DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Funcional

Paradigmas de Lenguajes de Programación

Segundo Cuatrimestre de 2014

Alumno	LU	E-mail
Almansi, Emilio Guido	674/12	ealmansi@gmail.com
Arjovsky, Martín	683/12	martinarjovsky@gmail.com

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

1. Código fuente

1.1. Solución

```
module MapReduce where

import Data.Ord
import Data.List

-- ---------------------------------Sección 1: Diccionario-------------------------------------
type Dict k v = [(k,v)]


-- Ejercicio 1

belongs :: Eq k => k -> Dict k v -> Bool
-- obtiene una lista con las keys del diccionario, y
-- verifica si la key provista es elemento de la lista

belongs k d = elem k $ map fst $ d

(?) :: Eq k => Dict k v -> k -> Bool
(?) = flip belongs


-- Ejercicio 2

get :: Eq k => k -> Dict k v -> v
-- precondicion: belongs k d == True
-- filtra todos los pares del diccionario donde la key no es la
-- key provista. en principio, el resultado podria tener 0 o 1 elementos,
-- pero dada la precondicion, el resultado tendra necesariamente 1 elemento.
-- al tomar head de la lista obtenida, se obtiene el par (k, v) correspondiente,
-- siendo el resultado final el value v

get k d = snd $ head $ filter (\kv -> (fst kv) == k) $ d

(!) :: Eq k => Dict k v -> k -> v
(!) = flip get


-- Ejercicio 3

insertWith :: Eq k => (v -> v -> v) -> k -> v -> Dict k v -> Dict k v
-- si la key no esta definida en el diccionario, simplemente se agrega el par
-- (k, v) al mismo.
-- de lo contrario, se mapea cada value del diccionario a si mismo, salvo
-- el value de la key que se esta modificando, el cual recibe el nuevo valor
-- de f(value_anterior, value_nuevo)

insertWith f k v d
  | not $ d ? k     = (k, v) : d
  | otherwise       = map (\kv -> newValue kv) d
  where
    newValue kv = if (fst kv) /= k then kv else (fst kv, f (snd kv) v)


-- Ejercicio 4
```

```
groupByKey :: Eq k => [(k,v)] -> Dict k [v]
-- obtiene una lista de todos los keys en la lista de entrada removiendo duplicados
-- luego, el resultado es una lista de pares donde a cada key unica ‘k‘ le asigna un listado
-- con todos los valores que aparecen en un par asociado con ‘k‘

groupByKey xs = [(k, valuesOf k) | k <- keys xs]
  where
    keys xs = nub $ map fst $ xs
    valuesOf k = map snd $ filter (\kv -> (fst kv) == k) $ xs


-- Ejercicio 5

unionWith :: Eq k => (v -> v -> v) -> Dict k v -> Dict k v -> Dict k v
-- devuelve una lista donde, por cada key ‘k‘ en la union de los keys
-- de los diccionarios de entrada ‘d1‘ y ‘d2‘, se incluye un par (k, v)
-- con el valor correspondiente a ‘k‘ si este aparece en un unico diccionario,
-- o con el valor f(v1, v2) si aparece en ambos

unionWith f d1 d2 = [(k, valueOf k) | k <- union (keys d1) (keys d2)]
  where
    keys d = map fst d
    valueOf k
      | (d1 ? k) && (d2 ? k)  = f (d1 ! k) (d2 ! k)
      | (d1 ? k)              = (d1 ! k)
      | otherwise             = (d2 ! k)


-- ----------------------------Sección 2 : MapReduce----------------------------------------

type Mapper a k v = a -> [(k,v)]
type Reducer k v b = (k, [v]) -> [b]
data Structure = Street | City | Monument deriving Show

-- Ejercicio 6

distributionProcess :: Int -> [a] -> [[a]]
-- a cada elemento x de xs lo transforma en (i, x), donde i es el
-- bucket que le corresponde. luego, la respuesta es una comprensión
-- de n buckets, cada uno de ellos con los elementos que le corresponden.

distributionProcess n xs = [bucketElements i | i <- [1..n]]
  where
    ixs = zip [1 + mod (i - 1) n | i <- [1..length xs]] xs
    bucketElements i = map snd $ filter (\ix -> (fst ix) == i) $ ixs


-- Ejercicio 7

mapperProcess :: Eq k => Mapper a k v -> [a] -> [(k,[v])]
-- aplica la funcion mp a cada elemento de xs, generando
-- muchas listas de tipo [(k, v)]. luego las concatena en
-- una unica lista y aplica la funcion groupByKey para
-- agrupar los pares (k, v) por clave

mapperProcess mp xs = groupByKey $ concat $ map mp $ xs
```

```
-- Ejercicio 8

combinerProcess :: (Eq k, Ord k) => [[(k, [v])]] -> [(k,[v])]
-- dada la lista con el resultado del mapperProcess 'ejecutado'
-- en cada maquina, une los resultados en un unico diccionario
-- donde a cada clave se le asigna la concatenacion de todos sus
-- values en todos los resultados parciales

combinerProcess xs = foldr (\x r -> unionWith (++) x r) [] $ xs


-- Ejercicio 9

reducerProcess :: Reducer k v b -> [(k, [v])] -> [b]
-- aplica la funcion reducer a cada par (k, [v]) producto del
-- proceso de combinacion, generando en cada caso una lista de
-- resultados. luego, se concatenan los resultados generados para
-- todas las claves

reducerProcess rd xs = concat $ map rd $ xs


-- Ejercicio 10

mapReduce :: (Eq k, Ord k) => Mapper a k v -> Reducer k v b -> [a] -> [b]
-- primero distribuye todos los documentos a procesar de la entrada
-- entre 100 maquinas, luego aplica el mapperProcess a cada subconjunto
-- de la entrada (simulando la ejecucion en multiples maquinas).
-- luego se combinan los resultados parciales del mapeo en cada maquina
-- y se realiza el proceso de reduccion

mapReduce mp rd xs = reduced $ combined $ mapped $ distributed $ xs
  where
    distributed = distributionProcess 100
    mapped = map $ mapperProcess mp
    combined = combinerProcess
    reduced = reducerProcess rd


-- Ejercicio 11

visitasPorMonumento :: [String] -> Dict String Int
-- cada vez que se procesa un monumento, se emite el par (nombre, 1)
-- luego la funcion de reduccion simplemente cuenta la cantidad de 1's
-- que aparecen como valor para una key determinada

visitasPorMonumento = mapReduce mp rd
  where
    mp s = [(s, 1)]
    rd (k, vs) = [(k, length vs)]


-- Ejercicio 12

monumentosTop :: [String] -> [String]
-- computa la cantidad de visitas por monumento (utilizando el ejercicio previo)
-- y luego realiza otra etapa de procesamiento mapReduce donde se ordenan
```

```
-- las tuplas (monumento, cantidad de visitas)

monumentosTop xs = mapReduce mp rd $ visitasPorMonumento $ xs
  where
    mp (s, i) = [(1, (s, i))]
    rd (k, vs) = map fst $ sortBy (\a b -> compare (snd b) (snd a)) $ vs


-- Ejercicio 13

monumentosPorPais :: [(Structure, Dict String String)] -> [(String, Int)]
-- la funcion de mapeo solo emite key-value's en el caso de que
-- el documento procesado sea un Monumento, ignorando los demas features.
-- para cada monumento, se emite el key-value (pais, 1), permitiendo que en
-- la etapa de reduccion se cuente la cantidad de veces que aparece cada pais
-- simplemente contando la cantidad de 1's (similar al ejercicio 11)

monumentosPorPais = mapReduce mp rd
  where
    mp (Monument, dt) = [(dt ! "country", 1)]
    mp (_, dt) = []
    rd (k, vs) = [(k, length vs)]
```

1.2. Tests

```
--  Para correr los tests deben cargar en hugs el módulo Tests
--  y evaluar la expresión "main".
-- Algunas funciones que pueden utilizar para chequear resultados:
-- http://hackage.haskell.org/package/hspec-expectations-0.6.1/docs/Test-Hspec-Expectations.html

import Test.Hspec
import Data.List
import MapReduce

-- ----------------------- Ejemplo de datos para el ejercicio 13 ----------------------

items1 :: [(Structure, Dict String String)]
items1 = [
    (Monument, [
      ("name","Obelisco"),
      ("latlong","-36.6033,-57.3817"),
      ("country", "Argentina")]),
    (Street, [
      ("name","Int. Güiraldes"),
      ("latlong","-34.5454,-58.4386"),
      ("country", "Argentina")]),
    (Monument, [
      ("name", "San Martín"),
      ("country", "Argentina"),
      ("latlong", "-34.6033,-58.3817")]),
    (City, [
      ("name", "Paris"),
      ("country", "Francia"),
      ("latlong", "-24.6033,-18.3817")]),
    (Monument, [
      ("name", "Bagdad Bridge"),
      ("country", "Irak"),
      ("new_field", "new"),
      ("latlong", "-11.6033,-12.3817")])
    ]

items2 :: [(Structure, Dict String String)]
items2 = [
    ]

items3 :: [(Structure, Dict String String)]
items3 = [
    (Monument, [
      ("name","Obelisco"),
      ("latlong","-36.6033,-57.3817"),
      ("country", "Argentina")]),
    (Monument, [
      ("name","Obelisco"),
      ("latlong","-36.6033,-57.3817"),
      ("country", "Brasil")]),
    (Monument, [
      ("name","Obelisco"),
      ("latlong","-36.6033,-57.3817"),
      ("country", "Yugoslavia")]),
    (Monument, [
      ("name","Obelisco"),
```

```
      ("latlong","-36.6033,-57.3817"),
      ("country", "Argentina")]),
    (Street, [
      ("name","Int. Güiraldes"),
      ("latlong","-34.5454,-58.4386"),
      ("country", "Argentina")]),
    (Monument, [
      ("name", "San Martín"),
      ("country", "Argentina"),
      ("latlong", "-34.6033,-58.3817")]),
    (City, [
      ("name", "Paris"),
      ("country", "Francia"),
      ("latlong", "-24.6033,-18.3817")]),
    (City, [
      ("name", "Paris"),
      ("country", "Francia"),
      ("latlong", "-24.6033,-18.3817")]),
    (Monument, [
      ("name", "Bagdad Bridge"),
      ("country", "Yugoslavia"),
      ("new_field", "new"),
      ("latlong", "-11.6033,-12.3817")])
    ]

-- ----------------------- Tests ----------------------

main :: IO ()
main = hspec $ do
  describe "Utilizando Diccionarios" $ do
    it "belongs, (?)" $ do
      belongs 3 [(3, "A"), (0, "R"), (7, "G")]
        `shouldBe` True
      belongs "k" []
        `shouldBe` False
      [("H", [1]), ("E", [2]), ("Y", [0])] ? "R"
        `shouldBe` False
      [("V", [1]), ("O", [2]), ("S", [0])] ? "V"
        `shouldBe` True
      [("calle",[3]),("city",[2,1])] ? "city"
        `shouldBe` True

    it "get, (!)" $ do
      get 3 [(3, "A"), (0, "R"), (7, "G")]
        `shouldBe` "A"
      [("V", [1]), ("O", [2]), ("S", [0])] ! "V"
        `shouldBe` [1]
      [("calle",[3]),("city",[2,1])] ! "city"
        `shouldBe` [2,1]

    it "insertWith" $ do
      insertWith (++) 1 [99] [(1 , [1]) , (2 , [2])]
        `shouldMatchList` [(1 ,[1 ,99]) ,(2 ,[2])]
      insertWith (++) 3 [99] [(1 , [1]) , (2 , [2])]
        `shouldMatchList` [(1 ,[1]) ,(2 ,[2]) ,(3 ,[99])]
      insertWith (++) 2 ’p’] (insertWith (++) 1 [’a’,’b’] (insertWith (++) 1 [’l’] []))
        `shouldMatchList` [(1,"lab"),(2,"p")]
```

```
  it "groupByKey" $ do
    groupByKey [("calle","Jean Jaures"),("ciudad","Brujas"),("ciudad","Kyoto"),("calle","7")]
      `shouldMatchList` [("calle",["Jean Jaures","7"]),("ciudad",["Brujas","Kyoto"])]
    groupByKey [("10",4),("33",756),("10",32),("95",76),("33",-68),("10",777)]
      `shouldMatchList` [("10",[4,32,777]), ("33",[756,-68]), ("95",[76])]
    groupByKey (groupByKey (groupByKey [("10",4), ("10",45)]))
      `shouldMatchList` [("10",[[[4,45]]])]

  it "unionWith" $ do
    unionWith (++) [("calle",[3]),("city",[2,1])] [("calle", [4]), ("altura", [1,3,2])]
      `shouldMatchList` [("calle",[3,4]),("city",[2,1]),("altura",[1,3,2])]
    unionWith (+) [("calle",23),("city",654)] [("calle", -23), ("altura", 435), ("city", -1)]
      `shouldMatchList` [("calle",0), ("city",653), ("altura",435)]
    unionWith union [("calle",[0]), ("city",[653]), ("altura",[435])]
                        [("calle",[3,4]),("city",[2,1]),("altura",[1,3,2])]
      `shouldMatchList` [("calle",[0,3,4]), ("city",[653,2,1]), ("altura",[435,1,3,2])]

describe "Utilizando Map Reduce" $ do
  it "distributionProcess" $ do
    distributionProcess 5 [1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12]
      `shouldBe` [[1 ,6 ,11] ,[2 ,7 ,12] ,[3 ,8] ,[4 ,9] ,[5 ,10]]
    distributionProcess 2 [1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12]
      `shouldBe` [[1,3,5,7,9,11],[2,4,6,8,10,12]]
    distributionProcess 97 [1..150]
      `shouldBe` [[1,98],[2,99],[3,100],[4,101],[5,102],[6,103],[7,104],
        [8,105],[9,106],[10,107],[11,108],[12,109],[13,110],[14,111],[15,112],
        [16,113],[17,114],[18,115],[19,116],[20,117],[21,118],[22,119],[23,120],
        [24,121],[25,122],[26,123],[27,124],[28,125],[29,126],[30,127],[31,128],
        [32,129],[33,130],[34,131],[35,132],[36,133],[37,134],[38,135],[39,136],
        [40,137],[41,138],[42,139],[43,140],[44,141],[45,142],[46,143],[47,144],
        [48,145],[49,146],[50,147],[51,148],[52,149],[53,150],[54],[55],[56],[57],
        [58],[59],[60],[61],[62],[63],[64],[65],[66],[67],[68],[69],[70],[71],[72],
        [73],[74],[75],[76],[77],[78],[79],[80],[81],[82],[83],[84],[85],[86],[87],
        [88],[89],[90],[91],[92],[93],[94],[95],[96],[97]]

  it "mapperProcess" $ do
    mapperProcess (\xs -> [(x, 1) | x <- xs]) [[1 ,6 ,11] ,[6 ,7 ,12] ,[12, 8] ,[11 ,9] ,[1 ,10]]
      `shouldBe` [(1,[1,1]),(6,[1,1]),(11,[1,1]),(7,[1]),(12,[1,1]),(8,[1]),(9,[1]),(10,[1])]
    mapperProcess (\xs -> [(product xs, sum xs)])
                    [[1 ,6 ,11] ,[6 ,7 ,12] ,[12, 8] ,[11 ,9] ,[66, 1]]
      `shouldBe` [(66,[18,67]),(504,[25]),(96,[20]),(99,[20])]

  it "combinerProcess" $ do
    combinerProcess [[(1,[1,1]),(66,[1,1]),(11,[1,1]),(7,[1]),(99,[1,1]),(8,[1]),
                      (9,[1]),(10,[1])], [(66,[18,67]), (504,[25]),(96,[20]),(99,[20])]]
      `shouldBe` [(1,[1,1]),(66,[1,1,18,67]),(11,[1,1]),(7,[1]),(99,[1,1,20]),
                  (8,[1]),(9,[1]),(10,[1]),(504,[25]),(96,[20])]

  it "reducerProcess" $ do
    reducerProcess (\(k, vs) -> k:vs)
                              [(1,[1,1]),(66,[1,1,18,67]),(11,[1,1]),(7,[1]),(99,[1,1,20]),
                                (8,[1]),(9,[1]),(10,[1]),(504,[25]),(96,[20])]
      `shouldBe` [1,1,1,66,1,1,18,67,11,1,1,7,1,99,1,1,20,8,1,9,1,10,1,504,25,96,20]

  it "visitasPorMonumento" $ do
    visitasPorMonumento [ "m1" ,"m2" ,"m3" ,"m2","m1", "m3", "m3"]
      `shouldMatchList` [("m3",3), ("m1",2), ("m2",2)]
    visitasPorMonumento [ "m1" ,"m2" ,"m3" ,"m2","m1", "m3", "m3",
```

```
                        "m1" ,"m2" ,"m3" ,"m2","m1", "m3", "m3"]
    `shouldMatchList` [("m3",6), ("m1",4), ("m2",4)]
  visitasPorMonumento ([ "m2" ,"m2" ,"m3" ,"m2","m1", "m3", "m2"] ++ ["m100" | i <- [1..100]])
    `shouldMatchList` [("m2",4), ("m100",100), ("m3",2), ("m1",1)]

it "monumentosTop" $ do
  monumentosTop [ "m1", "m0", "m0", "m0", "m2", "m2", "m3"]
    `shouldSatisfy` (\res -> res == ["m0", "m2", "m3", "m1"]
                      || res == ["m0", "m2", "m1", "m3"])
  monumentosTop ([ "m2" ,"m2" ,"m3" ,"m2","m1", "m3", "m2"] ++ ["m100" | i <- [1..100]])
    `shouldBe` ["m100","m2","m3","m1"]

it "monumentosPorPais" $ do
  monumentosPorPais items1
    `shouldMatchList` [("Argentina", 2), ("Irak", 1)]
  monumentosPorPais items2
    `shouldMatchList` []
  monumentosPorPais items3
    `shouldMatchList` [("Argentina",3), ("Brasil",1), ("Yugoslavia",2)]
```