Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 1

Funciones, tipos y alto orden

1. Objetivo

El objetivo de este proyecto es revisar la programación de funciones en Haskell, y comenzar a introducir algunos conceptos de polimorfismo y funciones de alto orden en los que profundizaremos en el siguiente proyecto. Se evaluará la definición de funciones recursivas usando caso base y caso inductivo (a través de análisis por casos, o pattern-matching), la definición de funciones por composición, y el uso de las funciones provistas por el lenguaje (en el Preludio), para la definición de funciones polimórficas.

En algunas de las funciones será necesario utilizar definiciones locales y también el uso de guardas para alternativas booleanas. En otros casos se deberá utilizar aplicación parcial de funciones y operadores binarios.

Algunas consideraciones que debes tener en cuenta:

- Hacé todo el proyecto en un mismo archivo.
- Usá `ghci`.
- Comentá el código, indicando a qué ejercicio corresponden las funciones (por ejemplo: `-- ejercicio n`)
- Nombrá las distintas versiones de una misma función utilizando ' (por ejemplo: `f`, `f''`, `f'''`, ...)
- Poner al principio del archivo {-# LANGUAGE NPlusKPatterns #-}
- **IMPORTANTE**: En cada ejercicio poné como comentario, al menos dos ejemplos de haber ejecutado la función en `ghci` y su resultado.

2. Ejercicios

1. Programá las siguientes funciones:

   a) `esCero :: Int -> Bool`, que verifica si un entero es igual a 0.

   b) `esPositivo :: Int -> Bool`, que verifica si un entero es estrictamente mayor a 0.

   c) `esVocal :: Char -> Bool`, que verifica si un caracter es una vocal en minúscula.

   d) `valorAbsoluto :: Int -> Int`, que devuelve el valor absoluto de un entero ingresado.

2. Programá las siguientes funciones usando recursión o composición:

   a) $todos.xs \doteq \langle \forall i \ : 0 \leq i < \#xs : \ xs.i \rangle$ (derivar en el teórico primero)

   `paratodo :: [Bool] -> Bool`, que verifica que *todos* los elementos de una lista sean `True`.

   b) `sumatoria :: [Int] -> Int`, que calcula la suma de todos los elementos de una lista de enteros.

c) `productoria :: [Int] -> Int`, que calcula el producto de todos los elementos de la lista de enteros.

d) `factorial :: Int -> Int`, que toma un número n y calcula $n!$.

e) Utilizá la función `sumatoria` para definir, `promedio :: [Int] -> Int`, que toma una lista de números no vacía y calcula el valor promedio (truncado, usando división entera).

A continuación mostramos algunos ejemplos del uso de las funciones en `ghci`:

```
$> paratodo [True, False, True]
False
$> paratodo [True, True]
True
$> sumatoria [1, 5, -4]
2
$> productoria [2, 4, 1]
8
```

3. Programá la función `pertenece :: Int -> [Int] -> Bool`, que verifica si un número se encuentra en una lista.

   Ejemplos de uso en `ghci`:

```
$> pertenece 4 [2,4,6]
True
$> pertenece 6 [2,4,6]
True
$> pertenece 7 [2,4,6]
False
```

4. Programá las siguientes funciones que implementan los cuantificadores generales. Notá que el segundo parámetro de cada función, es otra función!.

   a) $paratodo.xs.T \doteq \langle \forall i\ : 0 \leq i < \#xs :\ T.(xs.i) \rangle$ (derivar en el teórico primero)

   `paratodo' :: [a] -> (a -> Bool) -> Bool`, dada una lista xs de tipo `[a]` y un predicado `T :: a -> Bool`, determina si todos los elementos de xs satisfacen el predicado `T`.

   b) $existe.xs.T \doteq \langle \exists i\ : 0 \leq i < \#xs :\ T.(xs.i) \rangle$ (derivar en el teórico primero)

   `existe' :: [a] -> (a -> Bool) -> Bool`, dada una lista xs de tipo `[a]` y un predicado `T :: a -> Bool`, determina si algún elemento de xs satisface el predicado `T`.

   c) $sumatoria.xs.T = \langle \sum i\ : 0 \leq i < \#xs :\ T.(xs.i) \rangle$ (derivar en el teórico primero)

   `sumatoria' :: [a] -> (a -> Int) -> Int`, dada una lista xs de tipo `[a]` y una función `T :: a -> Int` (toma elementos de tipo `a` y devuelve enteros), calcula la suma de los valores que resultan de la aplicación de `T` a los elementos de xs.

   d) $productoria.xs.T = \langle \prod i\ : 0 \leq i < \#xs :\ T.(xs.i) \rangle$ (derivar en el teórico primero)

   `productoria' :: [a] -> (a -> Int) -> Int`, dada una lista de xs de tipo `[a]` y una función `T :: a -> Int`, calcula el producto de los valores que resultan de la aplicación de `T` a los elementos de xs.

   Ejemplos en `ghci`:

```
$> paratodo' [0,0,0,0] esCero
True
$> paratodo' [0,0,1,0] esCero
False
$> paratodo' "hola" esVocal
False
$> existe' [0,0,1,0] esCero
True
$> existe' "hola" esVocal
True
$> existe' "tnt" esVocal
False
```

5. Definí nuevamente la función `paratodo`, pero esta vez usando la función `paratodo'` (sin recursión ni análisis por casos!).

6. Utilizando las funciones del ejercicio 4, programá las siguientes funciones por composición, sin usar recursión ni análisis por casos.

   a) `todosPares :: [Int] -> Bool` verifica que todos los números de una lista sean pares.

   b) `hayMultiplo :: Int -> [Int] -> Bool` verifica si existe algún número dentro del segundo parámetro que sea múltiplo del primer parámetro.

   c) `sumaCuadrados :: Int -> Int`, dado un número no negativo n, calcula la suma de los primeros n cuadrados, es decir $\langle \sum i \ : 0 \leq i < n : \ i^2 \rangle$.

   Ayuda: *En Haskell se puede escribir la lista que contiene el rango de números entre* n *y* m *como* `[n..m]`.

   d) Programar la fución `existeDivisor::Int-> [Int] -> Bool`, que dado en entero `n` y una lista `ls` , devuelve `True` si y solo si, existe algún elemento en `ls` que divida a `n`.

   e) Utilizando la función del apartado anterior, definí la función `esPrimo:: Int -> Bool`, que dado un entero `n`, devuelve `True` si y solo si `n` es primo.

   Ayuda: *En Haskell se puede escribir la lista que contiene el rango de números entre* n *y* m *como* `[n..m]`.

   f) ¿Se te ocurre como redefinir `factorial` (ej. 2*d*) para evitar usar recursión?

   g) Programar la función `multiplicaPrimos :: [Int] -> Int` que calcula el producto de todos los números primos de una lista.

   h) Programar la función `esFib :: Int -> Bool`, que dado un entero `n`, devuelve `True` si y sólo si `n` está en la sucesión de Fibonacci.

   Ayuda: *Realizar una función auxiliar* `fib :: Int -> Int` *que dado un* `n` *devuelva el* `n`*-ésimo elemento de la sucesión.*

   i) Utilizando la función del apartado anterior, definí la función `todosFib :: [Int] -> Bool` que dada una lista `xs` de enteros, devuelva si todos los elementos de la lista pertenecen (o no) a la sucesión de Fibonacci.

7. Indagá en Hoogle sobre las funciones `map` y `filter`. También podes consultar su tipo en `ghci` con el comando `:t`.

   - ¿Qué hacen estas funciones?

- ¿A qué equivale la expresión `map succ [1, -4, 6, 2, -8]`, donde `succ n = n+1`?
- ¿Y la expresión `filter esPositivo [1, -4, 6, 2, -8]`?

8. Programá una función que dada una lista de números xs, devuelve la lista que resulta de duplicar cada valor de xs.

   a) Definila usando recursión.

   b) Definila utilizando la función `map`.

9. Programá una función que dada una lista de números xs, calcula una lista que tiene como elementos aquellos números de xs que son primos.

   a) Definila usando recursión.

   b) Definila utilizando la función `filter`.

   c) Revisá tu definición del ejercicio 6*g*. ¿Se puede mejorar?

10. La función `primIgualesA` toma un valor y una lista, y calcula el tramo inicial más largo de la lista cuyos elementos son iguales a ese valor. Por ejemplo:

```
primIgualesA 3 [3,3,4,1] = [3,3]
primIgualesA 3 [4,3,3,4,1] = []
primIgualesA 3 [] = []
primIgualesA 'a' "aaadaa" = "aaa"
```

   a) Programá `primIgualesA` por recursión.

   b) Programá nuevamente la función utilizando `takeWhile`.

11. La función `primIguales` toma una lista y devuelve el mayor tramo inicial de la lista cuyos elementos son todos iguales entre sí. Por ejemplo:

```
primIguales [3,3,4,1] = [3,3]
primIguales [4,3,3,4,1] = [4]
primIguales [] = []
primIguales "aaadaa" = "aaa"
```

   a) Programá `primIguales` por recursión.

   b) Usá cualquier versión de `primIgualesA` para programar `primIguales`. Está permitido dividir en casos, pero no usar recursión.

12. (*) Todas las funciones del ejercicio 4 son similares entre sí: cada una aplica la función término t a todos los elementos de una lista, y luego aplica algún operador entre todos ellos, obteniéndose así el resultado final. Para el caso de la lista vacía, se devuelve el elemento neutro. De esa manera cada una de ellas computa una cuantificación sobre los elementos de la lista transformados por t:

$$
\begin{aligned}
\texttt{paratodo'}.xs.t &= \langle \forall i \,: 0 \leq i < \#xs : \; t.xs!i \,\rangle \\
\texttt{existe'}.xs.t &= \langle \exists i \,: 0 \leq i < \#xs : \; t.xs!i \,\rangle \\
\texttt{sumatoria'}.xs.t &= \langle \Sigma i \,: 0 \leq i < \#xs : \; t.xs!i \,\rangle \\
\texttt{productoria'}.xs.t &= \langle \Pi i \,: 0 \leq i < \#xs : \; t.xs!i \,\rangle
\end{aligned}
$$

Por ejemplo, para `sumatoria'` el operador asociado al cuantificador Σ es la suma $(+)$, por lo que

```
sumatoria' [1,2,3] t = (t 1) + (t 2) + (t 3) + 0
```

donde el cálculo consistió en aplicar t a cada elemento, combinándolos con el operador $(+)$ hasta llegar a la lista vacía donde se devuelve el neutro de la suma (0). Guiándote por las observaciones anteriores, definí de manera recursiva la función $cuantGen$ (denota la cuantificación generalizada):

```
cuantGen :: (b -> b -> b) -> b -> [a] -> (a -> b) -> b
cuantGen op z xs t = ...
```

que tomando como argumento un operador `op`, su elemento neutro `z`, una lista de elementos `xs` y una función término `t`, aplica el operador a los elementos de la lista, transformados por la función término. En otras palabras, sea $\bigoplus$ un cuantificador cualquiera y $\oplus$ su operador asociado,

$$cuantGen.\oplus.z.xs.t = \langle \bigoplus i \ : 0 \leq i < \#xs : \ t.(xs\,!\,i) \rangle$$

Reescribir todas las funciones del punto 4 utilizando el cuantificador generalizado (sin usar inducción y en una línea por función).

13. (*) Definir una función que se denomina distancia de edición . Que toma como entrada dos strings (lista de caracteres). `distanciaEdicion::[Char]->[Char]-> Int`. La función `distanciaEdicion`, se comporta de la siguiente manera: - Si alguna de las listas es vacía, devuelve la longitud de la otra lista. - Si las dos listas son no vacías: `x:xs` e `y:ys`, compara los primeros elementos de cada lista:
    - Si `x==y`, no suma y sigue computando la distancia para `xs` e `ys` ,
    - Si `x!=y`, suma 1 y sigue computando la distancia para `xs` e `ys` .

14. (*) Definí una función primeros que cumplen, `primQueCumplen::[a]->(a->Bool)->[a]`, tal que, dada una lista `ls` y un predicado `p`, devuelve el tramo inicial de `ls` que cumple `p`.

15. (*) Para cada uno de los siguientes patrones, decidí si están bien tipados, y en tal caso dá los tipos de cada subexpresión. En caso de estar bien tipado, ¿el patrón cubre todos los casos de definición?

    a) `f :: (a, b) -> ...`
       `f (x , y) = ...`

    b) `f :: [(a, b)] -> ...`
       `f (a , b) = ...`

    c) `f :: [(a, b)] -> ...`
       `f (x:xs) = ...`

    d) `f :: [(a, b)] -> ...`
       `f ((x, y) : ((a, b) : xs)) = ...`

    e) `f :: [(Int, a)] -> ...`
       `f [(0, a)] = ...`

    f) `f :: [(Int, a)] -> ...`
       `f ((x, 1) : xs) = ...`

    g) `f :: (Int -> Int) -> Int -> ...`
       `f a b = ...`

    h) `f :: (Int -> Int) -> Int -> ...`
       `f a 3 = ...`

    i) `f :: (Int -> Int) -> Int -> ...`
       `f 0 1 2 = ...`

16. (*) Para las siguientes declaraciones de funciones, da al menos una definición cuando sea posible. ¿Podés dar alguna otra definición alternativa a la que diste en cada caso?

    Por ejemplo, si la declaración es `f :: (a, b) -> a`,
    la respuesta es: `f (x,y) = x`

    a) `f :: (a, b) -> b`

    b) `f :: (a, b) -> c`

    c) `f :: (a -> b) -> a -> b`

    d) `f :: (a -> b) -> [a] -> [b]`