Práctica de ejercicios # 2 - Listas y Recursión

Estructuras de Datos, Universidad Nacional de Quilmes

13 de septiembre de 2022

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 videos publicados o clases presenciales****, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

1. Recursión sobre listas

Defina las siguientes funciones utilizando *recursión estructural* sobre listas, salvo que se indique lo contrario:

1. `sumatoria :: [Int] -> Int`
   Dada una lista de enteros devuelve la suma de todos sus elementos.
2. `longitud :: [a] -> Int`
   Dada una lista de elementos de algún tipo devuelve el largo de esa lista, es decir, la cantidad de elementos que posee.
3. `sucesores :: [Int] -> [Int]`
   Dada una lista de enteros, devuelve la lista de los sucesores de cada entero.
4. `conjuncion :: [Bool] -> Bool`
   Dada una lista de booleanos devuelve `True` si todos sus elementos son `True`.
5. `disyuncion :: [Bool] -> Bool`
   Dada una lista de booleanos devuelve `True` si alguno de sus elementos es `True`.
6. `aplanar :: [[a]] -> [a]`
   Dada una lista de listas, devuelve una única lista con todos sus elementos.
7. `pertenece :: Eq a => a -> [a] -> Bool`
   Dados un elemento `e` y una lista `xs` devuelve `True` si existe un elemento en `xs` que sea igual a `e`.
8. `apariciones :: Eq a => a -> [a] -> Int`
   Dados un elemento `e` y una lista `xs` cuenta la cantidad de apariciones de `e` en `xs`.
9. `losMenoresA :: Int -> [Int] -> [Int]`
   Dados un número `n` y una lista `xs`, devuelve todos los elementos de `xs` que son menores a `n`.

10. `lasDeLongitudMayorA :: Int -> [[a]] -> [[a]]`
    Dados un número `n` y una lista de listas, devuelve la lista de aquellas listas que tienen más de `n` elementos.

11. `agregarAlFinal :: [a] -> a -> [a]`
    Dados una lista y un elemento, devuelve una lista con ese elemento agregado al final de la lista.

12. `agregar :: [a] -> [a] -> [a]`
    Dadas dos listas devuelve la lista con todos los elementos de la primera lista y todos los elementos de la segunda a continuación. Definida en Haskell como `(++)`.

13. `reversa :: [a] -> [a]`
    Dada una lista devuelve la lista con los mismos elementos de atrás para adelante. Definida en Haskell como `reverse`.

14. `zipMaximos :: [Int] -> [Int] -> [Int]`
    Dadas dos listas de enteros, devuelve una lista donde el elemento en la posición `n` es el máximo entre el elemento `n` de la primera lista y de la segunda lista, teniendo en cuenta que las listas no necesariamente tienen la misma longitud.

15. `elMinimo :: Ord a => [a] -> a`
    Dada una lista devuelve el mínimo

2. Recursión sobre números

Defina las siguientes funciones utilizando *recursión* sobre números enteros, salvo que se indique lo contrario:

1. `factorial :: Int -> Int`
   Dado un número `n` se devuelve la multiplicación de este número y todos sus anteriores hasta llegar a 0. Si `n` es 0 devuelve 1. La función es parcial si `n` es negativo.

2. `cuentaRegresiva :: Int -> [Int]`
   Dado un número `n` devuelve una lista cuyos elementos sean los números comprendidos entre `n` y 1 (incluidos). Si el número es inferior a 1, devuelve la lista vacía.

3. `repetir :: Int -> a -> [a]`
   Dado un número `n` y un elemento `e` devuelve una lista en la que el elemento `e` repite `n` veces.

4. `losPrimeros :: Int -> [a] -> [a]`
   Dados un número `n` y una lista `xs`, devuelve una lista con los `n` primeros elementos de `xs`. Si la lista es vacía, devuelve una lista vacía.

5. `sinLosPrimeros :: Int -> [a] -> [a]`
   Dados un número `n` y una lista `xs`, devuelve una lista sin los primeros `n` elementos de lista recibida. Si `n` es cero, devuelve la lista completa.

3. Registros

1. Definir el tipo de dato Persona, como un nombre y la edad de la persona. Realizar las siguientes funciones:

   - `mayoresA :: Int -> [Persona] -> [Persona]`
     Dados una edad y una lista de personas devuelve a las personas mayores a esa edad.
   - `promedioEdad :: [Persona] -> Int`
     Dada una lista de personas devuelve el promedio de edad entre esas personas. *Precondición*: la lista al menos posee una persona.
   - `elMasViejo :: [Persona] -> Persona`
     Dada una lista de personas devuelve la persona más vieja de la lista. *Precondición*: la lista al menos posee una persona.

2. Modificaremos la representación de `Entreador` y `Pokemon` de la práctica anterior de la siguiente manera:

   ```
   data TipoDePokemon = Agua | Fuego | Planta
   data Pokemon = ConsPokemon TipoDePokemon Int
   data Entrenador = ConsEntrenador String [Pokemon]
   ```

   Como puede observarse, ahora los entrenadores tienen una cantidad de `Pokemon` arbitraria. Definir en base a esa representación las siguientes funciones:

   - `cantPokemon :: Entrenador -> Int`
     Devuelve la cantidad de Pokémon que posee el entrenador.
   - `cantPokemonDe :: TipoDePokemon -> Entrenador -> Int`
     Devuelve la cantidad de Pokémon de determinado tipo que posee el entrenador.
   - `cuantosDeTipo_De_LeGananATodosLosDe_ :: TipoDePokemon -> Entrenador -> Entrenador -> Int`
     Dados dos entrenadores, indica la cantidad de `Pokemon` de cierto tipo, que le ganarían a los `Pokemon` del segundo entrenador.
   - `esMaestroPokemon :: Entrenador -> Bool`
     Dado un entrenador, devuelve `True` si posee al menos un Pokémon de cada tipo posible.

3. El tipo de dato `Rol` representa los roles (desarollo o management) de empleados IT dentro de una empresa de software, junto al proyecto en el que se encuentran. Así, una empresa es una lista de personas con diferente rol. La definición es la siguiente:

   ```
   data Seniority = Junior | SemiSenior | Senior
   data Proyecto  = ConsProyecto String
   data Rol = Developer Seniority Proyecto | Management Seniority Proyecto
   data Empresa = ConsEmpresa [Rol]
   ```

   Definir las siguientes funciones sobre el tipo `Empresa`:

   - `proyectos :: Empresa -> [Proyecto]`
     Dada una empresa denota la lista de proyectos en los que trabaja, sin elementos repetidos.
   - `losDevSenior :: Empresa -> [Proyecto] -> Int`
     Dada una empresa indica la cantidad de desarrolladores senior que posee, que pertecen además a los proyectos dados por parámetro.
   - `cantQueTrabajanEn :: [Proyecto] -> Empresa -> Int`
     Indica la cantidad de empleados que trabajan en alguno de los proyectos dados.
   - `asignadosPorProyecto :: Empresa -> [(Proyecto, Int)]`
     Devuelve una lista de pares que representa a los proyectos (sin repetir) junto con su cantidad de personas involucradas.