```haskell
module Grafo (Grafo, vacio, nodos, vecinos, agNodo, sacarNodo, agEje, lineal, union, clausura) where

import qualified Data.List (union)

data Grafo a = G [a] (a -> [a])

instance (Show a) => Show (Grafo a) where
    show (G n e) = "[\n" ++ concat (map (\x -> " " ++ show x ++ " -> " ++ show (e x) ++ "\n") n) ++ "]"

instance (Eq a) => Eq (Grafo a) where
    (G n1 e1) == (G n2 e2) = (listasIguales n1 n2) && (all (\n -> (listasIguales (e1 n) (e2 n))) n1)

-- Igualdad de listas sin importar el orden
listasIguales :: (Eq a) => [a] -> [a] -> Bool
listasIguales l1 l2 = (all (\x -> x `elem` l1) l2) && (all (\x -> x `elem` l2) l1)

-- ---------------------------------Sección 3--------- Grafos ---------------------------

-- Ejercicio 1
-- Crea un nuevo grafo con una lista vacía de nodos y una función que
-- devuelve siempre []. Es decir que si pedimos los vecinos de
-- cualquier nodo que no esté en el grafo, da una lista vacía. Esto es
-- para que la función sea total.
vacio :: Grafo a
vacio = G [] (const [])

-- Ejercicio 2
-- Devuelve la lista de nodos del grafo que se pasa por parámetro.
nodos :: Grafo a -> [a]
nodos (G ns ejes) = ns

-- Ejercicio 3
-- Dado un grafo, devuelve una función que toma un nodo y retorna la lista de vecinos del mismo.
vecinos :: Grafo a -> a -> [a]
vecinos (G ns ejes) = (\y -> ejes y)

-- Ejercicio 4
-- Agrega un nodo al grafo en el caso en el que el mismo no le pertenezca.
-- Caso contrario, devuelve el grafo original.
agNodo :: Eq a => a -> Grafo a -> Grafo a
agNodo x (G ns t) = if x `elem` ns then (G ns t) else (G (x:ns) t)

-- Ejercicio 5
-- Construye un nuevo grafo:
--    * Filtra la lista de nodos para sacar el nodo.
--    * Crea una nueva función que devuelve [] para el nodo que acabamos
--      de sacar y para los demás nodos devuelve los mismos vecinos de
--      antes salvo el nodo que se sacó.
sacarNodo :: Eq a => a -> Grafo a -> Grafo a
sacarNodo n (G nodos ejes) = G (filter (/=n) nodos)
                               (\x -> if (x==n) then [] else (filter (/=n) (ejes x)))

-- Ejercicio 6
-- Devuelve el grafo ingresado por parámetro con el agregado del eje que une el
-- primer nodo de la tupla con el segundo.
--    * Se verifica si y pertenece a la lista de vecinos de x
--    * Si lo hace, se devuelve el grafo con la funcion sin modificar
--    * Si no, se modifica la funcion para agregar la nueva arista y se devuelve el grafo.
agEje :: Eq a => (a,a) -> Grafo a -> Grafo a
agEje (x, y) (G ns t) = if y `elem` (t x) then (G ns t) else
                                      (G ns (\n -> if n == x then y:(t x) else (t n)))

-- Ejercicio 7
-- Si la lista es vacía, crea un grafo vacío.
-- Si la lista no es vacía, recursivamente va agregando nodos al grafo,
-- y a partir del segundo nodo que agrega, también va agregando ejes.
```

```haskell
-- Es por eso que en el paso inductivo primero pregunta si el grafo
-- "rec" ya tiene algún nodo y si es así agrega un eje entre el último
-- nodo agregado y el que se está agregando en este paso.
lineal :: Eq a => [a] -> Grafo a
lineal = foldr (\n rec ->  if null (nodos rec)
                           then agNodo n rec
                           else agEje (n,head (nodos rec)) (agNodo n rec)
               )
               vacio

-- Ejercicio 8
-- Devuelve un grafo con la unión de los nodos de los dos que entran por parámetro.
-- Utilizaremos la unión de conjuntos para unir los nodos y los vecinos evitando repetidos.
union :: Eq a => Grafo a -> Grafo a -> Grafo a
union ga gb = G (Data.List.union (nodos ga) (nodos gb))
                (\x -> Data.List.union ((vecinos ga) x) ((vecinos gb) x))

-- Ejercicio 9
-- Recorremos los nodos del grafo y por cada nodo agregamos los
-- vecinos que se obtienen por reflexividad y transitividad.
--   * Para reflexividad, simplemente agregamos un loop.
--   * Para transitividad, vamos a buscar los vecinos de los vecinos de los
--      vecinos ... de los vecinos del nodo (nodosAlcanzables). Agregamos
--      todos los ejes hacia esos nodos.
-- Observar que agregar ejes repetidos no modifica el grafo.
clausura :: (Eq a) => Grafo a -> Grafo a
clausura grafoOriginal@(G nodos vecinos) = foldr
                        (\x grec ->  agEje (x,x)
                                     (agEjesDesdeHasta grec x
                                         (nodosAlcanzables grec x)))
                        grafoOriginal
                        nodos


-- agEjesDesdeHasta g x [y1,...,yn] = Al grafo g le agrega los ejes
-- (x,y1),...,(x,yn).
agEjesDesdeHasta :: (Eq a) => Grafo a -> a -> [a] -> Grafo a
agEjesDesdeHasta grafo x = foldr (\y grec -> agEje (x,y) grec) grafo


-- Toma un grafo y un nodo y devuelve todos los nodos alcanzables
-- por transitividad.
--
-- Busca el punto fijo de una función lambda.
-- Esta función lambda, toma una lista de nodos y hace la unión
-- de esa lista con todos los vecinos de esos nodos. Aplicar muchas veces
-- esta función eventualmente tiene un punto fijo (porque siempre es la
-- lista de parámetro la que se une con otra, entonces el prefijo se
-- mantiene y a lo sumo se eliminan repetidos de la segunda lista).
--
-- El punto fijo justamente se alcanza cuando se recorrieron todos
-- los nodos alcanzables (clausura transitiva) desde el nodo inicial.
nodosAlcanzables :: (Eq a) => Grafo a -> a -> [a]
nodosAlcanzables grafo n = puntoFijo
                                (\listaNodos -> (Data.List.union listaNodos
                                (vecinosDeTodos grafo listaNodos))) [n]


-- Toma un grafo y una lista de nodos y devuelve una lista que tiene
-- todos los vecinos de esos (sin repetidos)
vecinosDeTodos :: (Eq a) => Grafo a -> [a] -> [a]
vecinosDeTodos (G nodos vecinos) = foldr (\x rec ->
                                    (Data.List.union rec (vecinos x))) []


-- Punto fijo de f para un valor x de entrada. Es decir devuelve
-- el resultado de aplicar f (f (f ...(f x)...)) hasta que f y = y.
-- Para hacer esto, usamos una lista por comprensión con un selector
-- infinito y la condición implica que el primer elemento de la lista,
```

```haskell
-- será el punto fijo de f.
puntoFijo :: (Eq a) => (a -> a) -> a -> a
puntoFijo f x = [(aplicarNVeces n f x) | n <- [1..],
                (aplicarNVeces n f x) == (aplicarNVeces (n-1) f x)] !! 0


-- Para aplicar n veces f, usamos un esquema de recursión sobre la lista
-- [1..n] y en cada paso aplicamos una vez f. Al terminar de recorrer
-- la lista habremos aplicado N veces f (esta función es el análogo a
-- un "for" imperativo).
aplicarNVeces :: Int -> (a -> a) -> a -> a
aplicarNVeces n f x = foldr (\_ res -> f res) x [1..n]
```