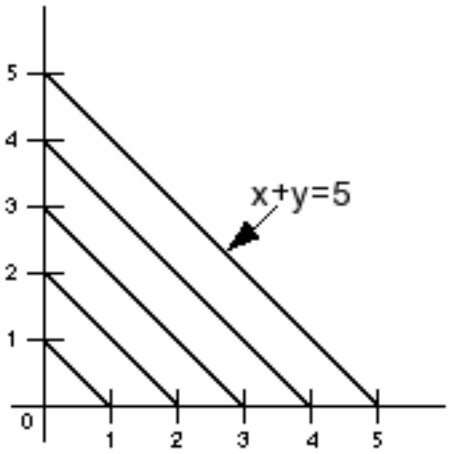
Ejercicio 1

Sea el siguiente tipo: `type Punto = (Int,Int)`

Definir, usando listas por comprensión, una lista (infinita) `puntosDelPlano::[Punto]` que contenga todos los puntos del cuadrante superior derecho del plano.

Atención: la función debe garantizar que eventualmente todo punto del cuadrante superior derecho será listado. Una forma de garantizar esto es recorrer los puntos siguiendo las diagonales $x + y = k$, con $k = 0, 1, 2, 3, \ldots$ (ver el gráfico).

Ejemplo: `[ (0,0), (0,1), (1,0), (0,2), (1,1), (2,0), (0,3), (1,2), (2,1), (3,0), (0,4), ... ]`



Solución 1

```
puntosDelPlano :: [Punto]
puntosDelPlano = [(x, y) | k <- [0..], x <- [0..k], y<- [0..k], x + y == k]
```

Solución 2

```
puntosDelPlano :: [Punto]
puntosDelPlano = [(x,k-x) | k<-[0..], x<-[0..k]]
```

Ejercicio 2

Definimos el siguiente tipo: `data Árbol a = Hoja a | Bin (Árbol a) a (Árbol a)`

Ejemplo: `miÁrbol = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))`

Sea `foldAB` el siguiente esquema de recursión genérico para árboles:

```
foldAB f g (Hoja n) = f n
foldAB f g (Bin t1 n t2) = g (foldAB f g t1) n (foldAB f g t2)
```

En la resolución de este ejercicio, no usar recursión explícita.

1. Decir cuál es el tipo de las funciones `f`, `g` y `foldAB`.

   Solución

   a) Miremos `foldAB`. Sabemos que recibe 3 parámetros:

   ```
   foldAB :: a -> b -> c -> d
   ```

Por la forma en que esta definida `foldAB`, sabemos que el 3er parámetro es un Árbol. Además, sabemos que el primer y segundo parámetro son funciones.

```
foldAB :: a -> b -> Árbol c -> d
```

b) Sabemos que `f` (el primer parámetro de `foldAB`) recibe un sólo parámetro:

```
foldAB :: (a -> e) -> b -> Árbol c -> d
```

Por los usos de `f` en la definición de `foldAB` debemos pensar que en principio `a` y `e` podrían ser tipos distintos.

c) Sabemos que `g` (el segundo parámetro de `foldAB`) recibe tres parámetros:

```
foldAB :: (a -> e) -> (b -> h -> i -> j) -> Árbol c -> d
```

d) Mirando:

`foldAB f g (Bin t1 n t2) = g (foldAB f g t1) n (foldAB f g t2)`

Vemos que `foldAB f g (Bin t1 n t2)` es de tipo `d` (el tipo de retorno de `foldAB`).

Además `foldAB f g t1`, `foldAB f g t2` y `g (foldAB f g t1) n (foldAB f g t2)` también deben ser del mismo tipo. Por lo tanto:

```
foldAB :: (a -> e) -> (d -> h -> d -> d) -> Árbol c -> d
```

e) Mirando ahora:

`foldAB f g (Hoja n) = f n`

y sabiendo que `foldAB f g (Hoja n)` tiene que ser de tipo `d`, podemos concluir que `f n` también será del mismo tipo. Por lo tanto:

```
foldAB :: (a -> d) -> (d -> h -> d -> d) -> Árbol c -> d
```

f) Mirando nuevamente:

`foldAB f g (Hoja n) = f n`

Vemos que el parámetro de `f` es del mismo tipo que los elementos del árbol. Lo mismo pasa con el segundo parámetro de `g` en:

`foldAB f g (Bin t1 n t2) = g (foldAB f g t1) n (foldAB f g t2)`

Por lo tanto:

```
foldAB :: (a -> d) -> (d -> a -> d -> d) -> Árbol a -> d
```

No podemos deducir nada más. Haciendo un renombre de `d` por `b` nos queda:

`foldAB :: (a ->b) ->(b ->a ->b ->b) ->Árbol a ->b`

---

2. Usando `foldAB`, definir una función `hojas :: Árbol a ->[a]`, que devuelva todas las hojas de un árbol.

   Ejemplo: `hojas miÁrbol` devuelve `[3, 7, 1]`

Solución

```
hojas = foldAB singleton union
           where
             singleton x = [x]
             union xs x ys = xs ++ ys
```

---

3. Usando `foldAB`, definir una función `espejo :: Árbol a ->Árbol a`, que devuelva el árbol que resulte de invertir en todos los niveles el subárbol izquierdo por el derecho.

   Ejemplo: `espejo miÁrbol` devuelve el árbol `Bin (Bin (Hoja 1) 8 (Hoja 7)) 5 (Hoja 3)`

   Solución

```
espejo = foldAB Hoja rbin
           where
             rbin t x t' = Bin t' x t
```

---

4. Usando `foldAB`, definir una función `ramas :: Árbol a -> [[a]]`, que devuelva la lista de ramas de un árbol.

   Ejemplo: `ramas miÁrbol` devuelve `[[5,3],[5,8,7],[5,8,1]]`

   Solución

```
ramas = foldAB singleton merge
          where
              singleton x = [[x]]
              merge xs x ys = (map (x:) xs) ++ (map (x:) ys)

ramas = foldAB ((:[]).(:[])) merge
            where
                merge xs x ys = (map (x:) xs) ++ (map (x:) ys)
```

Ejercicio 3

Dado el siguiente tipo, que representa un árbol arbitrario o *RoseTree*:
`data Árbol a = Rose(a, [Árbol a])`
deducir el tipo de la siguiente función:
`g a f = Rose(a, map (flip g f) (f a))`
donde:
`flip f x y = f y x`
Solución

- Sabemos que `g` recibe dos parámetros:

  `g :: a -> b -> c`

- El resultado de `g` es un `Árbol`: (por `Rose`):

  `g :: a -> b -> Árbol c`

- En la expresión `f a`, vemos que `f` (el segundo parámetro) debe ser una función de por lo menos un parámetro:

  `g :: a -> (b -> d) -> Árbol c`

- A su vez, `f a` es el segundo argumento de `map` en `map (flip g f) (f a)`, y por lo tanto debe ser una lista:

  `g :: a -> (b -> [d]) -> Árbol c`

- Notemos que el primer parámetro de `g` sirve como parámetro a `f` y como elemento del árbol resultado, con lo cual:

  `g :: a -> (a -> [d]) -> Árbol a`

- Por otro lado, sabemos que el tipo de `flip` es:

  `flip :: (a -> b -> c) -> (b -> a -> c)`

  Quiere decir que:

  `flip g :: (a -> [d]) -> a -> Árbol a`

  Entonces:

  `flip g f :: a -> Árbol a`

  Como el primer parámetro de `map` en `map (flip g f) (f a)` es una función de tipo `a -> Árbol a`, los elementos de la lista `f a` deben ser de tipo `a`. Por lo tanto:

  `g :: a -> (a -> [a]) -> Árbol a`

- Podemos ver que todo cierra porque todo el map devuelve `[Árbol a]`, es es lo que necesita el constructor `Rose` de `Árbol`

Ejercicio 4

1. Definir la función:

   `foldr2:: (a -> b -> c -> c) -> c -> [a] -> [b] -> c`

   Esta función es similar a `foldr`, pero trabaja sobre dos listas en lugar de una. La función que recibe como parámetro se aplica a un elemento de cada lista y al caso base calculado hasta el momento. Ambas listas son recorridas de derecha a izquierda y la función sólo debe estar definida para listas de igual longitud.

   Por ejemplo, el resultado de evaluar esta expresión:

   `foldr2 (\x y b -> (x,y):b) [] [1,2,3] [‘a’,‘b’,‘c’]`

   es: `[(1,‘a’),(2,‘b’),(3,‘c’)]`

   Solución

```
foldr2:: (a -> b -> c -> c) -> c -> [a] -> [b] -> c

foldr2 f b [] [] = b
foldr2 f b (x:xs) (y:ys) = f x y (foldr2 f b xs ys)
```

---

2. Sea el tipo:

   `type Matriz a = [[a]]`

   Utilizando `foldr2` y listas por comprensión, y sin utilizar recursión explícita, definir el operador:

   `(<*>) :: Matriz a -> Matriz a -> Matriz a`

   Este operador toma dos matrices y devuelve el producto entre ellas. Recordar que el producto entre dos matrices m_1 y m_2 sólo está definido cuando m_1 es de $N \times M$ y m_2 es de $M \times T$.

   Por ejemplo:

```
 1 2       1 2 3      9 12 15
 4 5   X   4 5 6  =  24 33 42
 7 8                 39 54 69
```

   Ayuda: Se cuenta con la siguiente función:

```
col :: Int -> Matriz a -> [a]
col i = map (!!i)
```

   que devuelve la i-ésima columna de una matriz (la primera columna corresponde al 0).

   Solución

```
cols m = [col z m | z <- [0..((length (m!!0)) - 1)]]

(<*>) m1 m2 =
    [  [   foldr2 (\x y b -> x*y + b) 0 (m1!!i) ((cols m2)!!j) |
                        j <- [0..(length (cols m2)) - 1] ]
        | i <- [0..(length m1) - 1]  ]
```

   - `i <- [0..(length m1) - 1]` es la lista de índices de filas de `m1`, es decir $0 \ldots N-1$.
   - `j <- [0..(length (cols m2)) - 1]` es la lista de índices de columnas de `m2`, es decir $0 \ldots T-1$.
   - La lista interna es una de las filas de la matriz resultado. Cada fila tendrá T columnas.
   - La lista de afuera es la lista de filas de la matriz resultado: habrá N filas, como era de esperarse.
   - `m1!!i` es la i-ésima fila de la matriz `m1`.
   - `cols m2!!j` es la j-ésima columna de la matriz `m2`.
   - La función `(\x y b -> x*y + b)` toma un elemento de la fila y otro de la columna, los multiplica entre sí y los suma al acumulado.
   - Quiere decir que cada elemento de la lista de más adentro es el resultado de multiplicar una fila de `m1` por una columna de `m2`.

   Es decir:

$$(m1 <*> m2)_{ij} = \sum_k m1_{ik} * m2_{kj}$$