Trabajo Práctico # 2

Programación Funcional, Universidad Nacional de Quilmes

18 de marzo de 2019

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

Resolver las siguientes funciones (no hace falta recursión explícita, definiciones y ejemplos encontrados en los módulos `Prelude` y `Data.List`, buscar en `Hoogle`):

1.
   - `apply :: (a -> b) -> a -> b`
     En Haskell se llama `($)`
   - `twice :: (a -> a) -> a -> a`
   - `flip :: (a -> b -> c) -> b -> a -> c`
   - `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
   - `curry :: ((a,b) -> c) -> a -> b -> c`
   - `uncurry :: (a -> b -> c) -> (a,b) -> c`
   - `map :: (a -> b) -> [a] -> [b]`
   - `filter :: (a -> Bool) -> [a] -> [a]`
   - `any, all :: (a -> Bool) -> [a] -> Bool`
   - `maybe :: b -> (a -> b) -> Maybe a -> b`
   - `either :: (a -> c) -> (b -> c) -> Either a b -> c`
   - `find :: (a -> Bool) -> [a] -> Maybe a`
   - `countBy :: (a -> Bool) -> [a] -> Int`
     Devuelve la cantidad de elementos que cumplen cierto predicado.
   - `partition :: (a -> Bool) -> [a] -> ([a], [a])`
   - `nubBy :: (a -> a -> Bool) -> [a] -> [a]`
   - `deleteBy :: (a -> a -> Bool) -> a -> [a] -> [a]`
   - `groupBy :: (a -> a -> Bool) -> [a] -> [[a]]`
   - `concatMap :: (a -> [b]) -> [a] -> [b]`
   - `until :: (a -> Bool) -> (a -> a) -> a -> a`
   - `takeWhile :: (a -> Bool) -> [a] -> [a]`

- `dropWhile :: (a -> Bool) -> [a] -> [a]`
- `span, break :: (a -> Bool) -> [a] -> ([a],[a])`
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
- `zipApply :: [(a -> b)] -> [a] -> [b]`
- `applyN :: Int -> (a -> a) -> a -> a`
- `iterate :: (a -> a) -> a -> [a]`
  `iterate f x == [x, f x, f (f x), f (f (f x)), ...]`
- `findIndex :: (a -> Bool) -> [a] -> Maybe Int`
- `findIndices :: (a -> Bool) -> [a] -> [Int]`
- `sortOn :: Ord b => (a -> b) -> [a] -> [a]`
- `unionBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`
- `intersectBy :: (a -> a -> Bool) -> [a] -> [a] -> [a]`

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. Eso implica que no hace falta recursión explícita.
2. Indicar la cantidad de parámetros que recibe cada función definida hasta el momento.

3. 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

4. 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
```

5. Más funciones

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