PRÁCTICA 2A

Aitor Melero Picón, Arturo Morcillo Penares

UAM Ampliación de Programación

PRÁCTICA 2a:

a) Analizar y ejecutar el códigoInicio, explicando la implementación y los resultados de ejecución para diferentes casos, considerando diferentes cantidades de colores y comunidades autónomas.

El *códigoInicio* está implementado dentro del fichero "CodigoInicio.hs" en nuestra carpeta "ApartadoA" dentro de la carpeta "src" de nuestro proyecto stack ("ProblemaColoreado"). A la misma altura que el fichero "CodigoInicio.hs" tenemos el fichero "A.hs" donde implementamos la función "mainApartadoA" que hará del main que llama a las funciones de "CodigoInicio.hs" para probarlas.

Pasando ya a analizar el *códigoInicio* vemos que lo primero que importa es el operador "(//)" de Data.List para usarlo más tarde, después, ya podemos ver las primeras definiciones de tipo. Tenemos las definiciones de tipo algebraico para "Color", para "Provincia" y, más adelante en el código, para "Mapa"; también tenemos la definición de tipo sinónimo para el tipo "Frontera". Como se puede apreciar por los nombres de los tipos, vamos a tener un tipo que representa un color, concretamente el color rojo, verde y azul (son instancias de la clase "Show", "Enum" y "Eq"), un tipo que representa las provincias de una comunidad autónoma (Andalucía) y que también es instancia de las clases "Show", "Enum" y "Eq", un tipo sinónimo que representa las fronteras de cada provincia con una función cuyo parámetro de entrada es una provincia y cuya salida es una lista con las fronteras, es decir, las provincias vecinas, y un tipo que representa el mapa de una comunidad autónoma. Este es el código para los tipos:

```
-- Declaracion del tipo Color como instancia de varias clases
data Color = Rojo | Verde | Azul deriving (Show,Enum,Eq)
-- Declaracion del tipo Provincia como instancia de varias clases
data Provincia = Al | Ca | Co | Gr | Ja | Hu | Ma | Se deriving (Show,Enum,Eq)
-- Creación del tipo sinonimo Frontera
type Frontera = Provincia -> [Provincia]
```

```
-- Declaracion del tipo Mapa
data Mapa = Atlas [Provincia] Frontera
```

Con respecto a las funciones, tenemos en primer lugar la función "frAndalucia". Esta función devuelve una lista con las provincias vecinas de la provincia dada. Su implementación es la siguiente:

```
-- Funcion que especifica las provincias vecinas de cada provincia de Andalucia
frAndalucia :: Frontera
frAndalucia Al = [Gr]; frAndalucia Ca = [Hu,Se,Ma]
frAndalucia Co = [Se,Ma,Ja,Gr]; frAndalucia Gr = [Ma,Co,Ja,Al]
frAndalucia Ja = [Co,Gr]; frAndalucia Hu = [Ca,Se]
frAndalucia Ma = [Ca,Se,Co,Gr]; frAndalucia Se = [Hu,Ca,Ma,Co]
```

Luego nos encontramos con la función "andalucia" que directamente nos devuelve un tipo "Mapa" que representa el mapa de Andalucía con la lista de las provincias y sus respectivas fronteras. Su implementación es la siguiente:

```
-- Funcion que devuelve el mapa de Andalucia, devuelve sus provincias y sus fronteras
andalucia :: Mapa
andalucia = Atlas [Al .. Se] frAndalucia
```

La siguiente función es "coloresFrontera". Esta función, dada una provincia, una lista de provincias con un color asociado y las fronteras de la provincia, irá asociando a las fronteras de la provincia el color correspondiente basándose en la lista de tuplas provincia-color pasada como argumento. Devolverá una lista con los colores asociados a las fronteras:

```
-- Colores de provincias vecinas para un coloreado
coloresFrontera ::
    Provincia->[(Provincia,Color)]->Frontera-> [Color]
coloresFrontera provincia coloreado frontera
    = [col | (prov,col)<- coloreado, elem prov (frontera provincia)]
```

La función “coloreados” devuelve una lista de lista de tuplas a partir de una tupla de un mapa y una lista de colores, es decir, devuelve las combinaciones de colores distintos para cada provincia. Su implementación es la siguiente:

```
-- Posibles coloreados para un mapa y una lista de colores
coloreados :: (Mapa,[Color]) -> [[(Provincia,Color)]]
coloreados ((Atlas [] _), _) = [[]]
coloreados ((Atlas (prov:provs) frontera), colores)
    = [(prov,color):coloreado' |
        coloreado' <- coloreados
                            ((Atlas provs frontera), colores)
        , color <- colores \\
                    (coloresFrontera prov coloreado' frontera)]
```

Como se puede ver en la imagen, se hace uso de la función “coloresFrontera” y de una lista de comprensión que va a almacenar la lista de lista de tuplas provincia-color a devolver.

Por último, tenemos las funciones “solucionColorear”, “sol1” y “sol2”. La función “solucionColorear” devuelve el primer elemento de la lista devuelta por la función “coloreados”, es decir, devuelve la solución al problema de los colores en caso de que exista dicha solución. La función “sol1” y “sol2” llaman a la función “solucionColorear” introduciendo una comunidad autónoma, en nuestro caso la comunidad de Andalucía, y una lista de colores. En “sol1”, con 3 colores, el problema de los colores tendrá solución, pero en “sol2”, con 2 colores, el problema no tendrá solución. La implementación de estas funciones es:

```
-- Funcion que devuelve una lista con las provincias asociadas a un color
solucionColorear:: (Mapa,[Color]) -> [(Provincia,Color)]
solucionColorear = head . coloreados
sol1 = solucionColorear (andalucia, [Rojo .. Azul]) -- encuentra una solucion
sol2 = solucionColorear (andalucia, [Rojo,Verde]) -- sin solución
```

Ahora implementamos en el fichero “A.hs” la función “mainApartadoA” para probar las funciones del *códigoInicial*. En esta función hemos asignados los colores y las provincias a distintas variables en primer lugar (realmente no hace falta ya que no tenemos porque usarlas, simplemente lo hemos hecho por comodidad en caso de tenerlas que usar más adelante) y después hemos ido llamando a las funciones correspondientes de “CodigoInicial.hs”. Aquí puede verse el resultado de la ejecución de la función “mainApartadoA” para la implementación hecha para Andalucía con 3 colores:

```
*ApartadoA.A> mainApartadoA
Las fronteras son:
"Almeria: [Gr]"
"Cadiz: [Hu,Se,Ma]"
"Cordoba: [Se,Ma,Ja,Gr]"
"Granada: [Ma,Co,Ja,Al]"
"Jaen: [Co,Gr]"
"Huelva: [Ca,Se]"
"Malaga: [Ca,Se,Co,Gr]"
"Sevilla: [Hu,Ca,Ma,Co]"

Los coloreados con rojo, verde y azul son:
[[(Al,Verde),(Ca,Azul),(Co,Azul),(Gr,Rojo),(Ja,Verde),(Hu,Verde),(Ma,Verde),(Se,Rojo)],[(Al,Azul),(Ca,Azul),(Co,Azul),(Gr,Rojo),(Ja,Verde),(Hu,Verde),(Ma,Verde),(Se,Rojo)],[(Al,Verde),(Ca,Verde),(Co,Verde),(Gr,Rojo),(Ja,Azul),(
Hu,Azul),(Ma,Azul),(Se,Rojo)],[(Al,Azul),(Ca,Verde),(Co,Verde),(Gr,Rojo),(Ja,Azul),(Hu,Azul),(Ma,Azul),(Se,Rojo)],[(Al,Rojo),(Ca,Azul),(Co,Azul),(Gr,Verde),(Ja,Rojo),(Hu,Rojo),(Ma,Rojo),(Se,Verde)],[(Al,Azul),(Ca,Azul),(Co,Azul
),(Gr,Verde),(Ja,Rojo),(Hu,Rojo),(Ma,Rojo),(Se,Verde)],[(Al,Rojo),(Ca,Rojo),(Co,Rojo),(Gr,Verde),(Ja,Azul),(Hu,Azul),(Ma,Azul),(Se,Verde)],[(Al,Azul),(Ca,Rojo),(Co,Rojo),(Gr,Verde),(Ja,Azul),(Hu,Azul),(Ma,Azul),(Se,Verde)],[(Al
,Rojo),(Ca,Verde),(Co,Verde),(Gr,Azul),(Ja,Rojo),(Hu,Rojo),(Ma,Rojo),(Se,Azul)],[(Al,Verde),(Ca,Verde),(Co,Verde),(Gr,Azul),(Ja,Rojo),(Hu,Rojo),(Ma,Rojo),(Se,Azul)],[(Al,Rojo),(Ca,Rojo),(Co,Rojo),(Gr,Azul),(Ja,Verde),(Hu,Verde)
,(Ma,Verde),(Se,Azul)],[(Al,Verde),(Ca,Rojo),(Co,Rojo),(Gr,Azul),(Ja,Verde),(Hu,Verde),(Ma,Verde),(Se,Azul)]]
Los coloreados con rojo y azul son:
[]

Las soluciones son:
[(Al,Verde),(Ca,Azul),(Co,Azul),(Gr,Rojo),(Ja,Verde),(Hu,Verde),(Ma,Verde),(Se,Rojo)]
*** Exception: Prelude.head: empty list
```

Como puede verse hemos obtenido las fronteras para cada provincia, todas las combinaciones de coloreados con 3 colores (en el caso de 2 colores no se pueden dar las combinaciones y por lo tanto se devuelve una lista vacía como puede verse en la imagen) y la solución al problema de los colores. En esto último hay que destacar que Andalucía con 3 colores si tiene solución

pero con 2 no y por ello aparece la excepción de la lista vacía. El problema de los colores dice que con 4 colores siempre va a existir una solución al problema, en el caso de Andalucía basta con 3 colores. Para probar que no siempre existirá solución con 3 colores vamos a implementar el mismo código para otra comunidad autónoma: Castilla y León.

Para la implementación de Castilla y León, hemos copiado la misma implementación que la de Andalucía debajo de esta en el mismo fichero ("CodigoInicial.hs"). No vamos a explicar el código porque como acabamos de decir es el mismo con la salvedad de que se cambian las provincias. Al probar el funcionamiento con Castilla y León con 3 colores comprobamos que no existía solución ya que como explicamos no tiene porque existir solución con 3 colores como en el caso de Andalucía pero si existirá siempre solución con 4 colores. Por esto último, añadimos un color más a la implementación de Castilla y León y entonces si pudimos comprobar que si existía solución en ese caso:

```
*ApartadoA.A> mainApartadoA
Las fronteras de Castilla y Leon son:
"Leon: [Za,Va,Pa]"
"Palencia: [Le,Va,Bu]"
"Burgos: [Pa,Va,Seg,So]"
"Soria: [Bu,Seg]"
"Segovia: [So,Bu,Va,Av]"
"Avila: [Seg,Va,Sa]"
"Salamanca: [Av,Va,Za]"
"Zamora: [Sa,Va,Le]"
"Valladolid: [Le,Pa,Bu,Seg,Av,Sa,Za]"


Las soluciones para Castilla y Leon son:
[(Le,Azul'),(Pa,Amarillo'),(Bu,Verde'),(So,Rojo'),(Seg,Amarillo'),(Av,Verde'),(Sa,Amarillo'),(Za,Verde'),(Va,Rojo')]
*** Exception: Prelude.head: empty list
```

Como podemos ver, con 4 colores si existe solución y con menos de 4 no, por eso aparece la excepción de lista vacía. Cabe destacar que no hemos mostrado la combinación de "coloreados" con 4 colores porque sale una combinación infinita.

Nota: *Hemos dejado las pruebas para las 2 comunidades autónomas en el "mainApartadoA" para que se ejecuten a la vez.*

b) Extender el problema para el caso en que las regiones sean rectángulos, al estilo de Ej4.hs. Un rectángulo estaría definido por su vértice superior izquierdo y su vértice inferior derecho, que suponemos de coordenadas enteras cualesquiera (Integer). Por ejemplo, los vértices (1,2) y (3,7) definirían un rectángulo con base 2 y altura 5. Se considera que 2 rectángulos son vecinos si su intersección es un segmento de longitud no nula. Los resultados de los coloreados se deberán mostrar en pantalla de acuerdo con Ej4.hs, usando mosaicos, foldl, etc. Suponer en este apartado que los rectángulos están construidos con coordenadas correctas y que no hay solapamiento entre ningún par de rectángulos.

Para este apartado, implementado en la carpeta "ApartadoB", hemos cogido la idea del apartado anterior y la hemos juntado con la idea del "Ej4.hs", es decir, ahora queremos representar las listas de provincia-color del problema del coloreado en un mapa gracias al tipo Mosaico.

En primer lugar, tenemos el tipo Mosaico que es una cadena de String:

```
type Mosaico = [String]
```

Este tipo es la representación del mapa donde se mostrará cada rectángulo (provincia) de tal manera que se pueda visualizar de una manera clara la solución al problema del coloreado.

El Mosaico contará con una dimensión de 11x24, es decir, altura de 11 y anchura de 24, y con dos funciones que permiten representar al Mosaico en un estado inicial con todo "." y la solución al problema del coloreado respectivamente:

```haskell
-- Mapa de 11x24
altura :: Int
altura = 11
anchura :: Int
anchura = 24

altura' :: Integer
altura' = 11
anchura' :: Integer
anchura' = 24

-- Mosaico inicial
mosaicoInicial :: Mosaico
mosaicoInicial = replicate altura (replicate anchura '.')

-- Funcion que dibuja un Mosaico
dibujarMosaico :: Mosaico -> IO ()
dibujarMosaico = putStr . unlines -- pointfree
```

Como puede verse en la imagen, hemos creado 4 tipos (2 para altura y 2 para anchura) para poder usar la altura y la anchura en la función "replicate" que necesita de tipos Int y para poder usar la altura' y anchura' de tipo Integer para la representación de los rectángulos (provincia) ya que en el enunciado se nos pide que las coordenadas sean de tipo Integer. Para representar el Mosaico hacemos uso de funciones como "replicate" o "unlines" vistas en teoría entre otras muchas.

Luego, de nuevo, tenemos el tipo Color en el que de momento hemos mantenido 3 colores para basarnos en el apartado anterior pero que luego a medida que avancemos en la práctica se incrementarán básicamente porque sin un mínimo de 4 colores no existirá siempre una solución. Además, tenemos el tipo Provincia que son los rectángulos pedidos en el enunciado (cada vez que nos refiramos a Provincia nos estaremos refiriendo a los rectángulos, lo llamamos Provincia por seguir el apartado anterior y porque el objetivo del problema del coloreado es trabajar en un mapa). El tipo Provincia es un rectángulo con una coordenada superior izquierda (x, y) y una coordenada inferior derecha (x, y). Otra vez, hemos inicializado todas las provincias de Andalucía para trabajar más rápido y ya que en este apartado todavía no se pide ir experimentando con distintos rectángulos, se verá a lo largo de la práctica. Todo esto queda resumido en:

```haskell
-- Tipo Color
data Color = Rojo | Verde | Azul deriving (Show,Enum,Eq)

-- Tipo Provincia
data Provincia = Provincia {arribaIzq::(Integer, Integer), abajoDer::(Integer, Integer)}
    deriving (Show,Eq)

-- Asignacion de provincias
hu = Provincia (4, 6) (5, 8)
ca = Provincia (6, 8) (8, 9)
se = Provincia (6, 5) (10, 7)
ma = Provincia (9, 8) (16, 9)
co = Provincia (11, 6) (16, 7)
ja = Provincia (13, 3) (16, 5)
gr = Provincia (17, 4) (19, 8)
al = Provincia (20, 6) (22, 8)
```

A continuación, tenemos una función que en este apartado no es de mucha importancia pero que en el resto de apartados sí lo será para resolver el problema del coloreado, esa función es

"sonVecinos". Esta función comprueba a partir de dos provincias (rectángulos) si estas son vecinas, es decir, si están en contacto:

```
-- Funcion que indica si dos provincias son vecinas
sonVecinos :: Provincia -> Provincia -> Bool
sonVecinos (Provincia (x11, y11) (x12, y12)) (Provincia (x21, y21) (x22, y22))
    | x11 == x21 && x12 == x22 && y11 == y21 && y12 == y22 = False
    | (x11 == (x22 + 1) || x12 == (x21 - 1) || x11 == (x22 - 1) || x12 == (x21 + 1)) && not (null ([y11 .. y12] `intersect` [y21 .. y22])) =
      True
    | (y11 == (y22 + 1) || y12 == (y21 - 1) || y11 == (y22 - 1) || y12 == (y21 + 1)) && not (null ([x11 .. x12] `intersect` [x21 .. x22])) =
      True
    | otherwise = False
```

Como puede verse, en caso de tener las mismas coordenadas se devolverá False ya que suponemos que es la misma provincia y en caso de que exista contacto entre los rectángulos, comprobado gracias al "intersect" y a la comparación "==", devolveremos True. En cualquier otro caso devolveremos lógicamente False.

Con respecto a las fronteras, es decir, las provincias-rectángulos que son vecinas de otras provincias-rectángulos, tenemos el tipo Frontera que es una función de Provincia a [Provincia] (cada provincia tiene una lista de provincias que son vecinas de la misma), la función "fronteras" que devuelve la lista de provincias vecinas de una provincia y la función "frAndalucia" que hemos usado en este apartado por simplicidad a la hora de representar el mapa de Andalucía como en el apartado anterior. En posteriores apartados, hemos usado claramente la función de vecinos para ir sacando las fronteras y así poder probar diversos casos y no solo de Andalucía como en este apartado. Todo esto queda en la siguiente implementación:

```
-- Tipo Frontera
type Frontera = Provincia -> [Provincia]

-- Funcion que devuelve una lista de fronteras para una lista de provincias
fronteras :: [Provincia] -> Provincia -> [Provincia]
fronteras lista provincia = filter (sonVecinos provincia) lista

-- Funcion que devuelve la frontera para cada provincia de Andalucia
frAndalucia :: Frontera
frAndalucia x
    | x == al = [gr]
    | x == ca = [hu,se,ma]
    | x == co = [se,ma,ja,gr]
    | x == gr = [ma,co,ja,al]
    | x == ja = [co,gr]
    | x == hu = [ca,se]
    | x == ma = [ca,se,co,gr]
    | x == se = [hu,ca,ma,co]
    | otherwise = []
```

Una vez visto el tipo Frontera, pasamos al tipo Mapa. El tipo Mapa representa una lista de provincias-rectángulos junto con la función que va sacando las fronteras (en este caso se sacan las fronteras directamente con la función "frAndalucia" como se explicaba arriba) gracias al constructor Atlas, en resumen, un tipo Mapa representa un mapa, valga la redundancia. Para el apartado B hemos representado el mapa de Andalucía:

```
-- Tipo Mapa
data Mapa = Atlas [Provincia] Frontera
-- Funcion que devuelve el mapa de Andalucia
andalucia :: Mapa
andalucia = Atlas [al, ca, co, gr, ja, hu, ma, se] frAndalucia
```

Para no repetir la explicación del apartado A de las funciones que resuelven el problema del coloreado, pasamos directamente a las funciones que pasan de dicha solución a la representación en el mapa.

En primer lugar, la función "incluirProvincias" parte de un mosaico (en este caso del mosaico inicial) y una lista de provincias con un color asignado. Con estos argumentos, la función nos devolverá el mosaico con la representación del problema del coloreado ya resuelto. Esta función simplemente llama gracias a la función de pliegue "foldl" a la función "incluirProvincia" que ahora más adelante comentamos. La implementación de "incluirProvincias" es:

```
-- Funcion que incluye las provincias en el mosaico
incluirProvincias :: Mosaico -> [(Provincia,Color)] -> Mosaico
incluirProvincias = foldl incluirProvincia  -- incluirProvincia es la funcion de pliegue
```

La función "incluirProvincia" va a incluir una provincia en el mosaico que representará la solución al problema del coloreado. Esta función básicamente va comprobando cada coordenada del mapa para ver si dicha coordenada forma parte de la provincia a representar, si esto es así se pinta del color asignado a la provincia dicha coordenada. Su implementación es la siguiente:

```
-- Superpone el cuadrado en las posiciones del mosaico
incluirProvincia :: Mosaico -> (Provincia,Color) -> Mosaico
incluirProvincia mosaico (provincia,color)  = map fila mapa_altura
        where mapa_altura :: [Integer]
              mapa_altura = [1..altura']
              fila n = map (letra n) mapa_anchura
              mapa_anchura :: [Integer]
              mapa_anchura = [1..anchura']
              (letra n) m | dentroProvincia n m provincia = getLetra color
                          | otherwise = mosaico !!! n' !!! m'
                            where n' = n-1
                                  m' = m-1
```

Es importante destacar que "incluirProvincia" es la función que va coordenada a coordenada comprobando si forma parte de la provincia pero realmente la función que comprueba si una coordenada forma parte de la provincia es "dentroProvincia", a la que se llama desde "incluirProvincia". Su implementación es:

```
-- Funcion que comprueba si una provincia esta dentro del mapa o no
dentroProvincia :: Integer -> Integer -> Provincia -> Bool
dentroProvincia n m provincia =
        (y2 >= n)  && (x1 <= m)
        && (y1 <= n) && (x2 >= m)
        where (x1, y1) = arribaIzq provincia
              (x2, y2) = abajoDer provincia
```

Otro detalle importante, es como se representan los colores de cada provincia en el mosaico. Para ello se asigna a cada color su letra inicial de la siguiente manera (en este caso, como se comentaba anteriormente, con 3 colores son suficientes):

```
-- Funcion que asigna una letra a cada color
getLetra :: Color -> Char
getLetra Rojo = 'r'
getLetra Verde = 'v'
getLetra Azul = 'a'
```

Con todo esto ya se puede representar el ejemplo para Andalucía del apartado A en un mosaico. En la función "mainApartadoB" simplemente se llama a la función que pinta un mosaico con la solución del problema del coloreado para Andalucía con 3 colores representado en un mosaico:

```
-- Main para el apartado B
mainApartadoB :: IO ()
mainApartadoB = do dibujarMosaico (incluirProvincias mosaicoInicial sol1 )
```

La representación de dicha solución es la siguiente:

```
*ApartadoB.B> mainApartadoB
.......................
.......................
...........vvvv........
...........vvvvrrr.....
.....rrrrr..vvvvrrr.....
...vvrrrrraaaaaarrrvvv..
...vvrrrrraaaaaarrrvvv..
...vvaaavvvvvvvvrrrvvv..
.....aaavvvvvvvv........
.......................
.......................
```

Como puede verse, se representa el mapa aproximado de Andalucía con cada provincia con su color correspondiente en función del problema del coloreado.

En conclusión, para este apartado simplemente se ha pasado la solución estática del código inicial dado en la teoría a una solución representativa gracias al "Ej4.hs". En los siguientes apartados, ya usamos más colores y distintas representaciones para probar en profundidad el problema del coloreado para diversos mapas.

c) Repetir el apartado anterior, pero considerando un control de errores usando el tipo Either String a. Este tipo es semejante a Maybe a, devolviendo valores correctos de tipo a y valores incorrectos de tipo String, en vez de Nothing. Considerar como posibles errores el intento de colorear una colección de rectángulos con algún solapamiento entre algún par de rectángulos, o valores incorrectos para las coordenadas de los vértices. Además, se deberá usar la notación do para las operaciones sobre Either, siempre que sea posible, basándose en que el tipo Either String es monádico.

Para este apartado ya queríamos implementar en profundidad el problema del coloreado junto con un control de errores gracias al tipo "Either String a" a falta de darle una funcionalidad dinámica al usuario, se verá en los apartados posteriores. A si que lo primero que hay que hacer es explicar que es el tipo "Either String a".

El tipo "Either a b" se construye de la siguiente manera:

>data Either a b = Left a | Right b

¿Y de qué nos sirve esto para el control de errores? Pues como puede apreciarse, en función del constructor que sea llamado, se tendrá un tipo a o un tipo b. En general, Left a devolverá un error y Right b devolverá el tipo correcto con el que se está trabajando. En nuestro caso, si existe algún error se devolverá Left String, donde String será una cadena en la que se explique el error,

y en caso de que no exista un error, se devolverá Right a, donde a representa el tipo con el que se está trabajando.

En el enunciado de este apartado, se comenta que los errores a tratar son para aquellos casos en que las coordenadas de una provincia (rectángulo) son erróneas y aquellos casos en los que dos provincias (rectángulos) se superponen (en el caso de que dos rectángulos sean idénticos hemos supuesto que son el mismo rectángulo y por lo tanto no es un error). Por ello, tenemos dos cadenas que representan el error en concreto:

```
-----------------------------------------------------------------------------------------
-- APARTADO C
-----------------------------------------------------------------------------------------
-- Mensaje de error por coordenadas en variable global y solapamiento
msg :: String
msg = "ERROR, valores incorrectos para coordenadas"
msg' :: String
msg' = "ERROR, existe solapamiento entre las provincias"
```

Para tratar el primer error de la manera que pensamos nosotros, ya que hay muchas maneras de realizar este apartado, implementamos una función que crea una Provincia, es decir, en lugar de llamar al constructor para crear una Provincia directamente, ahora llamamos a una función a la que se le pasan las coordenadas superior izquierda e inferior derecha y nos devolverá un tipo “Either String Provincia” en función de si las coordenadas son correctas o erróneas. En caso de que las coordenadas estén mal tendremos un “Left String” cuyo String será “msg” que es la cadena que explica que las coordenadas están mal, en caso contrario tendremos un “Right Provincia” cuya Provincia es el rectángulo creado. Las funciones que hacen posible todo esto son:

```
-- Funciones que comprueban si la x y la y se encuentran dentro de los parametros posibles
comprobarX :: Integer -> Integer -> Bool
comprobarX x x' = not ((x<0) || (x>anchura') || (x>x'))
comprobarY :: Integer -> Integer -> Bool
comprobarY y y' = not ((y<0) || (y>altura') || (y>y'))

-- Funcion que comprueba si los parametros de la nueva provincia estan bien y en funcion de eso se crea o no
crearProvincia :: (Integer, Integer) -> (Integer, Integer) -> Either String Provincia
crearProvincia (x1, y1) (x2, y2) = do let c1 = comprobarX x1 x2
                                          c2 = comprobarY y1 y2
                                      if not (c1&&c2) then
                                          Left msg
                                      else
                                          return (Provincia (x1,y1) (x2,y2))
```

Como puede verse, la función “crearProvincia” es la encargada de devolver el tipo “Either” correspondiente y esta llama a la función “comprobarX" y “comprobarY” que comprueban si las coordenadas x e y son correctas respectivamente. Ya se puede apreciar en la función “crearProvincia” el uso de funciones de la clase Monad. Esto es posible gracias a que “Either a b” es una instancia de dicha clase. Por ello podemos usar el “do” o “return” entre otras funciones de la clase Monad que se han implementado a lo largo del código del apartado C.

Para tratar el error del solapamiento de rectángulos, disponemos de la función “compruebaFronteras”, a la que se le pasa un “Either String [Provincia]” y devuelve True en caso de que no exista solapamiento entre ningún rectángulo pasado como argumento o False en caso contrario:

```
-- Funcion que comprueba las fronteras
compruebaFronteras :: Either String [Provincia] -> Bool
compruebaFronteras (Left _) = False
compruebaFronteras (Right []) = True
compruebaFronteras (Right [x]) = True
compruebaFronteras (Right (x:xs)) = Left msg' `notElem` [fronteras' x' (x:xs) | x' <- x:xs]
```

Esta función realmente solo comprueba si existe el valor “Left msg’” donde “msg’” representa la cadena que explica el error del solapamiento en la lista con todos los resultados a la llamada de la función “fronteras’” para cada provincia.

La función “fronteras’” a partir de un rectángulo (Provincia) y una lista de rectángulos devuelve un “Left msg’” en caso de que exista solapamiento entre alguna comprobación o un “Right [Provincia]”, es decir, una lista con todos los rectángulos que son frontera del rectángulo pasado como argumento. Su implementación es la siguiente:

```
fronteras' :: Provincia -> [Provincia] -> Either String [Provincia]
fronteras' _ [] = Right []
fronteras' provincia [x] = do let vfront = sonVecinos' provincia x
                              if provincia == x then
                                  return []
                              else if vfront == Left msg' then
                                  Left msg'
                              else if vfront == Right True then
                                  return [x]
                              else
                                  return []
fronteras' provincia (x:xs) = if provincia == x then
                                  fronteras' provincia xs else
                              do let vfront = sonVecinos' provincia x
                                     vfront' = fronteras' provincia xs
                                 if (vfront == Left msg') || (vfront' == Left msg') then
                                     Left msg'
                                 else if vfront == Right True then
                                     return (x : sacarProvincia vfront')
                                 else
                                     return (sacarProvincia vfront')
```

Hemos implementado dicha función de manera recursiva y como puede verse, esta vez si se hace uso de la función “sonVecinos” aunque esta vez la hemos modificado un poco transformándola en “sonVecinos’”. La pequeña modificación a la que nos referimos simplemente es a que esta vez se devuelva un tipo “Either String Bool” en lugar de un “Bool”. Además también hemos añadido la llamada a una nueva función “generaRectangulo” que simplemente genera una lista con todas las coordenadas que ocupa un rectángulo y que nos permitirá ver, gracias a la intersección de listas, si dos rectángulos tienen alguna coordenada en común y, por lo tanto, se solapan y existe un error (recordar que el caso para dos provincias iguales se ha tratado como bueno en “fronteras’” ya que suponemos que es la misma provincia). La implementación de estas dos funciones es la que mostramos a continuación:

```
-- Funcion que genera un rectangulo como area
generaRectangulo :: (Integer, Integer) -> (Integer, Integer) -> Integer -> [Integer]
generaRectangulo (x1, y1) (x2, y2) x
                                  | y1 > y2 = []
                                  | x1 == x2 = (x1+(anchura'*(y1-1))) : generaRectangulo (x, y1+1) (x2, y2) x
                                  | otherwise = (x1+(anchura'*(y1-1))) : generaRectangulo (x1+1, y1) (x2, y2) x

-- Funcion que indica si dos provincias son vecinas
sonVecinos' :: Provincia -> Provincia -> Either String Bool
sonVecinos' (Provincia (x11, y11) (x12, y12)) (Provincia (x21, y21) (x22, y22))
    | (x11 == x21) && (x12 == x22) && (y11 == y21) && (y12 == y22) = Left msg'
    | not (null (generaRectangulo (x11, y11) (x12, y12) x11 `intersect` generaRectangulo (x21, y21) (x22, y22) x21)) = Left msg'
    | (x11 == (x22 + 1) || x12 == (x21 - 1) || x11 == (x22 - 1) || x12 == (x21 + 1)) && not (null ([y11 .. y12] `intersect` [y21 .. y22])) =
        return True
    | (y11 == (y22 + 1) || y12 == (y21 - 1) || y11 == (y22 - 1) || y12 == (y21 + 1)) && not (null ([x11 .. x12] `intersect` [x21 .. x22])) =
        return True
    | otherwise = return False
```

Otro detalle a destacar, es el siguiente trozo de código:

```
-- Funcion que saca las fronteras de una provincia
fr :: [Provincia] -> Frontera
fr (x:xs) p = sacarProvincia (fronteras' p (x:xs))

-- Tipo Mapa
data Mapa = Atlas [Provincia] Frontera

-- Mapa con provincias
mapaAux :: [Provincia] -> Mapa
mapaAux (x:xs) = Atlas (x:xs) (fr (x:xs))
```

Ahora en lugar de pasar la función “frAndalucia” que directamente daba las provincias vecinas establecidas de antemano para cada provincia, pasamos la función “fr” al mapa (mapaAux) que se encargará de hallar todas las provincias vecinas para cada provincia.

Como puede verse en la imagen que se acaba de mostrar, no estamos trabajando ahí con el tipo “Either a b”, esto se debe a que hemos usado dicho tipo para hacer las comprobaciones de errores antes de llamar a la función “mapaAux” para crear el mapa. Lo hemos hecho de la siguiente manera para poder reutilizar código haciendo uso del tipo “Either a b”. O sea, que usamos el tipo “Either a b”, vemos que todo está correcto, convertimos dicho tipo en el tipo “Provincia” correspondiente y llamamos a las funciones creadas en los apartados anteriores.

Por lo tanto, en el “mainApartadoC” se crean los ejemplos correspondientes, se comprueban si están bien o mal devolviendo los mensajes correspondientes al error gracias al tipo “Left a”, y en caso de que todo este correcto se convierten los tipos con diversas funciones implementadas a lo largo del apartado para poder reutilizar el código del apartado anterior.

Hemos recreado el ejemplo de Andalucía y dos ejemplos más con los errores de las coordenadas y el solapamiento respectivamente, este es el resultado del “mainApartadoC”:

```
"######################### CASO CORRECTO ################################"
.........................
.........................
............rrrr.........
............rrrrvvv......
.....vvvvv..rrrrvvv......
...rrvvvvvaaaaaavvvrrr...
...rrvvvvvaaaaaavvvrrr...
...rraaarrrrrrrvvvrrr...
.....aaarrrrrrr.........
.........................
.........................
"######################### CASO INCORRECTO: COORDENADA INCORRECTA ##############################"
"ERROR, valores incorrectos para coordenadas"
"######################### CASO INCORRECTO: SUPERPOSICION DE PROVINCIAS ##############################"
"ERROR, existe solapamiento entre las provincias"
*ApartadoC.C>
```

Analizando posteriormente el código nos dimos cuenta que podíamos haber hecho más uso de (>>=) del que le hemos dado, pero realmente usamos el tipo “Either String a” de tal manera que controle los errores tal como se pide en el enunciado del apartado C.

d) Añadir interactividad al apartado anterior para permitir que un usuario pueda partir de un conjunto inicial de rectángulos e iterativamente vaya añadiendo nuevos rectángulos, cambiando la cantidad de colores y se vayan mostrando los resultados en la pantalla.

La idea de la que hemos partido para este ejercicio es comenzar con una lista vacía de rectángulos y permitir a partir de esa lista vacía que el usuario vaya añadiendo rectángulos iterativamente. Además, por cada iteración, el usuario tendrá que introducir todos los colores con los que probar los rectángulos de los que dispone en el momento preciso.

Para todo esto tenemos el "mainApartadoD" que llama simplemente a la función "introducirRectangulo". Esta función va pidiendo al usuario las coordenadas para crear un nuevo rectángulo y después de las comprobaciones sobre dichos rectángulos, llama a la función "introducirColores" que irá pidiendo al usuario que vaya introduciendo los colores con los que quiere realizar las pruebas. Para ambas funciones hemos usado la recursividad y hemos hecho uso de "<-" vistas en teoría entre otras cosas. La implementación de "introducirRectangulo" es la siguiente:

```
introducirRectangulo (r:rs) = do putStrLn "¿Desea introducir un nuevo rectángulo? (S):"
                                 introducirRect <- getLine
                                 Control.Monad.when (introducirRect == "S") $
                                        do putStrLn "Introduzca la coordenada x1 (x superior izquierda):"
                                           x1 <- getLine
                                           putStrLn "Introduzca la coordenada y1 (y superior izquierda):"
                                           y1 <- getLine
                                           putStrLn "Introduzca la coordenada x2 (x inferior derecha):"
                                           x2 <- getLine
                                           putStrLn "Introduzca la coordenada y2 (y inferior derecha):"
                                           y2 <- getLine
                                           let x1' = read x1 :: Integer
                                               y1' = read y1 :: Integer
                                               x2' = read x2 :: Integer
                                               y2' = read y2 :: Integer
                                               nuevoRectangulo = crearProvincia (x1',y1') (x2',y2')
                                               lista1 = nuevoRectangulo:r:rs
                                               listaComprueba = compruebaProvincias lista1
                                           if not listaComprueba then
                                                do putStrLn msg
                                                   introducirRectangulo (r:rs)
```

```
                                           else
                                                do let lista2 = convierteListaProvincias lista1
                                                       listaComprueba2 = compruebaFronteras lista2
                                                   if listaComprueba2 then
                                                       do colores <- introducirColores []
                                                          let mapa1 = mapaAux (sacarProvincia lista2)
                                                              sol1 = solucionColorear (mapa1, colores)
                                                          if sol1 /= [] then
                                                                do dibujarMosaico (incluirProvincias mosaicoInicial sol1)
                                                                   introducirRectangulo lista1
                                                          else
                                                                do putStrLn "No hay solucion."
                                                                   introducirRectangulo lista1
                                                   else
                                                       do putStrLn msg'
                                                          introducirRectangulo (r:rs)
```

Como podemos ver el usuario introduce los datos del rectángulo, se comprueba si las coordenadas son correctas y no hay solapamiento y se llama a la función "introducirColores" para conseguir los colores. Con los rectángulos y los colores se realiza el problema del coloreado, si hay solución se imprime el mosaico y si no la hay se imprime por pantalla "No hay solución".

La función "introducirColores" es la siguiente:

```
introducirColores (c:cs) = do putStrLn "¿Desea introducir un nuevo color? (S):"
                              introducirCol <- getLine
                              if introducirCol == "S" then
                                 do putStrLn "Introduzca un color entre el Rojo, Verde, Azul, Naranja y Morado:"
                                    nuevoColor <- getLine
                                    if nuevoColor == "Rojo" then
                                       do let nuevoColor' = Rojo
                                          if nuevoColor' `notElem` (c:cs) then
                                              introducirColores (nuevoColor':(c:cs))
                                          else
                                              do putStrLn "Color repetido. Intentalo de nuevo."
                                                 introducirColores (c:cs)
                                    else if nuevoColor == "Verde" then
                                       do let nuevoColor' = Verde
                                          if nuevoColor' `notElem` (c:cs) then
                                              introducirColores (nuevoColor':(c:cs))
                                          else
                                              do putStrLn "Color repetido. Intentalo de nuevo."
                                                 introducirColores (c:cs)
                                    else if nuevoColor == "Azul" then
                                       do let nuevoColor' = Azul
                                          if nuevoColor' `notElem` (c:cs) then
                                              introducirColores (nuevoColor':(c:cs))
                                          else
                                              do putStrLn "Color repetido. Intentalo de nuevo."
                                                 introducirColores (c:cs)
```

```
                                    else if nuevoColor == "Naranja" then
                                       do let nuevoColor' = Naranja
                                          if nuevoColor' `notElem` (c:cs) then
                                              introducirColores (nuevoColor':(c:cs))
                                          else
                                              do putStrLn "Color repetido. Intentalo de nuevo."
                                                 introducirColores (c:cs)
                                    else if nuevoColor == "Morado" then
                                       do let nuevoColor' = Morado
                                          if nuevoColor' `notElem` (c:cs) then
                                              introducirColores (nuevoColor':(c:cs))
                                          else
                                              do putStrLn "Color repetido. Intentalo de nuevo."
                                                 introducirColores (c:cs)
                                    else
                                       do putStrLn "Color incorrecto. Intentalo de nuevo."
                                          introducirColores (c:cs)
                              else
                                 return (c:cs)
```

Está función obliga al usuario a introducir al menos un color, ya que no tiene sentido no introducir ningún color. El usuario podrá introducir entre los 5 colores disponibles y una vez no quiera introducir más colores se devuelve la lista con los colores introducidos.

El programa funciona de esta manera iterativamente hasta que el usuario pulsa cualquier tecla distinta de la “S” cuando a este se le pide que si desea introducir un nuevo rectángulo.

Vamos a explicar paso a paso cómo funciona la ejecución del programa:

1. Aparece por pantalla:

   “¿Desea introducir un nuevo rectángulo? (S):

2. Si el usuario introduce cualquier cosa diferente a “S+Enter” el programa termina, sino se pasa al paso 3.
3. Aparece por pantalla:

   “Introduzca la coordenada x1 (x superior izquierda):”

4. El usuario introduce la coordenada correspondiente y se vuelve al paso 3 hasta que se han introducido las 4 coordenadas (si se introduce un valor diferente al de un Integer, se mostrará una excepción y finalizará la ejecución).
5. Si las coordenadas son incorrectas o el rectángulo se superpone a otro rectángulo introducido anteriormente, se mostrará por pantalla el mensaje de error correspondiente y se vuelve al paso 1. En caso contrario se pasa a 6.
6. Aparece por pantalla:
   “¿Desea introducir un nuevo color? (S):”

7. En caso de introducir un valor distinto al de “S+Enter” se devolverá la lista de colores ya introducidos y se pasará a 10 o se pedirá al usuario que ingrese al menos uno.
8. Aparece por pantalla:
   “Introduzca un color entre el Rojo, Verde, Azul, Naranja y Morado:”
9. Si el usuario introduce una cadena que no represente tal cuál a los colores mostrados, aparecerá un mensaje que indica que el color introducido es incorrecto. En caso contrario se pasa a 7.
10. Se muestra la solución al problema del coloreado con los rectángulos y los colores representada en un mosaico o se muestra una cadena indicando que no existe solución para ese caso. Se vuelve al paso 1 con los rectángulos introducidos hasta el momento.

Se pueden ver dos ejemplos ilustrados en el comentario final de “D.hs” en la carpeta “ApartadoD/”. Aún así, el usuario es libre de introducir valores dentro de lo establecido para probar el problema.

Nota: *Aparecen varios warnings al final del código. Esto se debe a que repetimos bastante el código final. Intentamos modificarlo, pero como obteníamos un error que no comprendíamos, preferimos repetir parte del código.*

e) Extender el apartado c) y d) para permitir que cada región a colorear sea a su vez una colección de rectángulos en los que no haya ninguno de ellos que no tenga vecinos en la región.

La clave en este ejercicio está en los siguientes tipos:

```
-- Tipo Region
type Region = [Provincia]
```

```
-- Tipo Frontera' para las fronteras de regiones
type Frontera' = Region -> [Region]
```

El tipo “Region”, es una lista de provincias (rectángulos) y ahora las fronteras las vamos a considerar respecto a las regiones. En resumen, en este apartado vamos a tratar los rectángulos como grupos que forman regiones, vamos a trabajar con elementos que pasan de tener de una forma rectangular a una forma más variada, más parecido a un mapa real.

Básicamente, hemos usado las mismas funciones que en el apartado anterior, pero cambiando el tipo de dato con el que trabajar, que en este caso en lugar de ser “Provincia” es “Region”. Aún así, vamos a explicar algunos cambios que si tienen importancia.

En primer lugar, tenemos la función “crearRegion” que como bien dice, su objetivo es crear regiones a partir de una lista de provincias (rectángulos). Hemos seguido la filosofía del apartado C y por ello primero trabajamos con el tipo “Either String a” y después de comprobar si existe algún error trabajamos con el tipo “Region” directamente. Para comprobar si existe algún error, es decir, si algún rectángulo no tiene ningún vecino o algún rectángulo está solapado con otro

rectángulo, llamamos a la función “comprobarRegion” que a su vez llama a la función “noSolapamiento” y “todosVecinos” que se encargan de comprobar que todas las provincias cumplen con los requisitos para formar la región. El código es el siguiente:

```haskell
-- Funcion que comprueba para una lista de provincias si no existe solapamiento entre ninguna
noSolapamiento :: [Provincia] -> Bool
noSolapamiento (p:ps) = compruebaFronteras (Right (p:ps))

-- Funcion que comprueba para una lista de provincias, si todas tienen al menos un vecino
todosVecinos :: [Provincia] -> Bool
todosVecinos (p:ps) = let listaFronteras = [fronteras' f (p:ps) | f <- p:ps] in
                      Right [] `notElem` listaFronteras && Left msg' `notElem` listaFronteras

-- Funcion que comprueba si una lista de provincias cumple los requisitos para ser una region
comprobarRegion :: [Provincia] -> Bool
comprobarRegion (p:ps) = noSolapamiento (p:ps) && todosVecinos (p:ps)

-- Funcion que crea una Region
crearRegion :: [Provincia] -> Either String Region
crearRegion [] = Right []
crearRegion [p] = Right [p]
crearRegion (p:ps) = if comprobarRegion (p:ps) then Right (p:ps) else Left msg''
```

Otro cambio a tener en cuenta es la manera de encontrar las regiones que son fronteras entre ellas. Para ello, primero comprobamos si hay solapamiento entre regiones, para ver si hay errores o no, no puede haber solapamiento. Simplemente generamos una lista con las coordenadas para cada rectángulo de cada región y, al igual que en el apartado C, comprobamos si la intersección entre la lista generada para una región y la lista para otra región es nula o no. Una vez comprobado que no hay solapamiento, para ver que dos regiones son vecinas o no, generamos el borde de cada rectángulo que forma parte de una región (borde representado de nuevo como una lista de coordenadas) y la lista con el borde exterior de cada rectángulo para otra región de tal manera que si este último borde exterior tiene algún elemento en común con el borde de la región anterior, dichas regiones serán vecinas. La implementación de todo esto es la siguiente:

```haskell
-- Funcion que genera el borde de un rectangulo
generaRectangulo' :: (Integer, Integer) -> (Integer, Integer) -> Integer -> Integer -> [Integer]
generaRectangulo' (x1, y1) (x2, y2) x y
                                  | y1 > y2 = []
                                  | x1 == x2 = (x1+(anchura'*(y1-1))) : generaRectangulo' (x, y1+1) (x2, y2) x y
                                  | y1 == y2 = (x1+(anchura'*(y1-1))) : generaRectangulo' (x1+1, y1) (x2, y2) x y
                                  | x1 == x = (x1+(anchura'*(y1-1))) : generaRectangulo' (x1+1, y1) (x2, y2) x y
                                  | y1 == y = (x1+(anchura'*(y1-1))) : generaRectangulo' (x1+1, y1) (x2, y2) x y
                                  | otherwise = generaRectangulo' (x1+1, y1) (x2, y2) x y

-- Funcion que genera el area de una region
generaArea :: Region -> [Integer]
generaArea [] = []
generaArea [Provincia (x1,y1) (x2,y2)] = generaRectangulo (x1,y1) (x2,y2) x1
generaArea (p:ps) = let listaRectangulos = [generaArea [p'] | p' <- p:ps] in
                    foldl (++) (head listaRectangulos) (tail listaRectangulos)

-- Funcion que genera el borde de una region
generaBorde :: Region -> [Integer]
generaBorde [] = []
generaBorde [Provincia (x1,y1) (x2,y2)] = generaRectangulo' (x1,y1) (x2,y2) x1 y1
generaBorde (p:ps) = let listaBorde = [generaBorde [p'] | p' <- p:ps] in
                     foldl (++) (head listaBorde) (tail listaBorde)
```

```haskell
-- Funcion que genera el borde auxiliar de una region para comprobar las regiones vecinas
generaBordeAux :: Region -> [Integer]
generaBordeAux [] = []
generaBordeAux r = let borde = generaBorde r
                       bordeArriba = [ba - anchura' | ba <- borde]
                       bordeDerecha = [if anchura' `mod` bd == 0 then bd else bd + 1  | bd <- borde]
                       bordeAbajo = [bb + anchura' | bb <- borde]
                       bordeIzquierda= [if anchura' `mod` bd == 1 then bd else bd - 1  | bd <- borde]
                       bordeAux = bordeArriba ++ bordeDerecha ++ bordeAbajo ++ bordeIzquierda in
                   bordeAux
```

```haskell
-- Funcion que indica si dos regiones son vecinas
sonVecinos'' :: Region -> Region -> Either String Bool
sonVecinos'' r1 r2
    | r1 == r2 = Left msg'''
    | not (null (generaArea r1 `intersect` generaArea r2)) = Left msg'''
    | not (null (generaBorde r1 `intersect` generaBordeAux r2)) =
      return True
    | otherwise = return False
```

Después de esto, como ya hemos comentado, simplemente hemos reusado el código anterior pero modificándolo lo mínimo posible para poder trabajar con el tipo “Region”. Además, en cuanto al “mainApartadoE”, hemos tenido que crear una función llamada “introducirRegion” que llama a la anterior “introducirRectangulo”. Se sigue por lo tanto la misma base que en el apartado anterior, salvo que ahora el usuario irá introduciendo cada región de manera secuencial de una en una por cada iteración, y las provincias se introducirán seguidas de manera similar a como se introducen los colores, con la salvedad de que las regiones se van almacenando, por lo tanto, las provincias también se almacenan.

En resumen, en este apartado se ha modificado el código que ya se tenía para que, en lugar de trabajar con rectángulos, se trabajase con conjuntos de rectángulos. Para ver un ejemplo de uso se recomienda ver el comentario final del fichero “E.hs”, aunque este puede variar ya que este programa es dinámico, es el usuario el que elige los datos de entrada.

NOTA FINAL DE EJECUCIÓN:

A diferencia de la práctica anterior, se recomienda ir ejecutando cada programa uno a uno porque en este caso existen programas en los que el usuario juega un papel importante. Además, el ejercicio del apartado a), tal como se ha explicado, mostrará una excepción parando la ejecución del proyecto stack al completo. No es que esto esté mal, es que es parte de lo que tiene que dar por la implementación del codigoInicial del que se parte.