MEMORIA DE EJERCICIOS

Modelos Avanzados de Computación

Universidad de Huelva

Grado en Ingeniería Informática
Especialidad en Computación
Curso 2022/23
Manuel Ramírez Ballesteros

Índice:

1- Introducción

En esta memoria se justificarán las soluciones propuestas para los cinco ejercicios comunes además de plantear y resolver otros cinco nuevos ejercicios en Haskell únicamente con las funciones y contenidos vistos en las sesiones de prácticas. Los ejercicios a resolver son:

- cambia_el_primero(*a*,*b*): Cambia el primer valor de la lista *b* por el valor de *a*.
- cambia_el_n(a,n,b): Cambia el valor de la posición n de la lista *b* por el valor de *a*.
- get_mayor_abs(*a*): Devuelve el mayor número en valor absoluto de la lista *a*.
- num_veces(*a*,*b*): Devuelve la cantidad de veces que aparece el valor *a* en la lista *b*.
- palabras_mayores_n(*n*,*a*): Devuelve una lista con las palabras de *a* cuya longitud sea mayor que la de *n*.

Los ejercicios nuevos que se han planteado son:

- esPrimo(*a*): Devuelve true si el número *a* es primo y false en caso contrario.
- ordenada(*a*): Devuelve true si la lista *a* está ordenada y false si no.
- quicksort(*a*): Devuelve la lista *a* ordenada mediante el algoritmo Quicksort donde el pivote será el primer elemento de la lista y se dividirá recursivamente en dos partes: a la izquierda los elementos menores al pivote y en la derecha los mayores, hasta que la lista quede ordenada.
- ordena_insercion(*a*): Devuelve la lista *a* ordenada por inserción, es decir, colocando cada elemento directamente en su posición correcta.
- factoriza(*a*)*:* Devuelve la lista de factores primos que son divisores de *a.*

2- ***Ejercicio común:*** cambia_el_primero *a b*

La función cambia_el_primero se ha implementado con la idea de, si recibimos una lista *b* vacía, devolver una lista con *a* como único elemento y, si no lo está, eliminar el primer elemento de la lista y construir una nueva con *a* como primer elemento:

```
cambia_el_primero :: a -> [a] -> [a]
cambia_el_primero a [] = [a]
cambia_el_primero a b = a : drop 1 b
```

Podemos ver a continuación algunas pruebas del funcionamiento de la función:

```
Main> cambia_el_primero 3 [1,2,3,4,5]
[3,2,3,4,5]
Main> cambia_el_primero 0 [1,2,3,4,5]
[0,2,3,4,5]
Main> cambia_el_primero 0 []
[0]
```

3- ***Ejercicio común:*** cambia_el_n *a n b*

Para esta función se ha considerado que:

1) si *n* es 0, se devolverá *b.*

2) si *b* está vacía, se devolverá una lista con el elemento *a*.

3) si n es 1, llamaremos a la función planteada en el ejercicio anterior.

4) para el resto de casos, concatenaremos los *n* primeros elementos de *b* descartando el n-ésimo, el cual será sustituido por *a,* y con el resto de los elementos (si los hubiese).

```
cambia_el_n :: a -> Int -> [a] -> [a]
cambia_el_n a 0 b = b
cambia_el_n a n [] = [a]
cambia_el_n a 1 b = cambia_el_primero a b
cambia_el_n a n b = init (take (n) b) ++ [a] ++ drop n b
-- Descartes :
-- cambia_el_n a n b = init b ++ [c]
-- cambia_el_n a n (c : r) = (c : cambia_el_n a (n-1) r)
```

Así conseguimos solventar problemas de que la longitud de *b* sea menor que *n*. A continuación, podemos ver algunos ejemplos de ejecución:

```
Main> cambia_el_n 3 1 [1,2,3,4]
[3,2,3,4]
Main> cambia_el_n 3 0 [1,2,3,4]
[1,2,3,4]
Main> cambia_el_n 3 0 []
[3]
Main> cambia_el_n 3 8 [1,2,3,4]
[1,2,3,3]
```

4- ***Ejercicio común:*** get_mayor_abs *a*

Para esta función se ha considerado que, si la lista está vacía, se devuelva 0, si no escogeremos el mayor elemento de la lista resultante de aplicar el valor absoluto a todos los elementos de *a:*

```
get_mayor_abs :: (Ord a, Num a) => [a] -> a
get_mayor_abs [] = 0
get_mayor_abs a = maximum(map abs a)
```

Podemos comprobar el funcionamiento con los siguientes ejemplos:

```
Main> cambia_el_n 1 0 [1,2,3]
[1,2,3]
Main> get_mayor_abs [1,2,4,8,12,-243]
243
Main> get_mayor_abs [1,2,4,8,-12,243]
243
Main> get_mayor_abs [1,-2222,4,8,-12,-243]
2222
Main> get_mayor_abs []
0
```

5- ***Ejercicio común:*** num_veces *a b*

Para la siguiente función se ha considerado que, si la lista *b* está vacía, se devuelva 0, si no se devolverá la longitud del vector formado tras filtrar los valores iguales a *a* en *b:*

```
num_veces :: Eq a => a -> [a] -> Int
num_veces a [] = 0
num_veces a b = length (filter (==a) b)
--num_veces a (c : r) = if a == c then 1 + num_veces a r else num_veces a r
```

Algunos ejemplos de su ejecución son:

```
Main> num_veces 1 []
0
Main> :reload
Main> num_veces 1 [1,2,3]
1
Main> num_veces 0 [1,2,3]
0
Main> num_veces 1 [1,2,1,3,1,1]
4
```

6- *Ejercicio común:* palabras_mayores_n *n b*

En esta función se ha considerado que, si la lista *b* está vacía, devolveremos una lista vacía al no existir palabras mayores, si no devolveremos una lista filtrando los elementos de *b* cuya longitud sea mayor que *n*:

```
palabras_mayores_n :: Int -> [[a]] -> [[a]]
palabras_mayores_n n [] = []
palabras_mayores_n n b = filter (\x -> length x > n) b
```

Para comprobar su funcionamiento:

```
Main> palabras_mayores_n 3 ["aaaaaaa","aaa","aaaa","aa"]
["aaaaaaa","aaaa"]
Main> palabras_mayores_n 3 []
[]
Main> palabras_mayores_n 0 []
[]
Main> palabras_mayores_n 0 ["aaaaaaa","aaa","aaaa","aa"]
["aaaaaaa","aaa","aaaa","aa"]
```

7- *Ejercicio nuevo:* esPrimo *a*

Para esta función se han considerado los casos base con *a* = 0 y *a* = 1 que devolverán un valor false al no ser primos. Para *a* = 2, se devolverá true y, para el resto de casos, se comprobará si existe al menos un valor en el rango [2, *a*/2] cuyo resto al dividir *a* sea 0. En caso de existir, implicará que el número no es primo. En caso de no existir, devolvería false, por lo que la solución será la negación de este resultado:

```
esPrimo :: Integral a => a -> Bool
esPrimo 0 = False
esPrimo 1 = False
esPrimo 2 = True
esPrimo a = not (any (\y -> mod a y == 0) [2.. (div a 2)])
```

Unos ejemplos de la ejecución de dicha función serían:

```
Main> esPrimo 3
True
Main> esPrimo 4
False
Main> esPrimo 10
False
Main> esPrimo 1
False
Main> esPrimo 17
True
Main> esPrimo 101
True
```

8- ***Ejercicio nuevo:*** ordenada *a*

Para esta función, se han establecido como casos base que, si la lista *a* está vacía o contiene un solo elemento, ésta está ordenada, devolviendo true. Para el resto de casos donde *a* tiene 2 o más elementos, se devolverá el resultado lógico de evaluar si el primer elemento es menor o igual que el segundo unido al resultado de aplicar recursivamente la función a la lista sin el primer elemento:

```
ordenada :: Ord a => [a] -> Bool
ordenada [] = True
ordenada [a] = True
ordenada (x:y:xs) = (x <= y) && ordenada (y:xs)
```

Algunos ejemplos de ejecución serían:

```
Main> ordenada [1,2,3,4]
True
Main> ordenada [1,2,3,4,1]
False
Main> ordenada [1,-1,2,3,4]
False
Main> ordenada [-1,1,2,3,4]
True
Main> ordenada [-1]
True
Main> ordenada []
True
```

9- ***Ejercicio nuevo:*** quicksort *a*

Para esta función se ha considerado que, si *a* es una lista vacía o con un único elemento, se devolverá la misma lista al estar ordenada. Si la lista *a* tiene más de un

elemento, se procederá filtrando los elementos menores o iguales al pivote en el resto de la lista y ordenándolos recursivamente por quicksort, obteniendo así los menores ordenados, los cuales se concatenarán al valor del pivote que, a su vez se concatenará con los elementos mayores ordenados por un procedimiento de filtrado similar al descrito con los menores:

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort [x] = [x]
quicksort (c : r) = quicksort (filter (\y -> y <= c) r)
                    ++ [c]
                    ++ quicksort (filter (\y -> y > c) r)
```

Algunos ejemplos de la ejecución de la función son:

```
Main> quicksort [1,4,2,3,0]
[0,1,2,3,4]
Main> quicksort [1,0,4,2,12,3,0]
[0,0,1,2,3,4,12]
Main> quicksort [1]
[1]
Main> quicksort []
[]
Main> quicksort [3,1]
[1,3]
```

10- Ejercicio nuevo: ordena_insercion *a*

Para resolver este ejercicio se han planteado dos funciones:

- inserta_elemento *a b*: Devuelve una lista con el elemento *a* en su posición correcta dentro de *b*. El caso base se tiene cuando la lista *b* es vacía, donde devolvemos una lista con el elemento *a*. Cuando la lista *b* no está vacía, filtramos los elementos menores que *a* en *b* para saber la posición correcta en la que insertarlo mediante length. Tomamos esa cantidad de elementos contenidos en *b*, los concatenamos a *a* y, por último, concatenamos el resto de elementos en *b* mediante la función drop, ya que conocemos la cantidad de elementos que tenemos que obviar.
- ordena_insercion *a*: Devuelve la lista *a* ordenada por inserción utilizando la función descrita anteriormente. Para ello, devolveremos una lista vacía cuando la lista *a* sea vacía como caso base. Cuando la lista no sea vacía, insertaremos el elemento de la cabeza de la lista en su posición correcta dentro del resto de la lista ordenado recursivamente por inserción.

El código de las funciones es el siguiente:

```
ordena_insercion :: (Ord a) => [a] -> [a]
ordena_insercion [] = []
ordena_insercion (c : r) = inserta_elemento c (ordena_insercion r)
```

```
inserta_elemento :: Ord a => a -> [a] -> [a]
inserta_elemento a [] = [a]
inserta_elemento a b = take (length (filter (\y -> y <= a) b)) b
                       ++ [a]
                       ++ drop (length(filter (\y -> y <= a) b)) b
```

Algunos ejemplos de ejecución de ambas funciones son:

```
Main> ordena_insercion [11,1,43,3,2,1]
[1,1,2,3,11,43]
Main> ordena_insercion [1,2,3,4]
[1,2,3,4]
Main> ordena_insercion [1]
[1]
Main> ordena_insercion []
[]
```

```
Main> inserta_elemento 1 []
[1]
Main> inserta_elemento 1 [3]
[1,3]
Main> inserta_elemento 1 [0]
[0,1]
Main> inserta_elemento 1 [5,2,7,1]
[5,1,2,7,1]
Main> inserta_elemento 9 [5,2,7,1]
[5,2,7,1,9]
```

Como se ha visto en clase, es más elegante utilizar las funciones takeWhile y dropWhile ya que se simplifica tanto el código (no requiere del uso de length) como el número de operaciones que se ejecutan.

11- ***Ejercicio nuevo:*** factoriza *a*

Para resolver este ejercicio se han planteado dos funciones:

- primer_factor *a*: Devuelve el primer divisor primo de *a*. Se ha considerado como caso base cuando *a* = 1, donde devolvemos como resultado 1. Cuando *a* tiene cualquier otro valor, escogeremos el primer elemento que resulte al filtrar todos sus divisores, el cual siempre será un número primo (incluido *a* en caso de que sea primo).
- factoriza *a*: Devuelve una lista con todos los factores primos de a. Como caso base se ha establecido que para *a* = 1 se devolverá una lista vacía y no [1] (evitando así que aparezca siempre el 1, que no es primo, en la lista de factores de cualquier número). Para el resto de valores, devolveremos en la cabeza el primer factor de *a* (obtenido con la función descrita anteriormente) y el resto de valores serán el resultado de factorizar recursivamente los cocientes de la división entre *a* y dicho primer factor.

El código de las funciones es el siguiente:

```
primer_factor :: Integral a => a -> a
primer_factor 1 = 1
primer_factor a = head(filter(\y -> mod a y == 0) [2..a])
```

```
factoriza :: Integral a => a -> [a]
factoriza 1 = [] -- No he puesto [1] para que no salga como factor siempre
factoriza a = primer_factor a : factoriza (div a (primer_factor a))
```

Algunos ejemplos de ejecución de las funciones son:

```
Main> primer_factor 1
1
Main> primer_factor 2
2
Main> primer_factor 10
2
Main> primer_factor 101
101
Main> primer_factor 77
7
```

```
Main> factoriza 100
[2,2,5,5]
Main> factoriza 1
[]
Main> factoriza 2
[2]
Main> factoriza 10
[2,5]
Main> factoriza 101
[101]
```

12- Conclusión

Estos ejercicios han servido para familiarizarme con el uso de algunas de las funciones básicas de Haskell, que es un lenguaje nuevo para mí, aunque al estar basado en funciones y conceptos más matemáticos, me resulta más sencillo desenvolverme en él. Por ello, se ha tratado de demostrar la capacidad de resolver problemas no triviales con las funciones vistas en clase, lo que supone una dificultad añadida al no poder emplear todas las herramientas que nos ofrece este lenguaje. Aunque esto implique no poder optimizar del todo el código ni ofrecer un tratamiento de errores adecuado, puede apreciarse la sencillez del mismo a la hora de resolver la mayoría de los ejercicios propuestos.