

```

1 module Grafo (Grafo, vacio, nodos, vecinos, agNodo, sacarNodo, agEje, lineal, union,
  clausura) where
2
3 import qualified Data.List (union)
4
5 data Grafo a = G [a] (a -> [a])
6
7 instance (Show a) => Show (Grafo a) where
8   show (G n e) = "[\n" ++ concat (map (\x -> " " ++ show x ++ " -> " ++ show (e x) ++
  "\n") n) ++ "]"
9
10 instance (Eq a) => Eq (Grafo a) where
11   (G n1 e1) == (G n2 e2) = (listasIguales n1 n2) && (all (\n -> (listasIguales (e1 n) (e2
  n))) n1)
12
13 -- Igualdad de listas sin importar el orden
14 listasIguales :: (Eq a) => [a] -> [a] -> Bool
15 listasIguales l1 l2 = (all (\x -> x `elem` l1) l2) && (all (\x -> x `elem` l2) l1)
16
17 -- -----Sección 3----- Grafos -----
18
19 -- Ejercicio 1
20 -- Crea un nuevo grafo con una lista vacía de nodos y una función que
21 -- devuelve siempre []. Es decir que si pedimos los vecinos de
22 -- cualquier nodo que no esté en el grafo, da una lista vacía. Esto es
23 -- para que la función sea total.
24 vacio :: Grafo a
25 vacio = G [] (const [])
26
27 -- Ejercicio 2
28 -- Devuelve la lista de nodos del grafo que se pasa por parámetro.
29 nodos :: Grafo a -> [a]
30 nodos (G ns ejes) = ns
31
32 -- Ejercicio 3
33 -- Dado un grafo, devuelve una función que toma un nodo y retorna la lista de vecinos del
  mismo.
34 vecinos :: Grafo a -> a -> [a]
35 vecinos (G ns ejes) = (\y -> ejes y)
36
37 -- Ejercicio 4
38 -- Agrega un nodo al grafo en el caso en el que el mismo no le pertenezca.
39 -- Caso contrario, devuelve el grafo original.
40 agNodo :: Eq a => a -> Grafo a -> Grafo a
41 agNodo x (G ns t) = if x `elem` ns then (G ns t) else (G (x:ns) t)
42
43 -- Ejercicio 5
44 -- Construye un nuevo grafo:
45 --   * Filtra la lista de nodos para sacar el nodo.
46 --   * Crea una nueva función que devuelve [] para el nodo que acabamos
47 --     de sacar y para los demás nodos devuelve los mismos vecinos de
48 --     antes salvo el nodo que se sacó.
49 sacarNodo :: Eq a => a -> Grafo a -> Grafo a
50 sacarNodo n (G nodos ejes) = G (filter (/=n) nodos)
51   (\x -> if (x==n) then [] else (filter (/=n) (ejes x)))
52
53 -- Ejercicio 6
54 -- Devuelve el grafo ingresado por parámetro con el agregado del eje que une el
55 -- primer nodo de la tupla con el segundo.
56 --   * Se verifica si y pertenece a la lista de vecinos de x
57 --   * Si lo hace, se devuelve el grafo con la función sin modificar
58 --   * Si no, se modifica la función para agregar la nueva arista y se devuelve el grafo.
59 agEje :: Eq a => (a,a) -> Grafo a -> Grafo a
60 agEje (x, y) (G ns t) = if y `elem` (t x) then (G ns t) else
61   (G ns (\n -> if n == x then y:(t x) else (t n)))
62
63
64
65

```

```

66
67 -- Ejercicio 7
68 -- Si la lista es vacía, crea un grafo vacío.
69 -- Si la lista no es vacía, recursivamente va agregando nodos al grafo,
70 -- y a partir del segundo nodo que agrega, también va agregando ejes.
71 -- Es por eso que en el paso inductivo primero pregunta si el grafo
72 -- "rec" ya tiene algún nodo y si es así agrega un eje entre el último
73 -- nodo agregado y el que se está agregando en este paso.
74 lineal :: Eq a => [a] -> Grafo a
75 lineal = foldr (\n rec -> if null (nodos rec)
76                       then agNodo n rec
77                       else agEje (n,head (nodos rec)) (agNodo n rec)
78                             )
79     vacio
80
81 -- Ejercicio 8
82 -- Devuelve un grafo con la unión de los nodos de los dos que entran por parámetro.
83 -- Utilizaremos la unión de conjuntos para unir los nodos y los vecinos evitando repetidos.
84 union :: Eq a => Grafo a -> Grafo a -> Grafo a
85 union ga gb = G (Data.List.union (nodos ga) (nodos gb))
86     (\x -> Data.List.union ((vecinos ga) x) ((vecinos gb) x))
87
88 -- Ejercicio 9
89 -- Recorremos los nodos del grafo y por cada nodo agregamos los
90 -- vecinos que se obtienen por reflexividad y transitividad.
91 -- * Para reflexividad, simplemente agregamos un loop.
92 -- * Para transitividad, vamos a buscar los vecinos de los vecinos de los
93 --   vecinos ... de los vecinos del nodo (nodosAlcanzables). Agregamos
94 --   todos los ejes hacia esos nodos.
95 -- Observar que agregar ejes repetidos no modifica el grafo.
96 clausura :: (Eq a) => Grafo a -> Grafo a
97 clausura grafoOriginal@(G nodos vecinos) = foldr
98     (\x grec -> agEje (x,x)
99                     (agEjesDesdeHasta grec x
100                      (nodosAlcanzables grec x)))
101     grafoOriginal
102     nodos
103
104
105 -- agEjesDesdeHasta g x [y1,...,yn] = Al grafo g le agrega los ejes
106 -- (x,y1),..., (x,yn).
107 agEjesDesdeHasta :: (Eq a) => Grafo a -> a -> [a] -> Grafo a
108 agEjesDesdeHasta grafo x = foldr (\y grec -> agEje (x,y) grec) grafo
109
110
111 -- Toma un grafo y un nodo y devuelve todos los nodos alcanzables
112 -- por transitividad.
113 --
114 -- Busca el punto fijo de una función lambda.
115 -- Esta función lambda, toma una lista de nodos y hace la unión
116 -- de esa lista con todos los vecinos de esos nodos. Aplicar muchas veces
117 -- esta función eventualmente tiene un punto fijo (porque siempre es la
118 -- lista de parámetro la que se une con otra, entonces el prefijo se
119 -- mantiene y a lo sumo se eliminan repetidos de la segunda lista).
120 --
121 -- El punto fijo justamente se alcanza cuando se recorrieron todos
122 -- los nodos alcanzables (clausura transitiva) desde el nodo inicial.
123 nodosAlcanzables :: (Eq a) => Grafo a -> a -> [a]
124 nodosAlcanzables grafo n = puntoFijo
125     (\listaNodos -> (Data.List.union listaNodos
126     (vecinosDeTodos grafo listaNodos))) [n]
127
128
129 -- Toma un grafo y una lista de nodos y devuelve una lista que tiene
130 -- todos los vecinos de esos (sin repetidos)
131 vecinosDeTodos :: (Eq a) => Grafo a -> [a] -> [a]
132 vecinosDeTodos (G nodos vecinos) = foldr (\x grec ->
133     (Data.List.union grec (vecinos x))) []
134

```

```
135
136 -- Punto fijo de f para un valor x de entrada. Es decir devuelve
137 -- el resultado de aplicar f (f (f ...(f x)...)) hasta que f y = y.
138 -- Para hacer esto, usamos una lista por comprensión con un selector
139 -- infinito y la condición implica que el primer elemento de la lista,
140 -- será el punto fijo de f.
141 puntoFijo :: (Eq a) => (a -> a) -> a -> a
142 puntoFijo f x = [(aplicarNVeces n f x) | n <- [1..],
143                (aplicarNVeces n f x) == (aplicarNVeces (n-1) f x)] !! 0
144
145
146 -- Para aplicar n veces f, usamos un esquema de recursión sobre la lista
147 -- [1..n] y en cada paso aplicamos una vez f. Al terminar de recorrer
148 -- la lista habremos aplicado N veces f (esta función es el análogo a
149 -- un "for" imperativo).
150 aplicarNVeces :: Int -> (a -> a) -> a -> a
151 aplicarNVeces n f x = foldr (\_ res -> f res) x [1..n]
152
153
154
155
156
```