Tema: Paradigma Funcional Haskell

Objetivos: Introducción. Lenguaje Haskell 98. Conceptos básicos, características. La biblioteca Prelude. Cálculo Lambda.

1. Con sus palabras explique que es el paradigma funcional, y que entiende por programación declarativa.

Programación Funcional: *basada en la definición los predicados, en el cual realizamos funciones, que resuelven problemas de índole matemático. Este es un paradigma de programación declarativo.*

Programación Declarativa: *está basada en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Los lenguajes declarativos tienen la ventaja de ser razonados matemáticamente, lo que permite el uso de mecanismos matemáticos para optimizar el rendimiento de los programas.*

2. Realice una investigación y:

a) Indique si las siguientes son todas las funciones de la biblioteca **prelude** de haskell.

(+), (-), (*), (/), (^), (^^), (**), (==), (/=), (<), (<=), (>), (>=), (&&), (||), (:), (++), (!!), (.), abs, all, and, any, ceiling, concat, concatMap, **curry**, div, divMod, drop, dropWhile, **elem, even, filter, flip**, floor, foldl, foldl1, foldr, fromIntegral, fst, gcd**, head, init**, iterate, last, lcm**, length, map**, max, maximum, min, minimum, mod, not, notElem, null, odd, or, product, quot, quotRem, read, rem, repeat, replicate, reverse, round, scanl, scanl1, scanr, show, signum, snd, splitAt, **sqrt**, sum, tail, take, takeWhile, truncate, uncurry, until, zip, zip3, zipWith

Todas las funciones estan incluidas dentro de la biblioteca ***prelude****, son las funciones predefinidas de Haskell.*

b) Elija no menos de 10 funciones y explique cuál es su propósito y ejemplifique su uso.

- ***odd*** *x se verifica si x es impar.*

```
λ> odd 23
True
λ> odd 32
False
```

- ***elem*** *x ys se verifica si x pertenece a ys.*

```
λ> elem 3 [5,3,7]
True
λ> 3 `elem` [5,3,7]
True
λ> 4 `elem` [5,3,7]
False
```

- ***sqrt*** *x es la raíz cuadrada de x.*

```
λ> sqrt 16
4.0
```

- ***even*** *x se verifica si x es par.*

```
λ> even 6
True
λ> even 7
False
```

- ***filter*** *p xs es la lista de elementos de la lista xs que verifican el predicado p.*

```
λ> filter even [3,4,6,7,5,0]
[4,6,0]
λ> filter (<6) [3,4,6,7,5,0]
[3,4,5,0]
```

- ***flip*** *f x y es f y x.*

```
λ> let f x y = x-y
λ> (flip f) 3 7
4
```

- ***head*** *xs es el primer elemento de la lista xs.*

```
λ> head [3,2,5]
3
λ> head "Betis"
'B'
```

- ***init*** *xs es la lista obtenida eliminando el último elemento de xs.*

```
λ> init [3,7,2]
[3,7]
λ> init "colas"
"cola"
```

- ***length*** *xs es el número de elementos de la lista xs.*

```
λ> length [4,2,5]
3
λ> length "Betis"
5
```

- ***map*** *f xs es la lista obtenida aplicado f a cada elemento de xs.*

```
λ> map (^2) [3,10,5]
[9,100,25]
λ> map (2^) [3,10,5]
[8,1024,32]
λ> map even [3,10,5]
[False,True,False]
```

3. Indique cuales son los tipos de datos con los que trabaja el lenguaje haskell, de ejemplos

Tipos básicos

Bool (Valores lógicos): *Sus valores son True y False*

Char (Caracteres): *Ejemplos: 'a', 'B', '3', '+'*

String (Cadena de caracteres): *Ejemplos: "abc", "1 + 2 = 3"*

Int (Enteros de precisión fija): *Enteros entre -2^{31} y 2^{31}-1. Ejemplos: 123, -12*

Integer (Enteros de precisión arbitraria): *Ejemplos: 126765060022822940149670320537 6.*

Float (Reales de precisión arbitraria): *Ejemplos: 1.2, -23.45, 45e-7*

Double (Reales de precisión doble): *Ejemplos: 1.2, -23.45, 45e-7*

4. Escribir y probar en Haskell sus propias expresiones. Verificar en Haskell ejecutando en la consola : type de los siguientes ejemplos y luego analizar el resultado

1. :t 'a'
2. :t "abcdf"
3. :t True
4. :t 4<5
5. :t "a"
6. length "perro"
7. Deducir el tipo de length: *length:: String -> Int*
8. Verificar :type length

```
Hugs> :t 'a'
'a' :: Char
Hugs> :t "abcdf"
"abcdf" :: String
Hugs> :t True
True :: Bool
Hugs>
Hugs> :t 4<5
4 < 5 :: (Ord a, Num a) => Bool
Hugs> :t "a"
"a" :: String
Hugs> length "perro"
5
Hugs> :type length
length :: [a] -> Int
```

5. Defina que es una función y escriba funciones no recursivas para:

Una función es una aplicación de valores de un tipo en valores de otro tipo. T1 -> T1 es el tipo de las funciones que aplica valores del tipo T1 en valores del tipo T2.

Ejemplos de funciones:

not :: Bool -> Bool

isDigit :: Char -> Bool

- fecha ingresada como una cadena de números: 20112019. Se pide una funcion para que extraer el dia, luego el mes y luego el año

  cadena 20112019 = putStrLn "20/11/2019"

- averiguar si un alumno aprobó un parcial (se considera aprobado con nota mayor igual a 6)

  aprobo :: Int -> String

  aprobo x = if x >= 6 then "aprobo" else "desaprobo"

- el promedio de 3 números

  prom x y z = (x+y+z)/3

- el volumen de una esfera

```
vol x = 1.33 * 3.14 * x^3
```

- el área de un circulo´

```
área x = 3.14 * x^2
```

- la última cifra de un numero entero

```
numero x = x `mod` 10
```

- dado tres números que representan longitudes indicar si pueden formar un triangulo

```
triangulo x y z = if (x == y)&&(y == z) then "Equilatero"
                  else if (x==y)&&(y/=z) then "Isoceles"
                  else if (x/=y)&&(y/=z) then "Escaleno"
                  else "No es Triangulo"
```

- nos dice si podemos avanzar o no nuestro auto en base a la indicación del semáforo:

Rojo o amarillo implica que no avanzaremos. Verde nos permite avanzar. Sin entrar en mayores detalles

```
semaforo :: String -> String
semaforo x = if x == "verde" then "Avanzar" else "Parar"
```

6. Defina que es una lista y escriba funciones Recursiva para:

Tipos listas: *Una lista es una sucesión de elementos del mismo tipo. [T] es el tipo de las listas de elementos de tipo T.*

Ejemplos de listas:

```
[False, True]  :: [Bool]

['a','b','d']  :: [Char]

["uno","tres"] :: [String]
```

- Longitud de una cadena

```
longCadena :: String -> Int

longCadena [] = 0

longCadena (x:xs) = 1 + longCadena(xs)
```

- Dada una cadena o palabra invertir la misma

```
invertirCadena :: String -> String
invertirCadena [] = []
invertirCadena (x:xs) = invertirCadena xs ++ [x]
```

- Verificar si un elemento pertenece a una lista

```
verificaElem :: Char -> String -> Bool
verificaElem y [] = False
verificaElem y (x:xs) | y == x = True
                      | otherwise = verificaElem y xs
```

- Eliminar los elementos repetidos de una lista

```
eliminaRep :: String -> String
eliminaRep [] = []
eliminaRep (x:xs) | verificaElem x xs = eliminaRep xs
                  | otherwise = [x] ++ eliminaRep xs
```

- Unir dos listas de cadena de caracteres puede elementos repetidos. Modificar la función UnirLista de tal modo que no incluya repetidos

```
unirLista :: String -> String -> String
unirLista [] [] = []
unirLista [] (y:ys) = y : unirLista [] ys
unirLista (x:xs) y = if verificaElem x y then unirLista xs y else x : unirLista xs y
```

- para separar de un String las vocales.

  > vocales "Programacion Funcional"

  > “oaaiouioa”

```
vocales :: String -> String
vocales [] = []
vocales (x:xs) | verificaElem x "aeiou" = x : vocales xs
               | otherwise = vocales xs
```

7. Haciendo uso de la lista [1..10] definir las siguientes listas por comprension.

```
xs1 = [11,12,13,14,15,16,17,18,19,20]
xs2 = [[2],[4],[6],[8],[10]]
xs3 = [10,9,8,7,6,5,4,3,2,1]
xs4 = [True,False,True,False,True,False,True,False,True,False]
xs5 = [(3,True),(6,True),(9,True),(12,False),(15,False),(18,False)]
xs6 = [(11,12),(13,14),(15,16),(17,18),(19,20)]
```

```
lista = [x | x <- [1..10]]
lista1 = [x | x <- [11..20]]
lista2 = [[x*2]| x <- [1..5]]
lista3 = [x | x <- [10,9..1]]
lista4 = [odd x | x <- [1..10]]
lista5 = [(x*3,y) | x <- [1..6],let y = if x<=3 then True else False]
lista6 = [(x,y) | x <- [10..20],odd x, let y = x+1]
```

8. Defina que es una tupla y luego resuelva

Tipos tuplas: *Una tupla es una sucesión de elementos. (T1, T2, ..., Tn) es el tipo de las n-tuplas cuya componente i-ésima es de tipo Ti.*

Ejemplos de tuplas:

```
(False,True)     :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)
```

- La suma de los componentes de una tupla

```
sumaComp :: (Int, Int) -> Int

sumaComp (x,y) = x+y
```

- Función que recibe 2 enteros y devuelva una tupla donde el primer componente sea el mayor de los dos y la segunda el menor.

```
maxmin :: Int -> Int -> (Int, Int)

maxmin x y = if x>=y then (x,y) else (y,x)
```

- La suma de los componentes para una lista de tupla de una tupla

```
tuplas = [(x,y) | x <- [1..10],let y = x+1]

sumaTuplas = [(x+y)| x <- [1..10],let y = x+1]
```

- Contamos con una base de datos de películas representada con una lista de tuplas. Cada tupla contiene la siguiente información: (<Nombre de la película>, <Año de estreno>,<Duración de la película>,<Nombre del director>). Observamos entonces que el tipo de la tupla que representa cada película es (String, Int, Int, String).

```
peliculas = [("Mad Max",2015,2,"Juan"),("Metegol",2015,2,"Juan"),("Jumanji",2019,3,"Juan")]
```

a) Definir la función verTodas : [(String, Int, Int, String)] → Int que dada una lista de películas devuelva el tiempo que tardaría en verlas a todas.

```
verTodas :: [(String, Int, Int, String)] -> Int

verTodas [] = 0

verTodas ((a,b,c,d):xs) = c + verTodas xs
```

b) Definir la función estrenos: [(String, Int, Int, String)] → [String] que dada una lista de películas devuelva el listado de películas que estrenaron en 2019.

```
estreno :: [(String, Int, Int, String)] -> [String]

estreno [] = []

estreno ((a,b,c,d):xs) = if b == 2019 then a : estreno xs else estreno xs
```

- Programe una función similar a map, pero que aplique dos funciones a cada elemento de la lista:

  > fun cuadrado siguiente [1,2,3,4]

  > [4,9,16,25]

```
lista = [1,2,3,4]

cuadrado x = x*x

listaCuadrado :: [Int] -> [Int]

listaCuadrado [] = []

listaCuadrado (x:xs) = cuadrado (x+1) : listaCuadrado xs
```