

Algoritmos y Estructuras de Datos I - Laboratorio

Proyecto 1

Funciones, tipos, clases 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 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` 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) `promedio :: [Int] -> Int`, que toma una lista de números 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 la función `encuentra` que dado un valor n de tipo `Int` y una lista de pares de tipo `[(Int, String)]` que asocia números a palabras, devuelve la palabra correspondiente a n , es decir, el segundo componente del par cuyo primer componente es igual a n . En caso que exista mas de una palabra asociada al mismo número, devuelve la primera de ellas. En caso que no exista palabra asociada al número, devuelve la palabra vacía `""`. Por ejemplo:

```
encuentra 10 [(40, "tos"), (10, "uno"), (16, "taza"), (10, "dos")] = "uno"
encuentra 102 [(40, "tos"), (103, "vela"), (16, "taza")] = ""
encuentra 102 [] = ""
```

5. 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
```

6. Definí nuevamente la función `paratodo`, pero esta vez usando la función `paratodo'` (sin recursión ni análisis por casos!).
7. Utilizando las funciones del ejercicio 5, 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) ¿Se te ocurre como redefinir `factorial` (ej. 2d) para evitar usar recursión?
 - e) `multiplicaPares :: [Int] -> Int` que calcula el producto de todos los números pares de una lista.
8. Indagá en Hoogle (no es un typo!) sobre las funciones `map` y `filter`. También puedes 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]`?

9. 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`.

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

- a) Definila usando recursión.
- b) Definila utilizando la función `filter`.
- c) Revisá tu definición del ejercicio 7e. ¿Cómo puedes mejorarla?

11. Considerá las siguientes funciones:

- `sumarALista :: Num a => a -> [a] -> [a]` que toma un número y una lista de números y le suma a cada elemento de la lista el primer parámetro. Por ejemplo:

`sumarALista 3 [4,6,7] = [7,9,10]`

- `encabezar :: a -> [[a]] -> [[a]]` que toma un valor de tipo `a` y lo introduce en la cabeza de cada lista del segundo parámetro. Por ejemplo:

`encabezar 3 [[2,1],[],[4,7]] = [[3,2,1],[3],[3,4,7]]`

- `mayoresA :: Ord a -> a -> [a] -> [a]` que toma un valor ordenable n y una lista de valores ordenables xs , y calcula una lista que contiene los elementos de xs que son mayores que n

`mayoresA 4 [1,2,3,4,5,6,7,8,9] = [5,6,7,8,9]`

- Programá las funciones usando recursión.
- Programá las funciones utilizando `map` y `filter` según corresponda.

12. ¿Se te ocurre cómo programar la función del ejercicio 4 utilizando composición y la función `filter`?

13. 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`.

14. 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"
```

- Programá `primIguales` por recursión.
- Usá cualquier versión de `primIgualesA` para programar `primIguales` sin recursión.

15. Considerá la función `minimo` que calcula cuál es el menor valor de una lista de tipo `[a]`.

- Definila sólo para listas no vacías.
- Definila para todos los casos, limitando el tipo `a` a la clase `Bounded` para poder definir el caso base.

Ayuda: Para probar esa función dentro de `ghci` con listas vacías, indicar el tipo concreto con tipos de la clase `Bounded`, por ejemplo: `([1,5,10] :: [Int])`, `([] :: [Bool])`, etc.

16. (*) 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?

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

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

17. (*) 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?

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