DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Programación funcional.

Paradigmas y lenguajes de programación

Grupo: Java decime que se siente

Integrante	LU	Correo electrónico
Leandro Matayoshi	79/11	leandro.matayoshi@gmail.com
Ignacio Niesz	722/10	ignacio.niesz@gmail.com
Raul Benitti	592/08	raulbenitti@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

Oogle Amps

Ejercicio 1

```
-- Define una funcion que recibe una clave y un diccionario. Esta funcion utiliza filter
-- para filtrar la clave buscada de las claves del diccionario. La funcion se asegura
-- que la lista de claves filtradas no sea vacio.
belongs :: Eq k => k -> Dict k v -> Bool
belongs = \k dict -> not ( null (filter (\e -> fst e == k) dict) )

-- Invierte el orden en que belongs recibe los parametros para que tenga la funcionalidad
-- pedida.
(?) :: Eq k => Dict k v -> k -> Bool
(?) = flip belongs


--Main> [("calle",[3]),("city",[2,1])] ? "city"
--True
```

Ejercicio 2

```
-- Define una funcion que recibe una clave y un diccionario. Esta funcion utiliza filter
-- para filtrar de las tuplas que componen el diccionario, la que concuerdo con la clave
-- buscada.
get :: Eq k => k -> Dict k v -> v
get = \k dict -> snd ((filter (\(x,y) -> x==k) dict) !! 0)


-- Invierte el orden en que get recibe los parametros para que tenga la funcionalidad
-- pedida.
(!) :: Eq k => Dict k v -> k -> v
(!) = flip get


--Main> [("calle",[3]),("city",[2,1])] ! "city"
--[2,1]
```

Ejercicio 3

```
-- Define una funcion que recibe una funcion para combinar los valores del diccionario
-- que poseen una misma clave, una clave del diccionario y un diccionario. La funcion se
-- asegura de modificar el diccionario en caso de que la clave exista o de insertar
-- un nuevo elemento en caso contrario.
insertWith :: Eq k => (v -> v -> v) -> k -> v -> Dict k v -> Dict k v
insertWith = \f k v dict -> if dict ? k then
                              updateDict f k v dict
                            else
                              insertInDict k v dict

-- Utiliza foldr para componer el valor insertado que corresponde a una clave ya existente.
-- El foldr se define recursivamente sobre los elementos del diccionario (una lista de tuplas,
-- (clave, valor)) asegurandose de llamar a la funcion para componer el resultado sobre los
-- elementos de la lista que tengan la misma clave que la pasada por parametro. El resto
-- de los elementos no se alteran.
updateDict :: Eq k => (v -> v -> v) -> k -> v -> Dict k v -> Dict k v
updateDict f k v dict = foldr (\e rec -> if fst e == k then (k, (f (snd e) v )) : rec else e:rec ) [] dict

-- Simplemente se agrega el nuevo elemento al diccionario.
insertInDict :: Eq k => k -> v -> Dict k v -> Dict k v
```

```
insertInDict  k v  dict  = dict  ++ [(k,v)]


--Main> insertWith (++) 2 ['p'] (insertWith (++) 1 ['a','b'] (insertWith (++) 1 ['l'] []))
--[(1,"lab"),(2,"p")]
```

Ejercicio 4

```
-- Define una funcion que recibe una lista de tuplas (clave, valor) y devuelve un diccionario.
-- La funcion se define mediante foldl que recibe la funcion insertWith para generar recursivamente
-- un diccionario en base a los elementos de la lista de tuplas. Notemos que en caso de coliciones
-- en las claves utilizara la concatenacion como funcion para componer los valores (agrupandolos
-- por clave).
groupByKey :: Eq k => [(k,v)] -> Dict k [v]
groupByKey = \l -> foldl (\rec (k,v)-> insertWith (++) k [v] rec) []  l


--Main> groupByKey [(1,"lab"), (1,"lab"),(2,"p")]
--[(1,["lab","lab "]),(2,[" p "])]
```

Ejercicio 5

```
-- Define una funcion que recibe una funcion de "valores, en valores en valores" y dos diccionarios.
-- La funcion se define mediante foldl que recibe la funcion insertWith que a su vez recibe la
-- funcion utilizada para componer los valores de los diccionarios. El foldl realiza recursion sobre
-- la concatenacion de ambos diccionarios generando como resultado un nuevo diccionario construido
-- a partir de los insertWith con su respectiva funcion para componer valores.
unionWith :: Eq k => (v -> v -> v) -> Dict k v -> Dict k v -> Dict k v
unionWith = \f d1 d2 -> foldl (\rec (k, v) -> insertWith f k v rec)  []  (d1 ++ d2)


--Main> unionWith (++) [("calle",[3]),("city",[2,1])] [("calle",  [4]),  ("altura",  [1,3,2])]
--[("calle ",[3,4]),(" city ",[2,1]),(" altura ",[1,3,2])]
```

Ejercicio 6

```
-- Se define mediante foldl. foldl recibe una funcion encargada de tomar la cola de la recursion
-- y concatenarla la cabeza a la que le agrega un elemento. foldl realiza la recursion sobre la lista
-- de procesos y tiene como caso base una lista de listas con tantos elementos como procesadores a utilizar.
-- De esta manera recorre cada elemento rotando las listas a las cuales debe agregarlos.
-- La idea subyacente es la siguiente: La funcion combinadora se encarga de mantener el resultado recursivo
-- ordenado de manera creciente en cantidad de elementos (en la primera posicion se ubicara siempre
-- el elemento con menor cantidad de elementos). De esta manera, luego de insertar un elemento en la
-- lista que ocupa la primera posicion, la misma debe ser enviada al final para que los nuevos elementos
-- sean insertados en las listas siguientes.
-- Ejemplo con distribucion de elementos en 3 listas:
-- Inicio: [1,2,3,4]  [  [],  [],  []  ]
-- Paso 1: [2,3,4] [  [],  [],  [1]  ]
-- Paso 2: [3,4] [  [],  [1],  [2]  ]
-- Paso 3: [4] [  [1],  [2],  [3]  ]
-- Paso 4: [  [2],  [3],  [1,4]  ]

distributionProcess  :: Int -> [a] -> [[a]]
distributionProcess  n l = foldl (\rec e -> (tail rec) ++ [(head rec) ++ [e]] ) (replicate n []) l
```

```
-- Main> distributionProcess 2 ["p1", "p2", "p3", "p4", "p5", "p6", "p7", "p8", "p9", "p10"]
-- [["p1","p3","p5","p7","p9"],["p2","p4","p6","p8","p10"]]

-- Main> distributionProcess 3 ["p1", "p2", "p3", "p4", "p5", "p6", "p7", "p8", "p9", "p10"]
-- [["p2","p5","p8"],["p3","p6","p9"],["p1","p4","p7","p10"]]
```

Ejercicio 7

```
-- Se define mediante la aplicacion de la funcion de mapeo (Mapper) utilizando map sobre la lista
-- de valores y concatenando los resultados (la lista de listas de tuplas clave valor). A su
-- vez este resultado se agrupa por clave utilizando groupByKey.
mapperProcess :: Eq k => Mapper a k v -> [a] -> [(k,[v])]
mapperProcess f l = groupByKey ( concat (map f l) )


-- Main> mapperProcess (\x -> [(show x, x)]) [1, 1, 1, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 5]
--     [("1",[1,1,1]),("2",[2]),("3",[3,3]),("4",[4]),("5",[5,5,5,5,5,5,5,5])]
```

Ejercicio 8

```
-- Se define como la aplicacion de map sobre la concatenacion de la lista de listas , agrupadas
-- y ordenadas por clave. Al agrupar por clave, se generan tuplas (k, [[v]]). Finalmente el map
-- se encarga de enviar cada tupla (k, [[v]]) a la correspondiente (k, [v]).
combinerProcess :: (Eq k, Ord k) => [[(k, [v])]] -> [(k,[v])]
combinerProcess ls = map (\(k, l) -> (k, concat l)) (sortByKey (groupByKey ( concat ls )))


-- Main> combinerProcess [[("1",[1,1,1]), ("2",[2,2])],  [("3",[3,3,3,3,3,3])],  [("1",[1,1,1])]]
--   [("1",[1,1,1,1,1,1]),("2",[2,2]),("3",[3,3,3,3,3,3])]


-- Se define como el sortBy de la lista de tuplas (k, [v]) utilizando como funcion de comparacion
-- una funcion que compara entre claves de tuplas.
sortByKey::(Eq k, Ord k) => [(k, [v])] -> [(k,[v])]
sortByKey = sortBy ( (\e1 e2 -> compare (fst e1) (fst e2) ) )
```

Ejercicio 9

```
-- Se define como la concatenacion del resultado de la funcion map que aplica la funcion "Reducer"
-- a todos los elementos de la lista de tuplas (k, [v]).
reducerProcess :: Reducer k v b -> [(k, [v])] -> [b]
reducerProcess f l = concat ( map f l)


-- -Main> reducerProcess (\(k, v) -> [(k, sum v) ])  [("1",[1,1,1,1,1,1]),("2",[2,2]),("3",[3,3,3,3,3,3])]
-- [("1",6),("2",4),("3",18)]
```

Ejercicio 10

```
-- Primero se mapea la funcion resultante de aplicar mapperProcess a la funcion mapper a la lista de listas obtenia
-- al utilizar distributionProcess en 100 maquinas. Luego se combinan los resultados con combinerProcess y finalmente
-- se aplica el reducer utilizando el reducerProcess.
mapReduce :: (Eq k, Ord k) => Mapper a k v -> Reducer k v b -> [a] -> [b]
mapReduce mapper reducer l = reducerProcess reducer (
```

```
        combinerProcess (map (mapperProcess mapper) (distributionProcess 100 l) )
          )


-- Main> mapReduce (\x -> [(show x, x)]) (\(k, v) -> [(k, length v) ]) [1, 1, 1, 2, 3, 3, 4, 5, 5, 5, 5, 5, 5, 5, 5]
-- [("1",3),("2",1),("3",2),("4",1),("5",8)]
```

Ejercicio 11

```
-- Se define aplicando mapReduce al mapper, reducer y la lista de monumentos. El mapper mapea cada elemento como
-- una tupla con (nombre, cantidad) que luego se reducen agrupando la cantidad total por clave.
visitasPorMonumento :: [String] -> Dict String Int
visitasPorMonumento = \lin -> mapReduce mapper reducer lin
 where mapper = \e -> [(e, 1)]
       reducer = \(k, l) -> [(k, (length l))]


-- Main> visitasPorMonumento ["m1", "m1", "m1", "m1", "m1", "m2", "m2", "m2", "m3", "m4"]
-- [("m1",5),("m2",3),("m3",1),("m4",1)]
```

Ejercicio 12

```
-- Aprovecha que las claves se devuelven ordenadas crecientemente y el mapper utiliza el valor negativo
-- de las visitas como clave para ordenarlas por mas visitados.
-- Luego las claves se descartan y se devuelven los valores.
-- Idea-ejemplo:
-- ["m1", "m2", "m2"]
--   visitasPorMonumentos --> [("m1", 1),("m2", 2)]
--     mapper->[(-1, "m1"), (-2, "m2")]
--       orderByKey --> [(-2, "m2"), (-1, "m1")]
--         reducecer -> ["m2", "m1"]
monumentosTop:: [String] -> [String]
monumentosTop = \ls -> mapReduce mapper reducer ( visitasPorMonumento ls )
  where mapper = \(k, v) -> [(-v, k)]
        reducer = \(k, l) -> l

-- Main> monumentosTop ["m1", "m1", "m1", "m1", "m1", "m2", "m2", "m2", "m3", "m4"]
-- ["m1","m2","m3","m4"]
```

Ejercicio 13

```
-- Se define igual que los anteriores con la particularidad de que el mapper utiliza pattern matching.
-- El mapper se encargara de mappear un 1 en el pais de cada uno de los monumentos mientras que
-- ignorara las estructuras de tipo ciudad o calle. El reducer se encargara de combinar sumar
-- todas las apariciones devolviendo el valor deseado.
monumentosPorPais :: [(Structure, Dict String String)] -> [(String, Int)]
monumentosPorPais = \lin -> mapReduce mapper reducer lin
  where mapper (Monument, dict) = [(dict!"country", 1)]
        mapper (City, dict) = []
        mapper (Street, dict) = []
        mapper _ = error "fst(argumento) no es un Structure"
        reducer = \(pais, l) -> [(pais, (length l))]


-- ---------------------- Ejemplo de datos del ejercicio 13 ----------------------
```

```
16 data Structure = Street | City | Monument deriving Show
17
18 isMonument:: Structure -> Bool
19 isMonument Street = False
20 isMonument City = False
21 isMonument Monument = True
22
23 items :: [(Structure, Dict String String)]
24 items = [
25     (Monument, [
26       ("name","Obelisco"),
27       ("latlong","-36.6033,-57.3817"),
28       ("country", "Argentina")]),
29     (Street, [
30       ("name","Int. Guiraldes"),
31       ("latlong","-34.5454,-58.4386"),
32       ("country", "Argentina")]),
33     (Monument, [
34       ("name", "San Martin"),
35       ("country", "Argentina"),
36       ("latlong", "-34.6033,-58.3817")]),
37     (City, [
38       ("name", "Paris"),
39       ("country", "Francia"),
40       ("latlong", "-24.6033,-18.3817")]),
41     (Monument, [
42       ("name", "Bagdad Bridge"),
43       ("country", "Irak"),
44       ("new_field", "new"),
45       ("latlong", "-11.6033,-12.3817")])
46     ]
47
48
49 ----------------------------------------------------
50 ----------------------------------------------------
51 -- Funciones para el Test "lista de monumentos"
52 mapperListaMonus (Monument, dict) = [(1, dict!"name")]
53 mapperListaMonus (Street, _) = []
54 mapperListaMonus (City, _) = []
55 reducerListaMonus = \(k, l) -> l
56
57 mapperCuentaLetras = \s -> foldr (\e rec -> (e,[1]) : rec) [] s
58 reducerCuentaLetras = \(k, l) -> [(k, (length l))]
```