Trabajo Práctico # 2

Programación Funcional, Universidad Nacional de Quilmes

19 de marzo de 2018

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluación principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. High Order Functions

1. *a*) `apply :: (a -> b) -> a -> b`
   En Haskell se llama `($)`
   b) `twice :: (a -> a) -> a -> a`
   c) `flip :: (a -> b -> c) -> b -> a -> c`
   d) `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
   e) `curry :: ((a,b) -> c) -> a -> b -> c`
   f) `uncurry :: (a -> b -> c) -> (a,b) -> c`
   g) `map :: (a -> b) -> [a] -> [b]`
   h) `filter :: (a -> Bool) -> [a] -> [a]`
   i) `any, all :: (a -> Bool) -> [a] -> Bool`
   j) `maybe :: b -> (a -> b) -> Maybe a -> b`
   k) `either :: (a -> c) -> (b -> c) -> Either a b -> c`
   l) `find :: (a -> Bool) -> [a] -> Maybe a`
   m) `partition :: (a -> Bool) -> [a] -> ([a], [a])`
   n) `nubBy :: (a -> a -> Bool) -> [a] -> [a]`
   ñ) `deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]`
   o) `groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`
   p) `concatMap :: (a -> [b]) -> [a] -> [b]`
   q) `until :: (a -> Bool) -> (a -> a) -> a -> a`
   r) `takeWhile :: (a -> Bool) -> [a] -> [a]`
   s) `dropWhile :: (a -> Bool) -> [a] -> [a]`
   t) `span, break :: (a -> Bool) -> [a] -> ([a],[a])`
   u) `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
   v) `zipApply :: [(a -> b)] -> [a] -> [b]`

w) `index :: [a] -> [(Int,a)]`

x) `applyN :: Int -> (a -> a) -> a -> a`

y) `iterate :: (a -> a) -> a -> [a]`
`iterate f x == [f x, f (f x), f (f (f x)), ...]`

z) `findIndex :: (a -> Bool) -> [a] -> Maybe Int`

2. Inidicar el resultado de las siguientes expresiones:

a) `id id`

b) `id id x`

c) `(*2) . (+2) $ 0`

d) `flip (-) 2 3`

e) `all (==True) $ map (const True) [1..5]`

f) `map (map (+1)) [[1..5], [6..10]]`

g) `map ((*2) . (+1)) [1..5]`

h) `iterate (+1) 0`

i) `maybe 0 (const 1) $ Just 1`

j) `until ((==2) . length) (map (+1)) [[1], [2], [3], [4,5]]`

2. Currificación

1. Mejorar la definición de todas las funciones de la práctica 1, haciendo uso de funciones de alto orden. Escribir las funciones estilo `pointfree` siempre que sea posible.

2. Indicar la cantidad de parámetros que recibe cada función definida hasta el momento.

3. Reducción

1. Demostrar las siguientes equivalencias

a) `map (+1) [1,2,3] == [2] ++ [3] ++ [4]`

b) `twice id 5 == (id . id . id) 5`

c) `maybe 0 (const 2) (Just Nothing) == head (map (+1) [1,2,3])`

d) `factorial 3 == product [1,2,3]`

e) `(iterate (1:) []) !! 3 == replicate 3 1`

f) `takeWhile (<3) [1,2,3] == map (+1) [0,1]`

g) `(curry . uncurry) (+) 1 2 == sum (filter (>=1)) [0,0,1,2]`

2. Indique la cantidad de reducciones que realizan las siguientes expresiones hasta llegar a una forma normal:

a) id

b) apply f

c) head (map (+1) [1..])

d) take (factorial 100) []

e) take (factorial 100) [x]

f) take (iterate (+1) 0) [1,2,3]

g) maybe (const False (take 100 [1..])) (+1) Nothing

4. Orden de evaluación

1. ¿Es posible implementar la estructura de control `if` en cualquier lenguaje de programación?

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True  thenBranch elseBranch = thenBranch
ifThenElse False thenBranch elseBranch = elseBranch
```

2. Nombre tres ventajas de lazy evaluation
3. Escriba 3 definiciones que hagan uso de lazy evaluation

5. Estructuras de Datos

Completar las siguientes definiciones:

```
type Set a = a -> Bool

belongs :: a -> Set a -> Bool
singleton :: Eq a => a -> Set a
empty :: Eq a => Set a
universal :: Eq a => Set a
evens :: Set Int
odds :: Set Int
greatherThan :: Int -> Set Int
hasElem :: Eq a => a -> Set [a]
complement :: Set a -> Set a
union :: Set a -> Set a -> Set a
intersection :: Set a -> Set a -> Set a
listToSet :: Eq a => [a] -> Set a
image :: Set (a,b) -> a -> Set b
```

6. Más funciones

1. `(&) :: a -> (a -> b) -> b`
2. `fix :: (a -> a) -> a`
3. `on :: (b -> b -> c) -> (a -> b) -> a -> a -> c`