

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 los siguientes proyectos. 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 Prelude), 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` con el flag `-Wall`. Se verificará que no haya *warnings* al cargar el archivo.
- 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'''`, ...)

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 carácter es una vocal en minúscula.

2. Programá las siguientes funciones usando recursión o composición:

- a) `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 una 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' :: [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' :: [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' :: [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' :: [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.
- `todosPares :: [Int] -> Bool` verifica que todos los números de una lista sean pares.
 - `hayMultiplo :: Int -> [Int] -> Bool` verifica si existe algún número dentro del segundo parámetro que sea múltiplo del primer parámetro.
 - `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]`.
- ¿Se te ocurre como redefinir `factorial` (ej. 2d) para evitar usar recursión?
 - `multiplicaPares :: [Int] -> Int` que calcula el producto de todos los números pares de una lista.
7. Indagá en [Hoogle](#) sobre las funciones `map` y `filter`. También podés 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`.
- Definila usando recursión.
 - 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 pares.
- Definila usando recursión.
 - Definila utilizando la función `filter`.
 - Revisá tu definición del ejercicio 6e. ¿Cómo podés mejorarla?
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"
```
- Programá `primIgualesA` por recursión.
  - 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. (\*) Para cada uno de los siguientes patrones, decidí si están bien tipados, y en tal caso da los tipos de cada subexpresión. En caso de estar bien tipado, ¿el patrón cubre todos los casos de definición?

- |                                                                                |                                                                                              |
|--------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| a) $f :: (a, b) \rightarrow \dots$<br>$f \ x = \dots$                          | g) $f :: [(Int, a)] \rightarrow \dots$<br>$f \ ((x, 1) : xs) = \dots$                        |
| b) $f :: (a, b) \rightarrow \dots$<br>$f \ (x, y) = \dots$                     | h) $f :: [(Int, a)] \rightarrow \dots$<br>$f \ ((1, x) : xs) = \dots$                        |
| c) $f :: [(a, b)] \rightarrow \dots$<br>$f \ (a, b) = \dots$                   | i) $f :: (Int \rightarrow Int) \rightarrow Int \rightarrow \dots$<br>$f \ a \ b = \dots$     |
| d) $f :: [(a, b)] \rightarrow \dots$<br>$f \ (x:xs) = \dots$                   | j) $f :: (Int \rightarrow Int) \rightarrow Int \rightarrow \dots$<br>$f \ a \ 3 = \dots$     |
| e) $f :: [(a, b)] \rightarrow \dots$<br>$f \ ((x, y) : ((a, b) : xs)) = \dots$ | k) $f :: (Int \rightarrow Int) \rightarrow Int \rightarrow \dots$<br>$f \ 0 \ 1 \ 2 = \dots$ |
| f) $f :: [(Int, a)] \rightarrow \dots$<br>$f \ [(0, a)] = \dots$               | l) $f :: a \rightarrow (a \rightarrow a) \rightarrow \dots$<br>$f \ a \ g = \dots$           |

13. (\*) Para las siguientes declaraciones de funciones, da al menos una definición cuando sea posible (que no sea la expresión `undefined`). ¿Podés dar alguna otra definición alternativa a la que diste en cada caso?

Por ejemplo, si la declaración es  $f :: (a, b) \rightarrow a$ ,  
la respuesta es:  $f \ (x, y) = x$

- |                                                         |                                                                                       |
|---------------------------------------------------------|---------------------------------------------------------------------------------------|
| a) $f :: (a, b) \rightarrow b$                          | e) $f :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$                           |
| b) $f :: (a, b) \rightarrow c$                          | f) $f :: (a \rightarrow b) \rightarrow a \rightarrow c$                               |
| c) $f :: a \rightarrow b$                               |                                                                                       |
| d) $f :: (a \rightarrow b) \rightarrow a \rightarrow b$ | g) $f :: (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow a \rightarrow c$ |