

```

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)
  (\x -> if (x==n) then [] else (filter (/=n) (ejes x)))
51
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
  (G ns (\n -> if n == x then y:(t x) else (t n)))
61
62
63 -- Ejercicio 7
64 -- Si la lista es vacía, crea un grafo vacío.
65 -- Si la lista no es vacía, recursivamente va agregando nodos al grafo,
66 -- y a partir del segundo nodo que agrega, también va agregando ejes.

```

```

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

```

```
136 -- será el punto fijo de f.
137 puntoFijo :: (Eq a) => (a -> a) -> a -> a
138 puntoFijo f x = [(aplicarNVeces n f x) | n <- [1..],
139                  (aplicarNVeces n f x) == (aplicarNVeces (n-1) f x)] !! 0
140
141
142 -- Para aplicar n veces f, usamos un esquema de recursión sobre la lista
143 -- [1..n] y en cada paso aplicamos una vez f. Al terminar de recorrer
144 -- la lista habremos aplicado N veces f (esta función es el análogo a
145 -- un "for" imperativo).
146 aplicarNVeces :: Int -> (a -> a) -> a -> a
147 aplicarNVeces n f x = foldr (\_ res -> f res) x [1..n]
148
149
150
151
152
```

```

1 module Lomoba where
2 import Grafo
3 import List
4 import qualified Data.List (union)
5 import Tipos
6 -----Sección 6----- Lomoba -----
7
8 -- Ejercicio 10
9 foldExp :: (Prop -> a) -- Función para aplicar a (Var p)
10         -> (a -> a)    -- Función para aplicar a (Not e)
11         -> (a -> a -> a) -- Función para aplicar a (Or e1 e2)
12         -> (a -> a -> a) -- Función para aplicar a (And e1 e2)
13         -> (a -> a)     -- Función para aplicar a (D e)
14         -> (a -> a)     -- Función para aplicar a (B e)
15         -> Exp -> a     -- Función que construye el fold
16 foldExp fVar fNot fOr fAnd fd fb (Var p) = fVar p
17 foldExp fVar fNot fOr fAnd fd fb (Not e) = fNot (foldExp fVar fNot fOr fAnd fd fb e)
18 foldExp fVar fNot fOr fAnd fd fb (Or a b) = fOr
19                                     (foldExp fVar fNot fOr fAnd fd fb a)
20                                     (foldExp fVar fNot fOr fAnd fd fb b)
21 foldExp fVar fNot fOr fAnd fd fb (And a b) = fAnd (foldExp fVar fNot fOr fAnd fd fb a)
22                                     (foldExp fVar fNot fOr fAnd fd fb b)
23 foldExp fVar fNot fOr fAnd fd fb (D e) = fd (foldExp fVar fNot fOr fAnd fd fb e)
24 foldExp fVar fNot fOr fAnd fd fb (B e) = fb (foldExp fVar fNot fOr fAnd fd fb e)
25
26 -- Ejercicio 11
27 -- Calcula la visibilidad de la f ormula. Cada vez que aparece <> o [] debo
28 -- incrementar en 1 el valor de la misma.
29 -- * Var es el caso base, por lo que utilizare const 0.
30 -- * Not no sumara ningun valor, por lo que utilizare id.
31 -- * Or y And que seran bifurcaciones en el arbol de recursion y por lo tanto
32 -- tomaremos el maximo resultante de ambas ramas de la recursion.
33 -- * D y B seran los casos en donde tendre que sumar uno, por lo que aplicare
34 -- la funcion + con la valuacion parcial en el primero de sus parametros.
35 visibilidad :: Exp -> Integer
36 visibilidad = foldExp fVar fNot fOr fAnd fd fb
37   where fVar = const 0
38         fNot = id
39         fOr = max
40         fAnd = max
41         fd = (+1)
42         fb = (+1)
43
44 -- Ejercicio 12
45 -- Extraer las variables proposicionales que aparecen en la formula, sin repetir.
46 -- * Var es el caso base, por lo que utilizare una lambda que dado p me devuelve [p].
47 -- * Or y And que seran bifurcaciones en el arbol de recursion y por lo tanto
48 -- utilizaremos la union de conjuntos.
49 -- * Not, D y B no adicionaran ningun simbolo, por lo que utilizare id.
50 extraer :: Exp -> [Prop]
51 extraer = foldExp fVar fNot fOr fAnd fd fb
52   where fVar = (\p -> [p])
53         fNot = id
54         fOr = Data.List.union
55         fAnd = Data.List.union
56         fd = id
57         fb = id
58
59 -- Ejercicio 13
60 eval :: Modelo -> Mundo -> Exp -> Bool
61 eval mod w exp = (eval' mod exp) w
62
63 -- Dado un modelo y una expresión, devuelve una función
64 -- que para un Mundo dado (en el modelo), devuelve la evaluación.
65 -- En los pasos de D y B, se aplica la función recursiva sobre todos
66 -- los mundos vecinos y se busca que en alguno o en todos la expresión
67 -- sea true.
68 eval' :: Modelo -> Exp -> (Mundo -> Bool)
69 eval' (K g mundosTrue) =
70   foldExp

```

```

71      (\p w -> w `elem` (mundosTrue p)) -- ::Prop -> (Mundo -> Bool)
72      (\rec w -> not (rec w)) -- ::(Mundo -> Bool) -> Mundo -> Bool
73      (\rec1 rec2 w -> (rec1 w) || (rec2 w))
74      (\rec1 rec2 w -> (rec1 w) && (rec2 w))
75      (\rec w -> or (map rec (vecinos g w))) -- rec::(Mundo -> Bool)
76      (\rec w -> and (map rec (vecinos g w)))
77
78
79
80 -- Ejercicio 14
81 -- Dadas todas las variables proposicionales del grafo, se devuelven los mundos
82 -- que al evaluarlos dan verdadero
83 valeEn :: Exp -> Modelo -> [Mundo]
84 valeEn exp mod@(K g mundosTrue) = filter (eval' mod exp) (nodos g)
85
86 -- Ejercicio 15
87 -- Usando foldr, voy construyendo un nuevo modelo de Kripke, partiendo
88 -- del modelo pasado por argumento y sacándole los mundos donde no vale
89 -- la expresión. Por cada mundo que saco, le saco el nodo al grafo y
90 -- lo saco del valor de retorno de la función (para cada símbolo
91 -- proposicional)
92 quitar :: Exp -> Modelo -> Modelo
93 quitar e mod@(K g mundosTrue) =
94     foldr (\w (K gRec fRec) ->
95         K (sacarNodo w gRec)
96           (\prop -> filter (/=w) (fRec prop))
97         )
98     mod -- Si e vale en todos los mundos, devuelvo el
99     modelo original
100     (valeEn (Not e) mod) -- mundos donde NO vale e
101
102 -- Ejercicio 16
103 -- Compara el modelo original con el modelo resultante de quitarle los mundos tales
104 -- que no valga e.
105 cierto :: Modelo -> Exp -> Bool
106 cierto mod@(K g mundosTrue) e = (sort (nodos g) == sort (valeEn e mod))
107
108

```

```

1 import Grafo
2 import Tipos
3 import Lomoba
4 import Parser
5 import Test.HUnit
6
7 -- evaluar t para correr todos los tests
8 t = runTestTT allTests
9
10 allTests = test [
11     "parser" ~: testsParser,
12     "grafo" ~: testsGrafo,
13     "lomoba" ~: testsLomoba
14 ]
15
16 testsParser = test [
17     (Var "p") ~=? (parse "p"),
18     (And (Var "p") (Var "q")) ~=? (parse "p && q"),
19     (Or (Var "p") (Var "q")) ~=? (parse "p || q"),
20     (Or (Not (Var "p")) (Var "q")) ~=? (parse "!p || q"),
21     (And (D (Var "p")) (Var "q")) ~=? (parse "<>p && q"),
22     (And (B (Var "p")) (Var "q")) ~=? (parse "[p && q]"),
23     (D (And (Var "p") (Var "q"))) ~=? (parse "<>(p && q)"),
24     (B (And (Var "p") (Var "q"))) ~=? (parse "[](p && q)"]
25 ]
26
27 testsGrafo = test [
28     -- Ej 1,2,4 (agregar nodos, ver nodos, grafo vacío)
29     [1] ~=? (nodos (agNodo 1 vacio)),
30     [1,2] ~=? (nodos (agNodo 2 (agNodo 1 vacio))),
31     [1,2] ~=? (nodos (agNodo 2 (agNodo 2 (agNodo 1 vacio)))),
32
33     -- Ej 3,6 (agrega ejes, ver vecinos)
34     [] ~=? (vecinos (agNodo 3 (agNodo 2 (agNodo 1 vacio))) 1),
35     [] ~=? (vecinos (agNodo 3 (agNodo 2 (agNodo 1 vacio))) 5), -- es total
36     [2] ~=? (vecinos (agEje (3,2) (agNodo 3 (agNodo 2 (agNodo 1 vacio)))) 3),
37     [2,3] ~=? (vecinos (agEje (3,3) (agEje (3,2)
38         (agNodo 3 (agNodo 2 (agNodo 1 vacio))))) 3),
39     [2] ~=? (vecinos (agEje (3,2) (agEje (3,2) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))) 3),
40     [1,2,3,4] ~=? (vecinos (agEje (5,1) (agEje (5,2) (agEje (5,3) (agEje (5,4)
41         (agNodo 5 (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))))) 5),
42
43     -- Ej 5 (sacar nodo)
44     [1,3] ~=? nodos (sacarNodo 2 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))),
45     [1,2,3] ~=? nodos (agNodo 2 (sacarNodo 2
46         (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))
47     [1] ~=? vecinos (sacarNodo 2 (agEje (3,2) (agEje (3,1) (agNodo 3 (agNodo 2 (agNodo 1
48         vacio))))) 3,
49     [] ~=? vecinos (sacarNodo 2 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))) 2,
50
51     -- Ej 7 (lineal)
52     (agEje (2,3) (agEje (1,2) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))) ~=? lineal [1,2,3],
53
54     -- Ej 8 (union de grafos)
55     -- Grafos disjuntos
56     (agEje (3,4) (agEje (1,2) (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))
57         ~=? union (agEje (3,4) (agNodo 4 (agNodo 3 vacio)))
58         (agEje (1,2) (agNodo 2 (agNodo 1 vacio))),
59
60     -- Grafo vacío
61     (agEje (1,2) (agNodo 2 (agNodo 1 vacio)))
62         ~=? union vacio
63         (agEje (1,2) (agNodo 2 (agNodo 1 vacio))),
64
65     -- Grafos lineales
66     (lineal [1,2,3,4,5,6]) ~=? union (lineal [1,2,3]) (lineal [3,4,5,6]),
67
68     -- Algunos nodos en común
69     (agEje (1,2) (agEje (1,3) (agEje (2,3) (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))
70         ~=? union (agEje (1,2) (agNodo 1 (agNodo 2 vacio)))
71         (agEje (1,3) (agEje (2,3)
72             (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))
73 ]

```

```

69 -- Grafos idénticos
70 (agEje (1,3) (agEje (2,3) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))
71   ==? union (agEje (1,3) (agEje (2,3)
72     (agNodo 3 (agNodo 2 (agNodo 1 vacio))))
73     (agEje (1,3) (agEje (2,3)
74       (agNodo 3 (agNodo 2 (agNodo 1 vacio))))),
75 -- Grafos idénticos en distinto orden
76 (agEje (1,3) (agEje (2,3) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))
77   ==? union (agEje (1,3) (agNodo 1 (agEje (2,3)
78     (agNodo 3 (agNodo 2 vacio))))
79     (agEje (1,3) (agEje (2,3)
80       (agNodo 3 (agNodo 2 (agNodo 1 vacio))))),
81
82 -- Ej 9 (clausura transitiva)
83 -- hago un grafo que es un ciclo y deberia obtener un completo con la clausura
84 [1,2,3,4] ==? vecinos (clausura (agEje (4,1) (lineal [1,2,3,4]))) 1,
85 [1,2,3,4] ==? vecinos (clausura (agEje (4,1) (lineal [1,2,3,4]))) 2,
86 [1,2,3,4] ==? vecinos (clausura (agEje (4,1) (lineal [1,2,3,4]))) 3,
87 [1,2,3,4] ==? vecinos (clausura (agEje (4,1) (lineal [1,2,3,4]))) 4
88 ]
89
90 testsLomoba = test [
91   -- Ej 11
92   0 ==? visibilidad (parse "p"),
93   1 ==? visibilidad (parse "<p"),
94   2 ==? visibilidad (parse "<!(<p)"),
95   2 ==? visibilidad (parse "<<p||<<q"),
96   3 ==? visibilidad (parse "<(<p||<<q)"),
97   3 ==? visibilidad (parse "[ ](<p&&<[ ]q)"),
98   2 ==? visibilidad (parse "<<p||<<q||<<r"),
99   0 ==? visibilidad (parse "p||q||r||s||t"),
100  10 ==? visibilidad (parse "[ ]<[ ]<[ ]<[ ]<[ ]<p"),
101  4 ==? visibilidad (parse "[ ][ ][ ](p||[ ]q||r||<s||t)"),
102
103   -- Ej 12
104   ["p"] ==? extraer (parse "p"),
105   ["p"] ==? extraer (parse "<p"),
106   ["p"] ==? extraer (parse "<!(<p)"),
107   ["p","q"] ==? extraer (parse "<<p||<<q"),
108   ["p","q"] ==? extraer (parse "<(<p||<<q)"),
109   ["p","q"] ==? extraer (parse "[ ](<p&&<[ ]q)"),
110   ["p","q","r"] ==? extraer (parse "<<p||<<q||<<r"),
111   ["p","q","r","s","t"] ==? extraer (parse "p||q||r||s||t"),
112   ["p"] ==? extraer (parse "<[ ]<[ ]<[ ]<[ ]<p"),
113   ["p","q","r","s","t"] ==? extraer (parse "[ ][ ][ ](p||[ ]q||r||<s||t)"),
114
115   -- Ej 13
116   True ==? eval modeloKrEnunciado 1 (parse "p&&[ ]q"),
117   True ==? eval modeloKrEnunciado 1 (parse "p&&<r"),
118   False ==? eval modeloKrEnunciado 1 (parse "[ ]r"),
119   True ==? eval modeloKrEnunciado 1 (parse "<(q&&r)"),
120   False ==? eval modeloKr1 1 (parse "<(q&&r)"),
121   True ==? eval modeloKr1 1 (parse "<(<r)"),
122   True ==? eval ciclo 2 (parse "<q"),
123
124   -- Ej 14
125   [1] ==? valeEn (parse "p") modeloKrEnunciado,
126   [5,4,3,2,1] ==? valeEn (parse "<p") ciclo,
127   [] ==? valeEn (parse "<!(<p)") ciclo,
128   [3,2,1] ==? valeEn (parse "<r||<q") modeloKr1,
129   [5,4,3,2,1] ==? valeEn (parse "<(<p||<<q)") ciclo,
130   [5,4,3,2] ==? valeEn (parse "!p||q||r") modeloKr1,
131
132   -- Ej 15
133   [1] ==? ((\ (K g f) -> nodos g)(quitar (parse "p&&[ ]q") modeloKrEnunciado),
134   [] ==? ((\ (K g f) -> f)(quitar (parse "p&&[ ]q") modeloKrEnunciado)) "q",
135   [1] ==? ((\ (K g f) -> f)(quitar (parse "p&&[ ]q") modeloKrEnunciado)) "p",
136
137

```

```

138
139 [2,3] ==? (\(K g f) -> nodos g)(quitar (parse "q") modeloKrEnunciado),
140 [3] ==? (\(K g f) -> f)(quitar (parse "q") modeloKrEnunciado)) "r",
141 [2,3] ==? (\(K g f) -> f)(quitar (parse "q") modeloKrEnunciado)) "q",
142
143 (\(K g f) -> g)modeloKrEnunciado ==?
144     (\(K g f) -> g)(quitar (parse "q||<>r") modeloKrEnunciado),
145
146 -- Ej 16
147 True ==? cierto ciclo (parse "p"),
148 False ==? cierto ciclo (parse "q"),
149 False ==? cierto modeloKrEnunciado (parse "q"),
150 False ==? cierto modeloKrEnunciado (parse "p&&r || q"),
151 True ==? cierto modeloKrEnunciado (parse "p || q || r"),
152 True ==? cierto ciclo (parse "p || q || r"),
153 True ==? cierto modeloKr1 (parse "p || q || r"),
154 False ==? cierto modeloKr1 (parse "[p]"),
155 False ==? cierto ciclo (parse "<>(q&&r)")
156 ]
157
158
159 -- El grafo del enunciado
160 modeloKrEnunciado = K (agEje (1,2) (agEje (1,3) (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))
161     (\p -> case () of
162         - | p=="p" -> [1]
163         - | p=="q" -> [2,3]
164         - | p=="r" -> [3]
165         - | otherwise -> []) -- debe ser total
166
167 modeloKr1 = K (agEje (1,2) (agEje (1,3) (agEje (3,4) (agEje (2,5)
168     (agNodo 5 (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio))))))))))
169     (\p -> case () of
170         - | p=="p" -> [1]
171         - | p=="q" -> [2,3]
172         - | p=="r" -> [4,5]
173         - | otherwise -> []) -- debe ser total
174
175 ciclo = K (agEje (1,2) (agEje (2,3) (agEje (3,4) (agEje (4,5) (agEje (5,1)
176     (agNodo 5 (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio))))))))))
177     (\p -> case () of
178         - | p=="p" -> [1, 2, 3, 4, 5]
179         - | p=="q" -> [3]
180         - | p=="r" -> [4]
181         - | otherwise -> []) -- debe ser total
182
183 -----
184 -- helpers --
185 -----
186
187 -- idem ==? pero sin importar el orden
188 (==?) :: (Ord a, Eq a, Show a) => [a] -> [a] -> Test
189 expected ==? actual = (sort expected) ==? (sort actual)
190 where
191     sort = foldl (\r e -> push r e) []
192     push r e = (filter (e<=) r) ++ [e] ++ (filter (e>) r)
193

```