DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Programación funcional.

Paradigmas y lenguajes de programación

Grupo: Zamba cálculo

Integrante	LU	Correo electrónico
Leandro Vega	698/11	leandrovega@gmail.com
Ignacio Niesz	722/10	ignacio.niesz@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

Lógica modal y modelo de Kripke

Ejercicio 1

```
-- Define un grafo sin nodos y una funcion total a lista vacia.
vacio :: Grafo a
vacio = G [] (\_ -> [])
```

Ejercicio 2

```
-- Devuelve los nodos de un grafo.
nodos :: Grafo a -> [a]
nodos (G ns r) = ns
```

Ejercicio 3

```
-- Devuelve los nodos relacionados al nodo pasado como argumento.
vecinos :: Grafo a -> a -> [a]
vecinos (G ns r) n = r n
```

Ejercicio 4

```
-- Genera un nuevo grafo agregando un nodo recibido como argumento
-- a un grafo recibido como argumento.
agNodo :: Eq a => a -> Grafo a -> Grafo a
agNodo n (G ns r) = G ( if elem n ns then ns else n:ns ) r
```

Ejercicio 5

```
-- Genera un nuevo grafo quitando un nodo recibido como argumento
-- a un grafo recibido como argumento.
sacarNodo :: Eq a => a -> Grafo a -> Grafo a
sacarNodo n (G ns r) = G (quitar n ns) (\x -> quitar n (r x))
            where quitar n = filter (/=n)
```

Ejercicio 6

```
-- Genera un nuevo grafo agregando una arista (origen, destino)
-- recibida como argumento a un grafo recibido como argumento.
agEje :: Eq a => (a,a) -> Grafo a -> Grafo a
agEje (n1,n2) (G ns r) = G ns ( if (elem n1 ns && elem n2 ns)then
                      (\x -> agEnN1 x n1 n2)
                    else
                      r)
            where agEnN1 x n1 n2 = if x == n1 then
                                agN2SiNoEsta n2 (r x)
                              else
                                r x
                  where agN2SiNoEsta n ns = if elem n ns then
                                        ns
                                      else
                                        n:ns
```

Ejercicio 7

```
-- Genera un nuevo grafo en base a una lista de nodos. El grafo
-- es una lista simplemente enlazada siguiendo el orden de la lista .
lineal :: Eq a => [a] -> Grafo a
lineal xs = foldl f vacio (connectPairs (pairing xs))
    where f = (\recG xs -> if length xs == 2 then
                  agEje (head xs, head (tail xs))
                  (agNodo (head (tail xs))
                    (agNodo (head xs) recG)
                    )
                else
                  agNodo (head xs) recG
          )


-- Funcion auxiliar que genera elementos intermedios que conectan
-- dos elementos de una lista de pares.
connectPairs :: Eq a => [[a]] -> [[a]]
connectPairs xss = case xss of
       [] -> []
       [[]] -> []
       [[x]] -> [[x]]
       _ -> foldl (\recur xs ->
                  if length (recur) > 0 then
                    recur ++ [[last (last recur), head xs]] ++ [xs]
                  else
                    recur ++ [xs]
          ) [] xss


-- Funcion auxiliar que transforma una lista de elementos en una
-- lista de listas de pares de elementos.
pairing :: Eq a => [a] -> [[a]]
pairing xs = foldl (\recur x -> (headpaired recur) ++ [(lastunpaired recur) ++ [x]] ) [[]] xs


-- Funcion auxiliar que devuelve la lista de listas de pares de elementos
-- hasta el primer elemento unitario de la lista .
headpaired :: Eq a => [[a]] -> [[a]]
headpaired xxs = (\xss -> if null xss then
                  []
                else
                  xss
          ) (takeWhile ((\xs -> length xs == 2)) xxs)


-- Funcion auxiliar que devuelve el primer elemento de una lista de listas
-- de pares de elementos que no esta en un par.
lastunpaired :: Eq a => [[a]] -> [a]
lastunpaired xxs = (\xss -> if null xss then
                  []
                else
                  head xss
          ) (dropWhile ((\xs -> length xs == 2)) xxs)
```

Ejercicio 8

```
-- Genera la union de dos grafos recibidos como parametro.
union :: Eq a => Grafo a -> Grafo a -> Grafo a
union (G ns1 r1) (G ns2 r2) = foldr agEje (foldr agNodo (G ns1 r1) ns2) (obtenerListaDeEjesR2)
```

```
  where obtenerListaDeEjesR2 = foldr (++) [] (listasDeListas)
  -- listasDeListas tiene una lista de nodos que estan en ns2, en donde cada
  -- nodo tiene una lista de tuplas de el con cada uno de sus vecinos
    where listasDeListas = foldr (\x c -> (obtenerVecinos x):c) [] ns2
    -- armo lista con la tupla mencionada en el comentario de arriba
      where obtenerVecinos x = foldr (\y b -> (x,y):b) [] (r2 x)
```

Ejercicio 9

```
-- Genera un grafo clausurando la relacion del grafo recibido como
-- argumento.
-- La funcion pide que el grafo sea de elementos ordenados para
-- facilitar el uso de puntofijo (que la igualdad entre conjuntos
-- sea la igualdad de listas ordenandolas).
clausura :: (Ord a) => Grafo a -> Grafo a
clausura (G ns r) = G ns (clausurar ns r)

-- Genera una nueva relacion que es la clausura de la relacion
-- recibida como argumento. Ademas recibe como argumento la
-- lista de elementos del grafo para poder representar la relacion
-- con una funcion total.
-- La clausura se genera aplicando punto fijo a la funcion g
-- de la relacion extendida.
-- (ver extenderR :: (Ord a) => (a -> [a]) -> ([a] -> [a]))
-- La funcion g filtra los elementos que no pertenecen al grafo
-- y para los que pertenecen al grafo agrega reflexividad
-- y une el resultado
clausurar :: (Ord a) => [a] -> (a -> [a]) -> (a -> [a])
clausurar ns r = (\a -> (puntofijo ( g (extenderR r) )) [a] )
  where g = (\eR res -> if (eR res) == [] then
                 filter (\n -> elem n ns) res
               else
                 List.sort $ List.nub (res ++ (eR res))
         )

-- Define la aplicacion infinita de f, [f, f.f, f.f.f, f.f.f.f, ...]
infiniteComposition :: (a -> a) -> [a -> a]
infiniteComposition f = iterate (h f) f
  where h = (\f g -> f . g)


-- Realiza la aplicacion infinita de f hasta que pf(a) = a donde pf
-- es la ultima composicion generada.
-- Punto fijo se define sobre elementos que aceptan un orden para que sea
-- mas legible. Esto permite definir puntofijo sobre funciones que van
-- de conjuntos en conjuntos (listas ordenadas) y usar la igualdad de
-- listas (==).
-- Ej:
-- r [1] == [1, 2, 3]
-- r [2] == [2, 4]
-- r [3] == [3]
-- r [4] == [4]
-- r [5] == [5]
-- r _ == []
-- (puntofijo r) [1]
-- > (puntofijo r) [1,2,3]
-- > (puntofijo r) [1,2,3,4]
-- > (puntofijo r) [1,2,3,4,5] == [1,2,3,4,5]
-- Entonces, (puntofijo r) [1] == [1,2,3,4,5]
puntofijo :: (Ord a) => (a -> a) -> a -> a
puntofijo f a = f ( compose f a)
    where compose = (\f a -> lastComposition (takeWhile (pred f a) $ infiniteComposition f) a)
```

```
        lastComposition = (\pfs -> if null pfs then
                             id
                           else
                             last pfs)
        pred = (\f a pf -> not (((f . pf) a) == pf a))


-- Extiende la definicion de una relacion definida como una
-- funcion a elementos con los que se relaciona como una funcion
-- de conjuntos en conjuntos (listas ordenadas sin repetidos ).
-- Ej:
-- r 1 == [2,3] --> (extenderR r) [1] == [2,3]
-- r 2 == [2,4] --> (extenderR r) [2] == [2,4]
-- r 4 == [5] --> (extenderR r) [4] == [5]
-- r _ == [] --> (extenderR r) _ == []
-- Ademas,
-- (extenderR r) [1,2] == [2,3,4]
-- (extenderR r) [2,4] == [2,4,5]
-- (extenderR r) [1,2,4,9999] == [2,3,4,5]
extenderR :: (Ord a) => (a -> [a]) -> ([a] -> [a])
extenderR r = foldr (g r) []
  where g = (\r n recur -> List.sort $ List.nub $ r n ++ recur)
```

Ejercicio 10

```
-- Ejercicio 10
-- Fold para la estructura de expresiones de logica modal. El fold utiliza una funcion por cada
-- generador del tipo Exp.
foldExp :: (Prop -> a) -> (a -> a) -> (a -> a -> a) -> (a -> a -> a) -> (a -> a) -> (a -> a)
-> Exp -> a
foldExp fVar fNot fOr fAnd fD fB exp =
  case exp of
    (Var p) -> fVar p
    (Not e) -> fNot (foldExp fVar fNot fOr fAnd fD fB e)
    (Or e1 e2) -> fOr (foldExp fVar fNot fOr fAnd fD fB e1) (foldExp fVar fNot fOr fAnd fD fB e2)
    (And e1 e2) -> fAnd (foldExp fVar fNot fOr fAnd fD fB e1) (foldExp fVar fNot fOr fAnd fD fB e2)
    (D e) -> fD (foldExp fVar fNot fOr fAnd fD fB e)
    (B e) -> fB (foldExp fVar fNot fOr fAnd fD fB e)
```

Ejercicio 11

```
-- Funcion que devuelve en nivel de visibilidad de una expresion de logica modal.
-- Suma uno por cada cuantificador y se queda con el maximo en oepraciones and y or.
visibilidad :: Exp -> Integer
visibilidad e = foldExp fVar fNot fOr fAnd fD fB e
    where fVar = const 0
          fNot = const 0
          fOr = max
          fAnd = max
          fD = (1+)
          fB = (1+)
```

Ejercicio 12

```
-- Extrae las variables propocicionales de una expresion de logica modal.
extraer :: Exp -> [Prop]
```

```
extraer e = List.nub (foldExp lista id (++) (++) id id e)
  where lista x = [x]
```

Ejercicio 13

```
-- Funcion que dado un modelo y un mundo permite evaluar una expresion. Cada
-- funcion del fold recibe un parametro extra para especificar en que mundo
-- debe evaluarse la expresion. Notemos que los operadores de la logica modal
-- cambian el mundo donde debe evaluarse la subexpresion.
eval :: Modelo -> Mundo -> Exp -> Bool
eval (K g v) w exp = (foldExp fVar fNot fOr fAnd fD fB exp) w
    where fVar = (\p w -> elem w (v p) )
          fNot = (\expRec w -> not $ expRec w )
          fOr = (\expRec1 expRec2 w -> expRec1 w || expRec2 w )
          fAnd = (\expRec1 expRec2 w -> expRec1 w && expRec2 w )
          fD = (\expRec w -> foldr (\w' recur -> (expRec w') || recur) False (vecinos g w) )
          fB = (\expRec w -> foldr (\w' recur -> (expRec w') && recur) True (vecinos g w) )
```

Ejercicio 14

```
-- Funcion que genera una lista de los mundos donde evaluo correctamente la expresion
-- dado un determinado modelo.
valeEn :: Exp -> Modelo -> [Mundo]
valeEn exp (K g v) = foldr f [] (nodos g)
  where f = (\w ws -> if (eval (K g v) w exp) then
                w:ws
              else
                ws
        )
```

Ejercicio 15

```
-- Funcion que dada una expresion y un modelo, genera un modelo removiendo todos los
-- mundos donde no es cierta la expresion (esto es analogo a decir donde es cierta su negacion).
-- Los mundos son removidos tanto del listado del Modelo como de la relacion.
quitar :: Exp -> Modelo -> Modelo
quitar exp (K g v) = foldr f (K g v) (valeEn (Not exp) (K g v) )
    where f = (\w (K recG recV) -> K (sacarNodo w recG) (h w recV) )
          h = (\w recV p -> if elem p (extraer exp) then
                  List.delete w (recV p)
                else
                  recV p
        )
```

Ejercicio 16

```
-- Funcion que dado un modelo y una expresion devuelve verdadero si la expresion es cierta
-- en todos los mundos del modelo.
cierto :: Modelo -> Exp -> Bool
cierto (K g v) e = cantidad (nodos g) == cantidad (valeEn e (K g v))
  where cantidad = foldr (+) 0
```