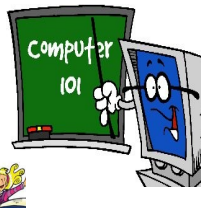

Guía de ejercitación nro. 9

Tema: Programación Funcional.

Temas específicos: Repaso de expresiones en Haskell. Listas. Funciones con listas. Funciones con recursividad.

A) Utilización de funciones predefinidas para listas:

1) Muestra la cantidad de elementos de la siguiente lista: [(-2),(-1),0,1,2].

```
Hugs> length [(-2),(-1),0,1,2]
5
```

2) Escribe 6 representaciones diferentes de los elementos de la lista del punto 1).

```
[(-2)..2]
(-2):[(-1),0,1,2]
(-2):((-1):[0,1,2])
(-2):((-1):(0:[1,2]))
(-2):((-1):(0:(1:[2])))
(-2):((-1):(0:(1:(2:[]))))
```

3) Verifica si el elemento (-1) se encuentra en la lista del punto 1).

```
Hugs> elem (-1) [(-2)..2]
True
```

4) Verifica si el elemento 5 se encuentra en la lista del punto 1).

```
Hugs> elem 5 [(-2)..2]
False
```

5) Genera una nueva lista con los cuadrados de los elementos de la lista del punto 1).

```
--esta función habría que tener implementada
cuadrado :: Int->Int
cuadrado x = x^2

Main> map cuadrado [(-2)..2]
[4,1,0,1,4]
```

6) Genera una nueva lista con los elementos de la lista del punto 1) y los elementos de la siguiente lista: [1,2,3,4,5].

```
Hugs> [(-2)..2]++[1,2,3,4,5]
[-2,-1,0,1,2,1,2,3,4,5]

Hugs> concat[[(-2)..2],[1..5]]
[-2,-1,0,1,2,1,2,3,4,5]

Hugs> :load List
List> union [(-2)..2] [1..5]
[-2,-1,0,1,2,3,4,5]
```

7) Obtiene el primer elemento de la lista del punto 1).

```
Hugs> head [(-2)..2]
-2
```

8) Obtiene una lista con todos los elementos de la lista del punto 1) pero sin incluir el primer elemento.

```
Hugs> tail [(-2)..2]
[-1,0,1,2]
```

9) Suma los elementos de la lista del punto 1).

```
Hugs> sum [(-2)..2]
0
```

10) A partir de la lista generada en el punto 1), obtiene el elemento con máximo valor relativo.

```
List> maximum [(-2)..2]
2
```

11) Concatena los elementos de las siguientes listas[1.5,1.6,2.1],[],[2.4,2.6].

```
Hugs> concat [[1.5,1.6,2.1],[],[2.4,2.6]]
[1.5,1.6,2.1,2.4,2.6]
```

12) Genera una lista con los 3 primeros elementos de la lista del punto 1).

```
Hugs> take 3 [(-2)..2]
[-2,-1,0]
```

13) Genera una lista que no incluya los 3 primeros elementos de la lista del punto 1).

```
Hugs> drop 3 [(-2)..2]
[1,2]
```

B) Utilización de funciones predefinidas para listas en List:

--Se deberá cargar el módulo List para poder utilizar las funciones propias de este módulo:

Hugs>:load List

14) Genera una nueva lista con los elementos de la lista del punto 1) insertando además el valor 3 como nuevo elemento.

List> insert 3 [(-2)..2]

[-2,-1,0,1,2,3]

15) Genera una nueva lista con los elementos de la lista del punto 1) y los elementos de la siguiente lista [1..5]. Realiza la unión de elementos de ambas listas.

List> union [(-2)..2] [1..5]

[-2,-1,0,1,2,3,4,5]

16) Ordena de menor a mayor los elementos de la siguiente lista: [3,1,5,3,4,3,4,5,-3,0,-1,2].

List> sort [3,1,5,3,4,3,4,5,-3,0,-1,2]

[-3,-1,0,1,2,3,3,3,4,4,5,5]

C) Definición de funciones recursivas:

17) Realiza una función recursiva que permita obtener la sumatoria básica de un número entero positivo. Suponer que solo se ingresarán valores positivos.

--Esta función la creamos en un archivo con la extensión hs, por ejemplo:

ppr2k10_2024_clase9.hs.

sumatoria 0= 0

sumatoria x = x + sumatoria(x-1)

--Luego cargamos el archivo ppr2k10_2024_clase9.hs y ejecutamos la función sumatoria.

Hugs> :load ppr2k10_2024_clase9.hs

Main> sumatoria 5

15

18) Realiza una función recursiva que permita obtener la sumatoria de un número entero positivo. En caso que se ingrese un número entero negativo, la función deberá devolver un -1.

--función sumatoria SÓLO para números positivos o 0. Si devuelve -1 significa que el número ingresado es no válido (cualquier nro. negativo sería no válido).

sumatoria1 0= 0

sumatoria1 x = if x>0 then x + sumatoria1(x-1) else -1

--Ejecutamos la función

Main> sumatoria1 5

15

Main> sumatoria1 (-5)

-1 *--Se interpreta como que el argumento recibido en la función no es válido.*

19) Realiza una función recursiva que permita obtener la sumatoria tanto de número entero positivo como negativo.

```
--función sumatoria tanto para números positivos como negativos
sumatoria2 0= 0
sumatoria2 x = if x>0 then x + sumatoria2(x-1) else x + sumatoria2(x+1)
--Ejecutamos la función
Main> sumatoria2 5
15
Main> sumatoria2 (-5)
-15
```

20) Realiza una función recursiva que permita obtener la sumatoria tanto de número entero positivo como negativo, pero utilizando guardas.

```
sumatoria3 x  | x == 0 = x
              | x > 0 = x + sumatoria3(x-1)
              | x < 0 = x + sumatoria3(x+1)
--Ejecutamos la función
Main> sumatoria3 5
15
Main> sumatoria3 (-5)
-15
```

21) Realiza una función que permita contar la cantidad de elementos de una lista.

```
--función que recibe como parámetro una lista de elementos y devuelve la cantidad de elementos de la lista
contar :: [Int] -> Int
contar [] = 0
contar (h:t)= 1 + contar(t)
    --Ejecutamos la función
    Main> contar [1,5,2]
    3
    Main> contar []
    0
```

D) Definición de funciones propias con listas:

22) Realiza una función que permita obtener el primer elemento de una lista.

```
--función que recibe como parámetro una lista de elementos y devuelve la cabeza de la lista
--(el primer elemento de la misma).
cabeza :: [Int] -> Int
```

cabeza (h:t) = h

23) Realiza una función que permita obtener el último elemento de una lista.

```
--función que recibe como parámetro una lista de elementos y devuelve el último elemento de la lista
ultimo :: [Int] -> Int
ultimo [x] = x
ultimo (h:t)= ultimo (t)

--esta otra versión, directamente utiliza como variable anónima a la que representa la cabeza de la lista,
--puesto que ésta no se utiliza en la implementación de la función en sí.
ultimo2 :: [Int] -> Int
ultimo2 [x] = x
ultimo2 (_:t)= ultimo2 (t)
```

24) Realiza una función que permita obtener el penúltimo elemento de una lista.

```
--función que recibe como parámetro una lista de elementos y devuelve el penúltimo elemento de la lista
penultimo :: [Int] -> Int
penultimo [y,x] = y
penultimo (h:t)= penultimo (t)

--esta otra versión, también utiliza como variable anónima a la que representa la cabeza de la lista,
--puesto que ésta no se utiliza en la implementación de la función en sí.
penultimo2 :: [Int] -> Int
penultimo2 [y,x] = y
penultimo2 (_:t)= penultimo2 (t)
```

25) Realiza una función que permita sumar todos los elementos de una lista. La lista debe ser de números.

```
--función que recibe como parámetro una lista y devuelve la suma de todos los elementos de la misma.
sumatoria_lista :: [Int] -> Int
sumatoria_lista [] = 0
sumatoria_lista (h:t) = h + sumatoria_lista(t)
```

26) Realiza una función que permita devolver True en caso que algún elemento sea un True. Considerar una lista de Bool.

```
--Función que recibe una lista de Bool y devuelve True en caso que haya al menos un valor True
--en la lista, y False en caso que todos los elementos de la lista fuesen False.
--Esta función ni bien determina si hay algún True, detiene el procesamiento de los elementos de la lista y
--devuelve True, utilizando if-then-else.
hay_algun_true :: [Bool] -> Bool
hay_algun_true [] = False
```

```
hay_algun_true (x:xs) = if x then True else hay_algun_true (xs)

--Esta función también ni bien determina si hay algún True, detiene el procesamiento de los
--elementos de la lista y devuelve True, pero utilizando guardas.
hay_algun_true2 :: [Bool] -> Bool
hay_algun_true2 [] = False
hay_algun_true2 (x:xs) | x = True
                       | otherwise = hay_algun_true2 (xs)

--Esta otra función tiene la contraparte que evaluará cada uno de los elementos de la lista hasta
--el último para determinar si hay algún True.
hay_algun_true3 :: [Bool] -> Bool
hay_algun_true3 [] = False
hay_algun_true3 (x:xs) = x || hay_algun_true3 xs
```

Ejercicios de tarea:

27) Realiza una función que permita devolver False en caso que algún elemento sea un False. Considerar una lista de Bool.

```
--Función que recibe una lista de Bool y devuelve False en caso que haya al menos un valor False
--en la lista, y True en caso contrario, es decir, en caso que no haya ningún False en la lista.
no_hay_ningun_false :: [Bool] -> Bool
no_hay_ningun_false [] = True
no_hay_ningun_false (x:xs) = if not (x) then False else no_hay_ningun_false (xs)
```

28) Realiza una función que permita devolver False en caso que algún elemento sea un True. Considerar una lista de Bool.

```
--Función que recibe una lista de Bool y devuelve False en caso que haya al menos un valor True en la lista,
-- y True en caso que ninguno de los elementos sea True. Ni bien detecta que hay algún True, corta la
-- búsqueda de los valores Bool de la lista, devolviendo un False. Utiliza la expresión if-then-else.
no_hay_ningun_true :: [Bool] -> Bool
no_hay_ningun_true [] = True
no_hay_ningun_true (x:xs) = if x then False else no_hay_ningun_true (xs)

--Esta función, también, ni bien detecta que hay algún True, corta la búsqueda de los valores Bool de la
--lista, devolviendo un False. Utiliza guardas.
no_hay_ningun_true2 :: [Bool] -> Bool
no_hay_ningun_true2 [] = False
no_hay_ningun_true2 (x:xs)  | not (x) = True
```

```
                    | otherwise =  no_hay_ningun_true2(xs)
```

29) Realiza una función que genere una lista con las palabras "par" e "impar" según sea par o impar cada uno de los elementos de una lista que se reciba como argumento a la misma.

```
--Parámetro: un entero; devuelve: un String.
--Utiliza expresión case of. También se podría haber resuelto con una expresión if-then-else, o
--con guardas.
paridad :: Int -> String
paridad x = case mod x 2 of
              0 -> "par"
              1 -> "impar"

--En esta segunda versión, la implemento con la expresión if-then-else
paridad2 x = if mod x 2 == 0 then "par" else "impar"

--En esta tercera versión, la implemento con guardas
paridad3 | res == 0 = "par"
         | res == 1 = "impar"
    where res = mod x 2

--Parámetro una lista de enteros y devuelve una lista de String.
--Devuelve una lista de String con las cadenas par o impar según sea par o impar cada elemento
--de la lista que se recibe como parámetro.
par_impar :: [Int] -> [String]
par_impar (x) = map paridad x

--También podría haberla implementado utilizando recursividad, mediante el operador de
--construcción.
par_impar2 :: [Int] -> [String]
par_impar2 [] = []
par_impar2 (h:t) = paridad h : par_impar2 t
```