Trabajo Práctico # 1

Estructuras de Datos, Universidad Nacional de Quilmes

25 de agosto de 2014

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 evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*
- *No dude en manifestar observaciones y críticas sobre los ejercicios de esta práctica, que con gusto serán recibidas por los docentes.*
- *Los ejercicios del anexo pueden obviarse, pero recuerde que aportan una compresión más profunda sobre los temás que aborda esta práctica. Considere resolverlos si se encuentra practicando para una instancia de evaluación y ya resolvió todos los anteriores.*

1. Conceptos básicos

1. Defina las siguientes funciones:

   a) `sucesor :: Int -> Int`
   Dado un número devuelve su sucesor

   b) `sumar :: Int -> Int -> Int`
   Dados dos números devuelve su suma utilizando la operación +.

   c) `maximo :: Int -> Int -> Int`
   Dados dos números devuelve el mayor de estos.

2. Defina las siguientes funciones utilizando pattern matching:

   a) `negar :: Bool -> Bool`
   Dado un booleano, si es $True$ devuelve $False$, y si es $False$ devuelve $True$.
   Definida en Haskell como *not*.

   b) `andLogico :: Bool -> Bool -> Bool`
   Dados dos booleanos si ambos son $True$ devuelve $True$, sino devuelve $False$.
   Definida en Haskell como &&.

   c) `orLogico :: Bool -> Bool -> Bool`
   Dados dos booleanos si alguno de ellos es $True$ devuelve $True$, sino devuelve $False$.
   Definida en Haskell como ||.

   d) `primera :: (Int,Int) -> Int`
   Dado un par de números devuelve la primera componente.
   Definida en Haskell como fst.

e) `segunda :: (Int,Int) -> Int`
Dado un par de números devuelve la segunda componente.
Definida en Haskell como *snd*.

f) `sumaPar :: (Int,Int) -> Int`
Dado un par de números devuelve su suma.

g) `maxDelPar :: (Int,Int) -> Int`
Dado un par de números devuelve el mayor de estos.

3. Defina las siguientes funciones polimórficas:

a) `loMismo :: a -> a`
Dado un elemento de algún tipo devuelve ese mismo elemento.

b) `siempreSiete :: a -> Int`
Dado un elemento de algún tipo devuelve el número 7.

c) `duplicar :: a -> (a,a)`
Dado un elemento de algún tipo devuelve un par con ese elemento en ambas componentes.

d) `singleton :: a -> [a]`
Dado un elemento de algún tipo devuelve una lista con este único elemento.

4. Defina las siguientes funciones polimórficas utilizando pattern matching sobre listas:

a) `isEmpty :: [a] -> Bool`
Dada una lista de elementos, si es vacía devuelve $True$, sino devuelve $False$.

b) `head' :: [a] -> a`
Dada una lista devuelve su primer elemento.

c) `tail' :: [a] -> [a]`
Dada una lista devuelve esa lista menos el primer elemento.

2. Recursión

2.1. 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. `promedio :: [Int] -> Int`
Dada una lista de enteros, devuelve un número que es el promedio entre todos los elementos de la lista. ¿Pudo resolverla utilizando recursión estructural?

4. `mapSucesor :: [Int] -> [Int]`
Dada una lista de enteros, devuelve la lista de los sucesores de cada entero.

5. `mapSumaPar :: [(Int,Int)] -> [Int]`
Dada una lista de pares de enteros, devuelve una nueva lista en la que cada elemento es la suma de los elementos de cada par.

6. `mapMaxDelPar :: [(Int,Int)] -> [Int]`
   Dada una lista de pares, devuelve una nueva lista en la que cada elemento es el mayor de las componentes de cada par.

7. `todoVerdad :: [Bool] -> Bool`
   Dada una lista de booleanos devuelve $True$ si todos sus elementos son $True$.

8. `algunaVerdad :: [Bool] -> Bool`
   Dada una lista de booleanos devuelve $True$ si alguno de sus elementos es $True$.

9. `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.

10. `apariciones :: Eq a => a -> [a] -> Int`
    Dados un elemento e y una lista xs cuenta la cantidad de apariciones de e en xs.

11. `filtrarMenoresA :: Int -> [Int] -> [Int]`
    Dados un número n y una lista xs, devuelve todos los elementos de xs que son menores a n.

12. `filtrarElemento :: Eq a => a -> [a] -> [a]`
    Dados un elemento y una lista filtra (elimina) todas las ocurrencias de ese elemento en la lista.

13. `mapLongitudes :: [[a]] -> [Int]`
    Dada una lista de listas, devuelve la lista de sus longitudes. Aplique esta función a la lista de strings `["Estructuras", "de", "datos"]` y observe el resultado.

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

15. `intercalar :: a -> [a] -> [a]`
    Dado un elemento e y una lista xs, ubica a e entre medio de todos los elementos de xs. Ejemplo:

    ```
    intercalar ',' "abcde" == "a,b,c,d,e"
    ```

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

17. `append :: [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 ++.

18. `aplanar :: [[a]] -> [a]`
    Dada una lista de listas, devuelve una única lista con todos sus elementos.

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

20. `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.

21. `zipSort :: [Int] -> [Int] -> [(Int, Int)]`
    Dadas dos listas de enteros de igual longitud, devuelve una lista de pares (min, max), donde min y max son el mínimo y el máximo entre los elementos de ambas listas en la misma posición.

2.2. 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. `contarHasta :: Int -> [Int]`
   Dado un número n devuelve una lista cuyos elementos sean los números entre 1 y n (incluidos).

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

5. `desdeHasta :: Int -> Int -> [Int]`
   Dados dos números n y m devuelve una lista cuyos elementos sean los números entre n y m (incluidos).

6. `takeN :: Int -> [a] -> [a]`
   Dados un número n y una lista xs, devuelve una lista con los primeros n elementos de xs. Si xs posee menos de n elementos, se devuelve la lista completa.

7. `dropN :: Int -> [a] -> [a]`
   Dados un número n y una lista xs, devuelve una lista sin los primeros n elementos de lista recibida. Si la lista posee menos de n elementos, se devuelve una lista vacía.

8. `splitN :: Int -> [a] -> ([a], [a])`
   Dados un número n y una lista xs, devuelve un par donde la primera componente es la lista que resulta de aplicar $takeN$ a xs, y la segunda componente el resultado de aplicar $dropN$ a xs. ¿Conviene utilizar recursión?

Anexo con ejercicios adicionales

Ejercicios adicionales para seguir practicando. Defina las siguientes funciones:

1. `particionPorSigno :: [Int] -> ([Int], [Int])`
   Dada una lista xs de enteros devuelva una tupla de listas, donde la primera componente contiene todos aquellos números positivos de xs y la segunda todos aquellos números negativos de xs. ¿Conviene utilizar recursión? Considere utilizar funciones auxiliares.

2. `particionPorParidad :: [Int] -> ([Int], [Int])`
   Dada una lista xs de enteros devuelva una tupla de listas, donde la primera componente contiene todos aquellos números pares de xs y la segunda todos aquellos números impares de xs. ¿Conviene utilizar recursión? Considere utilizar funciones auxiliares.

3. `subtails :: [a] -> [[a]]`
   Dada una lista devuelve cada sublista resultante de aplicar tail en cada paso. Ejemplo:

   ```
   subtails "abc" == ["abc", "bc", "c",""]
   ```

4. `agrupar :: Eq a => [a] -> [[a]]`
   Dada una lista xs devuelve una lista de listas donde cada sublista contiene elementos contiguos iguales de xs. Ejemplo:

   ```
   agrupar "AABCCC" = ["AA","B","CC"]
   ```

5. `esPrefijo :: Eq a => [a] -> [a] -> Bool`
   Devuelve $True$ si la primera lista es prefijo de la segunda.

6. `esSufijo :: Eq a => [a] -> [a] -> Bool`
   Devuelve $True$ si la primera lista es sufijo de la segunda.