PLP - TP Funcional

Jueves 23 de Abril de 2015 Grupo Alpha

Integrante	LU	Correo electrónico
Izcovich, Sabrina	550/11	sizcovich@gmail.com
Rama, Roberto	490/11	bertoski@gmail.com
Requeni, Gastón	400/11	grequeni@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellon I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autonoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

```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)))
```

```haskell

-- 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.
-- 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))) []

```

```haskell

-- 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,
-- 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]
```

```haskell
module Lomoba where
import Grafo
import List
import qualified Data.List (union)
import Tipos
-- ---------------------------------Sección 6--------- Lomoba --------------------------
-- Ejercicio 10
foldExp ::     (Prop -> a)    -- Función para aplicar a (Var p)
            -> (a -> a)       -- Función para aplicar a (Not e)
            -> (a -> a -> a)  -- Función para aplicar a (Or e1 e2)
            -> (a -> a -> a)  -- Función para aplicar a (And e1 e2)
            -> (a-> a)        -- Función para aplicar a (D e)
            -> (a -> a)       -- Función para aplicar a (B e)
            -> Exp -> a       -- Función que construye el fold
foldExp fVar fNot fOr fAnd fd fb (Var p) = fVar p
foldExp fVar fNot fOr fAnd fd fb (Not e) = fNot (foldExp fVar fNot fOr fAnd fd fb e)
foldExp fVar fNot fOr fAnd fd fb (Or a b) = fOr
                                            (foldExp fVar fNot fOr fAnd fd fb a)
                                            (foldExp fVar fNot fOr fAnd fd fb b)
foldExp fVar fNot fOr fAnd fd fb (And a b) = fAnd (foldExp fVar fNot fOr fAnd fd fb a)
                                              (foldExp fVar fNot fOr fAnd fd fb b)
foldExp fVar fNot fOr fAnd fd fb (D e) = fd (foldExp fVar fNot fOr fAnd fd fb e)
foldExp fVar fNot fOr fAnd fd fb (B e) = fb (foldExp fVar fNot fOr fAnd fd fb e)

-- Ejercicio 11
-- Calcula la visibilidad de la f ormula. Cada vez que aparece <> o [] debo
-- incrementar en 1 el valor de la misma.
--    * Var es el caso base, por lo que utilizare const 0.
--    * Not no sumara ningun valor, por lo que utilizare id.
--    * Or y And que seran bifurcaciones en el arbol de recursion y por lo tanto
--      tomaremos el maximo resultante de ambas ramas de la recursion.
--    * D y B seran los casos en donde tendre que sumar uno, por lo que aplicare
--      la funcion + con la valuacion parcial en el primero de sus parametros.
visibilidad :: Exp -> Integer
visibilidad = foldExp fVar fNot fOr fAnd fd fb
    where   fVar = const 0
            fNot = id
            fOr = max
            fAnd = max
            fd = (+1)
            fb= (+1)

-- Ejercicio 12
-- Extraer las variables proposicionales que aparecen en la formula, sin repetir.
--    * Var es el caso base, por lo que utilizare una lambda que dado p me devuelve [p].
--    * Or y And que seran bifurcaciones en el arbol de recursion y por lo tanto
--      utilizaremos la union de conjuntos.
--    * Not, D y B no adicionaran ningun simbolo, por lo que utilizare id.
extraer :: Exp -> [Prop]
extraer = foldExp fVar fNot fOr fAnd fd fb
    where   fVar = (\p -> [p])
            fNot = id
            fOr = Data.List.union
            fAnd = Data.List.union
            fd = id
            fb = id
```

```haskell

-- Ejercicio 13
eval :: Modelo -> Mundo -> Exp -> Bool
eval mod w exp = (eval' mod exp) w

-- Dado un modelo y una expresión, devuelve una función
-- que para un Mundo dado (en el modelo), devuelve la evaluación.
-- En los pasos de D y B, se aplica la función recursiva sobre todos
-- los mundos vecinos y se busca que en alguno o en todos la expresión
-- sea true.
eval' :: Modelo -> Exp -> (Mundo -> Bool)
eval' (K g mundosTrue) =
    foldExp
        (\p w -> w `elem` (mundosTrue p)) -- ::Prop -> (Mundo -> Bool)
        (\rec w -> not (rec w)) -- ::(Mundo -> Bool) -> Mundo -> Bool
        (\rec1 rec2 w -> (rec1 w) || (rec2 w))
        (\rec1 rec2 w -> (rec1 w) && (rec2 w))
        (\rec w -> or (map rec (vecinos g w))) -- rec::(Mundo -> Bool)
        (\rec w -> and (map rec (vecinos g w)))



-- Ejercicio 14
-- Dadas todas las variables proposicionales del grafo, se devuelven los mundos
-- que al evaluarlos dan verdadero
valeEn :: Exp -> Modelo -> [Mundo]
valeEn exp mod@(K g mundosTrue) = filter (eval' mod exp) (nodos g)

-- Ejercicio 15
-- Usando foldr, voy construyendo un nuevo modelo de Kripke, partiendo
-- del modelo pasado por argumento y sacándole los mundos donde no vale
-- la expresión. Por cada mundo que saco, le saco el nodo al grafo y
-- lo saco del valor de retorno de la función (para cada símbolo
-- proposicional)
quitar :: Exp -> Modelo -> Modelo
quitar e mod@(K g mundosTrue) =
            foldr (\w (K gRec fRec) ->
                            K (sacarNodo w gRec)
                              (\prop -> filter (/=w) (fRec prop))
                  )
                  mod                   -- Si e vale en todos los mundos, devuelvo el modelo original
                  (valeEn (Not e) mod) -- mundos donde NO vale e

-- Ejercicio 16
-- Compara el modelo original con el modelo resultante de quitarle los mundos tales
-- que no valga e.
cierto :: Modelo -> Exp -> Bool
cierto mod@(K g mundosTrue) e = (sort (nodos g) == sort (valeEn e mod))


```

```haskell
import Grafo
import Tipos
import Lomoba
import Parser
import Test.HUnit

-- evaluar t para correr todos los tests
t = runTestTT allTests

allTests = test [
    "parser" ~: testsParser,
    "grafo" ~: testsGrafo,
    "lomoba" ~: testsLomoba
    ]

testsParser = test [
    (Var "p")                       ~=? (parse "p"),
    (And (Var "p") (Var "q"))       ~=? (parse "p && q"),
    (Or (Var "p") (Var "q"))        ~=? (parse "p || q"),
    (Or (Not (Var "p")) (Var "q"))  ~=? (parse "!p || q"),
    (And (D (Var "p")) (Var "q"))   ~=? (parse "<>p && q"),
    (And (B (Var "p")) (Var "q"))   ~=? (parse "[]p && q"),
    (D (And (Var "p") (Var "q")))   ~=? (parse "<>(p && q)"),
    (B (And (Var "p") (Var "q")))   ~=? (parse "[](p && q)")]

testsGrafo = test [
    -- Ej 1,2,4 (agregar nodos, ver nodos, grafo vacío)
    [1] ~~? (nodos (agNodo 1 vacio)),
    [1,2] ~~? (nodos (agNodo 2 (agNodo 1 vacio))),
    [1,2] ~~? (nodos (agNodo 2 (agNodo 2 (agNodo 1 vacio)))),

    -- Ej 3,6 (agrega ejes, ver vecinos)
    [] ~~? (vecinos (agNodo 3 (agNodo 2 (agNodo 1 vacio))) 1),
    [] ~~? (vecinos (agNodo 3 (agNodo 2 (agNodo 1 vacio))) 5), -- es total
    [2] ~~? (vecinos (agEje (3,2) (agNodo 3 (agNodo 2 (agNodo 1 vacio)))) 3),
    [2,3] ~~? (vecinos (agEje (3,3) (agEje (3,2)
                  (agNodo 3 (agNodo 2 (agNodo 1 vacio))))) 3),
    [2] ~~? (vecinos (agEje (3,2) (agEje (3,2) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))) 3),
    [1,2,3,4] ~~? (vecinos (agEje (5,1) (agEje (5,2) (agEje (5,3) (agEje (5,4)
                     (agNodo 5 (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))))))) 5),

    -- Ej 5 (sacar nodo)
    [1,3] ~~? nodos (sacarNodo 2 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))),
    [1,2,3] ~~? nodos (agNodo 2 (sacarNodo 2
                         (agNodo 3 (agNodo 2 (agNodo 1 vacio))))),
    [1] ~~? vecinos (sacarNodo 2 (agEje (3,2) (agEje (3,1) (agNodo 3 (agNodo 2 (agNodo 1
    vacio))))))) 3,
    [] ~~? vecinos (sacarNodo 2 (agNodo 3 (agNodo 2 (agNodo 1 vacio)))) 2,

    -- Ej 7 (lineal)
    (agEje (2,3) (agEje (1,2) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))) ~=? lineal [1,2,3],

    -- Ej 8 (union de grafos)
    -- Grafos disjuntos
    (agEje (3,4) (agEje (1,2) (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio))))))
                         ~=? union (agEje (3,4) (agNodo 4 (agNodo 3 vacio)))
                                   (agEje (1,2)(agNodo 2 (agNodo 1 vacio))),
    -- Grafo vacio
    (agEje (1,2) (agNodo 2 (agNodo 1 vacio)))
                         ~=? union vacio
                                   (agEje (1,2)(agNodo 2 (agNodo 1 vacio))),
    -- Grafos lineales
    (lineal [1,2,3,4,5,6]) ~=? union (lineal [1,2,3]) (lineal [3,4,5,6]),
    -- Algunos nodos en común
    (agEje (1,2) (agEje (1,3) (agEje (2,3) (agNodo 3 (agNodo 2 (agNodo 1 vacio))))))
                         ~=? union (agEje (1,2) (agNodo 1 (agNodo 2 vacio)))
                                   (agEje (1,3) (agEje (2,3)
                                         (agNodo 3 (agNodo 2 (agNodo 1 vacio))))),

```

```
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
```

```haskell

    [2,3] ~~? (\(K g f) -> nodos g)(quitar (parse "q") modeloKrEnunciado),
    [3] ~~? ((\(K g f) -> f)(quitar (parse "q") modeloKrEnunciado)) "r",
    [2,3] ~~? ((\(K g f) -> f)(quitar (parse "q") modeloKrEnunciado)) "q",

    (\(K g f) -> g)modeloKrEnunciado ~=?
                (\(K g f) -> g)(quitar (parse "q||<>r") modeloKrEnunciado),

    -- Ej 16
    True ~=? cierto ciclo (parse "p"),
    False ~=? cierto ciclo (parse "q"),
    False ~=? cierto modeloKrEnunciado (parse "q"),
    False ~=? cierto modeloKrEnunciado (parse "p&&r || q"),
    True ~=? cierto modeloKrEnunciado (parse "p || q || r"),
    True ~=? cierto ciclo (parse "p || q || r"),
    True ~=? cierto modeloKr1 (parse "p || q || r"),
    False ~=? cierto modeloKr1 (parse "[]p"),
    False ~=? cierto ciclo (parse "<>(q&&r)")
    ]


-- El grafo del enunciado
modeloKrEnunciado = K (agEje (1,2) (agEje (1,3) (agNodo 3 (agNodo 2 (agNodo 1 vacio)))))
                        (\p -> case () of
                                  _ | p=="p" -> [1]
                                    | p=="q" -> [2,3]
                                    | p=="r" -> [3]
                                    | otherwise -> []) -- debe ser total

modeloKr1 = K (agEje (1,2) (agEje (1,3) (agEje (3,4) (agEje (2,5)
                (agNodo 5 (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio))))))))
                        (\p -> case () of
                                  _ | p=="p" -> [1]
                                    | p=="q" -> [2,3]
                                    | p=="r" -> [4,5]
                                    | otherwise -> []) -- debe ser total

ciclo = K (agEje (1,2) (agEje (2,3) (agEje (3,4) (agEje (4,5) (agEje (5,1)
                (agNodo 5 (agNodo 4 (agNodo 3 (agNodo 2 (agNodo 1 vacio))))))))))
                        (\p -> case () of
                                  _ | p=="p" -> [1, 2, 3, 4, 5]
                                    | p=="q" -> [3]
                                    | p=="r" -> [4]
                                    | otherwise -> []) -- debe ser total

---------------
--  helpers  --
---------------

-- idem ~=? pero sin importar el orden
(~~?) :: (Ord a, Eq a, Show a) => [a] -> [a] -> Test
expected ~~? actual = (sort expected) ~=? (sort actual)
    where
        sort = foldl (\r e -> push r e) []
        push r e = (filter (e<=) r) ++ [e] ++ (filter (e>) r)

```