColumnas)
      } else {
        matrizAuxiliar.head :: moverDerecha(matrizOriginal.tail, ponerTablero(num_coincidentes, valor_cas, matrizAuxiliar.tail), numColumnas)
      }
    }
  }
}
```

El objetivo de esta función es mover todas las casillas que tengan algún valor hacia la derecha. Para ello, utilizamos dos matrices, una original, donde se encuentran todos los datos iniciales y otra auxiliar del mismo tamaño que la original pero con todo ceros, que será donde registremos los cambios a tener en cuenta.

Primero comprobamos si la casilla en la que nos encontramos está en el límite del tablero, en caso de ser así, si el valor de la casilla es cero concatenamos el head de la matriz auxiliar y llamamos recursivamente a moverDerecha con los tails correspondientes. Si la casilla es distinta de cero entonces concatenamos el head de la matriz original y llamamos de la misma forma a moverDerecha.

En caso de que no estemos en el límite, comprobamos si la casilla es cero; si es así concatenamos el head de la matriz auxiliar y llamamos a moverDerecha con los correspondientes tails. En caso de que no sea cero, contabilizamos el número de ceros que tiene a la derecha en esa fila a través de la función contCeros.

Si el número de ceros a la derecha es cero entonces concatenamos el head de la matriz original con la correspondiente llamada recursiva a moverDerecha. En cambio si el número de ceros es distinto de cero concatenamos el head auxiliar con moverDerecha, llamándola como matriz auxiliar la resultante de llamar a ponerTablero, es decir, la matriz con el movimiento correspondiente realizado. Por ejemplo, sii ha detectado dos ceros, se mueve dos posiciones a la derecha.

Finalmente, se devuelve la lista auxiliar que nos dará el tablero ya modificado.

invertir

```
def invertir(matriz: List[Int]): List[Int] = {
  if (matriz == Nil) matriz
  else invertir(matriz.tail) ::: List(matriz.head)
}
```

Esta función invierte los valores de una lista que se le pasa por parámetro.

transpuesta

```
def transpuesta(lista: List[Int]): List[Int] = {
  transpuesta_aux1(lista, 1, raiz(lista.length))
}
```

Esta función retorna la lista transpuesta de la lista pasada por parámetro

transpuesta_aux1

```
def transpuesta_aux1(lista: List[Int], posActual: Int, numColumnas: Int): List[Int] = {
  if (posActual > numColumnas) Nil
  else transpuesta_aux2(lista, numColumnas, posActual) ::: transpuesta_aux1(lista, posActual + 1, numColumnas)
}
```

Esta función cambia los valores de las filas por los valores de las columnas. Para ello, concatena cada columna cambiada con su sucesora hasta llegar a que no hay más columnas para transponer.

transpuesta_aux2

```
def transpuesta_aux2(lista: List[Int], numColumnas: Int, posActual: Int): List[Int] = {
  if (posActual > numColumnas * (numColumnas - 1)) valorCas(lista, posActual) :: Nil
  else {
    valorCas(lista, posActual) :: transpuesta_aux2(lista, numColumnas, posActual + numColumnas)
  }
}
```

Esta función retorna una lista que, pasándole por parámetros una lista y el número de columna, cambia los valores de esa columna al mismo número de fila de la lista.

irDerecha

```
def irDerecha(lista: List[Int]): List[Int] = {
  val lista_ceros = generarTablero(lista.length) //Se crea una lis
  val suma = sumarDerecha(lista, lista, raiz(lista.length)) //Se
  val mover = moverDerecha(suma, lista_ceros, raiz(lista.length))

  return mover
}
```

Esta función retorna una lista cuyos elementos se desplazan hacia la derecha, sumándose los coincidentes. Para ello, en primer lugar, se crea una lista auxiliar de ceros de tamaño de la lista a mover, después realiza la suma de aquellas casillas que tengan el mismo valor y sean adyacentes, y finalmente se desplazan las casillas hacia la derecha.

irIzquierda

```
def irIzquierda(lista: List[Int]): List[Int] = {
  val inversa = invertir(lista) //Se invierte la lista
  val mover = irDerecha(inversa) //Se realiza el movimiento hacia la derecha
  val inversa2 = invertir(mover) //Se invierte la matriz resultado

  return inversa2
}
```

Esta función retorna una lista cuyos elementos se desplazan hacia la izquierda, sumándose los coincidentes. Para ello, en primer lugar, se invierte la lista, después se llama a mover hacia la derecha (donde se suman los valores coincidentes y se realiza el movimiento), y finalmente se invierte la lista resultado.

irAbajo

```
def irAbajo(lista: List[Int]): List[Int] = {
  val trans = transpuesta(lista) //Se transpone la lista (matriz)
  val mover = irDerecha(trans) //Se realiza el movimiento hacia la derecha
  val trans2 = transpuesta(mover) //Se transpone la matriz resultado

  return trans2
}
```

Esta función retorna una lista cuyos elementos se desplazan hacia abajo, sumándose los coincidentes. Para ello, en primer lugar, se transpone la lista, después se llama a mover hacia la derecha (donde se suman los valores coincidentes y se realiza el movimiento), y finalmente se transpone la lista resultado.

irArriba

```
def irArriba(lista: List[Int]): List[Int] = {
  val trans = transpuesta(lista) //Se transpone la lista (matriz)
  val mover = irIzquierda(trans) //Se realiza el movimiento hacia la izquierda
  val trans2 = transpuesta(mover) //Se transpone la matriz resultado

  return trans2
}
```

Esta función retorna una lista cuyos elementos se desplazan hacia arriba, sumándose los coincidentes. Para ello, en primer lugar, se transpone la lista, después se llama a mover hacia la izquierda (donde se invierte la lista y se realiza el movimiento), y finalmente se transpone la lista resultado.

puntosDerecha

```
def puntosDerecha(matrizOriginal: List[Int], numColumnas: Int, puntos: Int): Int = {
  //Si el tablero está vacío
  if (matrizOriginal.length == 0) puntos
  //Si es el limite de una fila
  else if (limite(matrizOriginal, numColumnas)) puntosDerecha(matrizOriginal.tail, numColumnas, puntos)
  else {
    val valor_cas = matrizOriginal.head
    val valor_cas_s = (matrizOriginal.tail).head
    //Si el valor actual de la casilla es 0 o el adyacente a el es de distinto valor
    if (valor_cas == 0 || (valor_cas_s != 0 && valor_cas_s != valor_cas)) puntosDerecha(matrizOriginal.tail, numColumnas, puntos)
    else {
      //Calculamos el numero de casilla con mismo valor de su misma fila,siendo adyacentes
      val num_coincidentes = paresIguales_D(numColumnas, matrizOriginal.tail, 1, matrizOriginal.head)
      //Si no es par, no se suma
      if (num_coincidentes % 2 != 0) puntosDerecha(matrizOriginal.tail, numColumnas, puntos)
      //Si es par, se añaden a los puntos el doble del valor de la casilla
      else (valor_cas * 2) + puntosDerecha(matrizOriginal.tail, numColumnas, puntos)
    }
  }
}
```

Esta función retorna los puntos que se hacen al realizar el movimiento hacia la derecha. Para ello, tomando la lista que representa al tablero, se sumarán al contador de puntos aquellas casillas que tengan un número impar de casillas del mismo valor en la misma fila. Para ello, se llamará a la función num_coincidentes, y en el caso de que sea impar, se sumará al contador de puntos el valor del doble de esa casilla.

maxLista

```
def maxLista(lista: List[Int], punt_max: Int, pos: Int, pos_aux: Int): Int = {
  if (lista.isEmpty) pos
  else {
    val actual = lista.head
    if (actual >= punt_max) maxLista(lista.tail, actual, pos_aux, pos_aux + 1)
    else maxLista(lista.tail, punt_max, pos, pos_aux + 1)
  }
}
```

Esta función, que se le pasa por parámetro una lista de números, devuelve la posición de la lista que más valor tenga.

mejorMovimiento

```
def mejorMovimiento(lista: List[Int], totalPuntos: Int): Int = {
  val pos = maxLista(listaPuntajes(lista, totalPuntos), 0, 1, 1)
  return pos /*Derecha,Izquierda,Arriba,Abajo*/
}
```

Esta función devuelve un número identificativo del mejor movimiento a realizar. Esta función llama a la función *maxLista,* de modo que devuelve la posición del máximo de puntos de la lista pasada por parámetro. El 1 representa el movimiento hacia la derecha, el 2 el movimiento hacia la izquierda, el 3 el movimiento hacia arriba y el 4 el movimiento hacia abajo.

printMejorMovimiento

```
def printMejorMovimiento(lista: List[Int], totalPuntos: Int) {
  val pos = mejorMovimiento(lista, totalPuntos)
  if (pos == 1) print("\n#Mejor movimiento Derecha-\n")
  else if (pos == 2) print("\n#Mejor movimiento Izquierda-Derecha\n")
  else if (pos == 3) print("\n#Mejor movimiento Arriba\n")
  else print("\n#Mejor movimiento Abajo-Arriba\n")
}
```

Esta función muestra por pantalla el movimiento que genera más puntos sobre la lista que pasamos por parámetro.

imprimirMatriz

```
def imprimirMatriz(tablero: List[Int]): Unit = {
  val cMax = cifras(valorMax(maxList(0, tablero), raiz(tablero.length)))

  imprimirLineaSeparadora(1, raiz(tablero.length), cMax)
  imprimirSeparador(1, raiz(tablero.length), cMax)
  imprimirMatrizAux(tablero, tablero.length, cMax, 1)
}
```

Esta función imprime por pantalla una matriz de números (tablero) de forma que se salgan con los espacios correctos y con los marcadores de filas y columnas.

En primer lugar, se llama a la función auxiliar que calcula el número de cifras del mayor mayor de la matriz, para así imprimir correctamente las casillas del tablero y que estás no queden descompensadas.

En segundo lugar, se imprime por pantalla los marcadores de las columnas, pasándole como parámetro el número de columnas de el tablero y las cifras máximas que existen.

En segundo lugar, se imprime por pantalla los separadores entre el marcador de columnas y la matriz en sí.

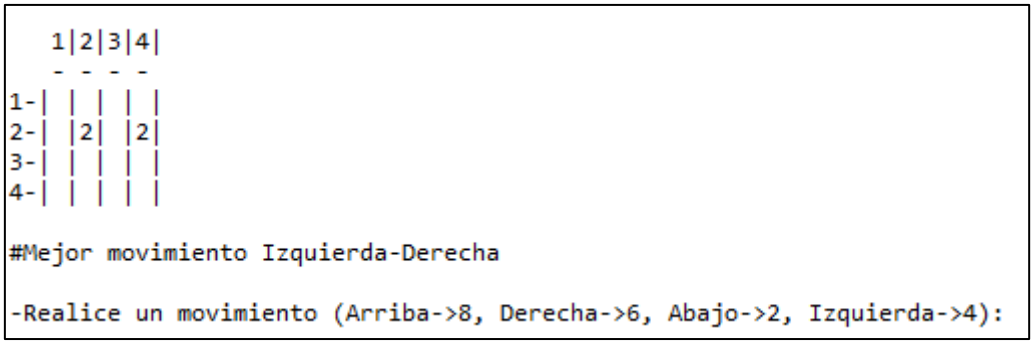
Finalnmente, se imprime por pantalla los marcadores de filas y las casillas del tablero.

PARTE DE OPTIMIZACIÓN

Para la parte de optimización, hemos añadido una nueva funcionalidad del modo manual o automático del juego. Para ello, como hemos descrito anteriormente aquellas funciones referidas al movimiento automático, el movimiento elegido será aquel que consiga la mayor puntuación. De esta forma, al conseguir los mismos puntos hacia la derecha y hacia la izquierda, y de igual modo hacia arriba y hacia abajo, se han realizado las siguientes acciones:

- En el caso de que el mejor movimiento sea hacia la izquierda, se generará un aleatorio que represente el movimiento hacia la izquierda o hacia la derecha.
- En el caso de que el mejor movimiento sea hacia abajo, se generará un aleatorio que represente el movimiento hacia abajo o hacia arriba.

En el caso de que el modo sea manual, se mostrará por pantalla el mejor movimiento a realizar antes de pedir e insertar el movimiento elegido:



FUNCIONAMIENTO DEL JUEGO

En primer lugar, introducimos los valores del nivel y el modo en el que se quiere jugar. Los valores relativos al nivel serán los números 1,2,3 o 4, y para el modo los números 1 (manual) o 2 (automático). Al introducir correctamente los valores, nos saldrá por consola el tablero inicial y la puntuación máxima conseguida en el juego:

```
------------------BIENVENIDO A 16384------------------
-Introduzca el nivel en el que desea jugar (1, 2, 3, 4): 1
-Introduzca el modo en el que desea jugar (1-Manual, 2-Automatico): 1
·SCORE MAXIMO: 6816460

   1|2|3|4|
   - - - -
1-| | |2| |
2-| | | | |
3-| | |2| |
4-| | | | |
```

Tras ello, saldrá por pantalla el mejor movimiento a realizar (el que mejor puntuación consiga), y en el caso de que el modo sea manual, se solicitará el movimiento (4 para mover hacia la izquierda, 8 para mover hacia arriba, 6 para mover hacia la derecha y 2 para mover hacia abajo). En las siguientes capturas se mostrará el tablero obtenido tras realizar cada uno de estos movimientos y la puntuación obtenida:

- Izquierda:

```
#Mejor movimiento Abajo-Arriba

-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 4
·Número de vidas: 3
·Puntos: 0
·Puntuacion Maxima: 6816460

   1|2|3|4|
   - - - -
1-|2| | | |
2-| | | | |
3-|2| | | |
4-| | | |2|
```

- Arriba:

```
#Mejor movimiento Abajo-Arriba

-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 8
·Número de vidas: 3
·Puntos: 4
·Puntuacion Maxima: 6816460

   1|2|3|4|
   - - - -
1-|4| | |2|
2-| |2| | |
3-| | | | |
4-| | | | |
```

- Derecha:

```
#Mejor movimiento Abajo-Arriba

-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 6
·Número de vidas: 3
·Puntos: 4
·Puntuacion Maxima: 6816460

   1|2|3|4|
   - - - -
1-| | |4|2|
2-| |2| |2|
3-| | | | |
4-| | | | |
```

- Abajo:

```
#Mejor movimiento Abajo-Arriba

-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 2
·Número de vidas: 3
·Puntos: 8
·Puntuacion Maxima: 6816460

   1|2|3|4|
   - - - -
1-| | |2| |
2-| | | | |
3-| | | | |
4-| |2|4|4|
```

Al llenarse el tablero (en el momento de que no se pueda rellenar con más semillas del nivel elegido), se decrementará el número de vidas en uno y se generará otro tablero:

```
    1| 2| 3| 4|
   -- -- -- --
 1-|16| 8| 4| 2|
 2-|16| 8| 4| 2|
 3-|16| 8| 4| 2|
 4-|16| 8| 4| 2|

#Mejor movimiento Abajo-Arriba

-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 4


**VIDA AGOTADA**



Nuevo tablero:

   1|2|3|4|
   - - - -
1-| | |2| |
2-| | |2| |
3-| | | | |
4-| | | | |
```

Al acabarse las tres vidas, se mostrará por pantalla el final del juego y el juego terminará su ejecución. En la siguiente captura (en modo automático), se ilustra esta acción:

```
#Mejor movimiento Abajo-Arriba
·Número de vidas: 1
·Puntos: 6516
·Puntuacion Maxima: 6816460

      1|  2|  3|  4|
    --- --- --- ---
  1-|  2| 16|  4|  2|
  2-|  4| 32|256| 64|
  3-|  8| 64| 16|  8|
  4-| 16|  2|  8|  2|

#Mejor movimiento Abajo-Arriba


**VIDA AGOTADA**




GAME OVER
```

En el caso de realizar un movimiento incorrecto, se mostrará por pantalla:

```
-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 7
MOVIMIENTO INCORRECTO
```

PARTE GRÁFICA

Para la realización de la parte gráfica, hemos hecho uso de la librería scala-swing_2.12-2.0.0.

A diferencia de la parte obligatoria de la práctica, para esta parte hemos implementado solamente el nivel 1 (tablero de 4x4) y modo manual. Para ello, hemos eliminado aquellas funciones innecesarias, como generar los tableros para los niveles 2,3 y 4, pedir el modo, nivel, generar los movimientos generados de forma automática, etc., así como las funciones que imprimían por pantalla los puntos, vidas y tablero.

Por ello, hemos creado una clase aparte llamada PR2G que extiende de la SImpleSwingApplication. En esta, hemos declarado los labels que incluirán el número de vidas, puntos, puntuación máxima y 16 labels que representarán a las casillas del tablero. A su vez, hemos definido el método que incluirá por pantalla una ventana en la que se mostrarán estos labels (objeto de la clase MainFrame). En este hemos definido la estructura que tendrá la ventana (tendrá un botón que al pulsarle saldrá del juego).

A su vez, se han incluido tres métodos que, al llamarlos desde el objeto que incluye todo el código del juego, cambiarán el contenido de los labels referentes a las casillas del tablero, el número de vidas, de puntos y de puntuación máxima:

```
def cambiarTablero(lista:List[Int]){
  labelTablero1.text_=(""+lista(0)+"-")
  labelTablero2.text_=(""+lista(1)+"-")
  labelTablero3.text_=(""+lista(2)+"-")
  labelTablero4.text_=(""+lista(3))
  labelTablero5.text_=(""+lista(4)+"-")
  labelTablero6.text_=(""+lista(5)+"-")
  labelTablero7.text_=(""+lista(6)+"-")
  labelTablero8.text_=(""+lista(7))
  labelTablero9.text_=(""+lista(8)+"-")
  labelTablero10.text_=(""+lista(9)+"-")
  labelTablero11.text_=(""+lista(10)+"-")
  labelTablero12.text_=(""+lista(11))
  labelTablero13.text_=(""+lista(12)+"-")
  labelTablero14.text_=(""+lista(13)+"-")
  labelTablero15.text_=(""+lista(14)+"-")
  labelTablero16.text_=(""+lista(15))
}
```

En el objeto Practica2Graf, en el que se incluye todas las funciones del juego en sí, se ha instanciado el objeto *interfaz* de la clase PR2G, y así, a la hora de cambiar el contenido de los labels, se les llamará a los métodos anteriormente mencionados:

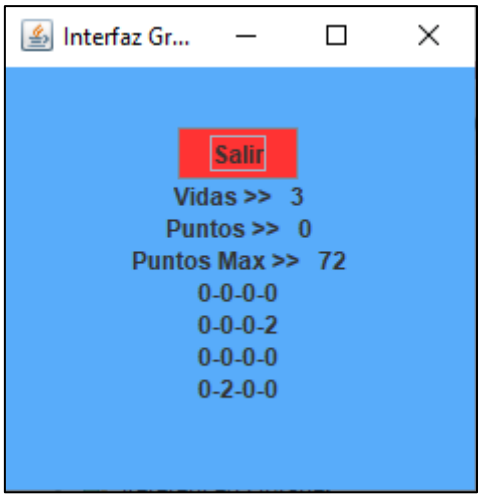
```
val interfaz = new PR2G

interfaz.cambiarTablero(inicial)
```

```
interfaz.cambiarPuntos(puntos)
interfaz.cambiarPuntosMax(puntMaximo)
interfaz.cambiarVida(vidas-1)
```

Funcionamiento juego parte gráfica

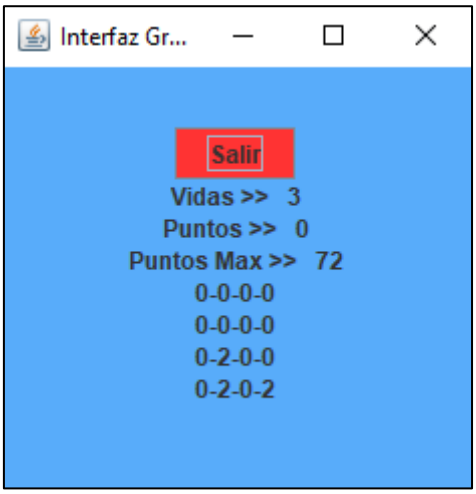
En el caso de la parte gráfica, hemos implementado solamente el nivel uno y los movimientos solo se pueden realizar a través de consola. En primer lugar, se muestra el tablero inicial con el número de vidas, puntos y puntuación máxima:



Al realizar movimientos:

- Abajo:

```
-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 2
```



- Izquierda:

```
-Realice un movimiento (Arriba->8, Derecha->6, Abajo->2, Izquierda->4): 4
```

Al perder en una jugada, el número de vidas se resta en uno: