Haskell

¿Qué es Haskell?

1. Es un lenguaje de programación
2. Utiliza el paradigma funcional
3. Los funciones son ciudadanos de primer nivel.
    a. Evaluación de expresiones en lugar de ejecución de instrucciones.
    b. Es un lenguaje funcional puro
4. Es inmutable por diseño
    a. El ser inmutable garantiza la ausencia de efectos secundarios
    b. Tiene características de idempotencia
5. Es un lenguaje con evaluación tardía (lazy)

Parte 1: La fácil

Funciones propias: Sumar dos elementos

```
sumar :: Int -> Int -> Int
sumar x y = x + y

sumar 10 15
```

```
25
```

Entrando en calor: Sumatoria con recursividad

```
sumatoria :: Int -> Int
sumatoria 0 = 0
sumatoria n = n + sumatoria (n - 1)

sumatoria 10
```

```
55
```

Condicionales: hailstone, se lo aplica con map a cada elemento de la lista

```
hailstone :: Int -> Int
hailstone n = if even n
  then n `div` 2
  else 3 * n + 1

map hailstone [5, 4, 3, 9, 17, 36]
```

```
[16,2,10,28,52,18]
```

Ejercicio: Fibonacci

```
fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)

fibonacci 10
```

```
55
```

Un fibonacci más eficiente

```
fibonacci :: Int -> Integer
fibonacci n = fibonacciHelper n 0 1

fibonacciHelper :: Int -> Integer -> Integer -> Integer
fibonacciHelper n a b = if n == 0
  then a
  else fibonacciHelper (n - 1) b (a + b)

fibonacci 1000
```

Funciones sobre listas

```
-- primeros conceptos
lst = [1, 2, 3, 4, 5, 6]

head lst
tail lst
init lst
last lst
```

```
1
[2,3,4,5,6]
[1,2,3,4,5]
6
```

Cantidad de elementos de una lista separando el elemento de su lista

```
-- ¿Qué hace este código?
misterio :: [Int] -> Int
misterio [] = 0
misterio (x:xs) = 1 + misterio xs

misterio [1, 2, 3, 4, 5]
```

potencia al cuadrado a cada elemento de la lista

```
misterioDos []     = []
misterioDos (x:xs) = (x * x) : (misterioDos xs)

misterioDos [1, 2, 3, 4, 5]
```

Ejercicio: Contar los elementos pares de una lista

```
contar :: Int -> Int
contar x = if x `mod` 2 == 0
  then 1
  else 0

contarPares :: [Int] -> Int
contarPares [] = 0
contarPares (x:xs) = (contar x) + contarPares xs

contarPares [1, 2, 3, 4, 5, 6]
```

Ejercicio: Sumar los elementos pares de una lista

```
sumar :: Int -> Int
sumar x = if x `mod` 2 == 0
  then x
  else 0

sumarPares :: [Int] -> Int
sumarPares [] = 0
sumarPares (x:xs) = (sumar x) + sumarPares xs

sumarPares [1, 2, 3, 4, 5, 6]
```

Contar “notables”

```
notable :: Int -> Int
notable x = if x `mod` 3 == 0
  then 1
  else 0

notable2 :: Int -> Int
notable2 x = if x `mod` 2 == 0
  then 1
  else 0

contarNotables :: (Int -> Int) -> [Int] -> Int
contarNotables f [] = 0
contarNotables f (x:xs) = (f x) + contarNotables f xs

contarNotables notable2 [1, 2, 3, 4, 5, 6]
```

Se permite pasar funciones por parámetros

Interludio: Algunas funciones pre-cocidas sobre listas

zip: Combina dos listas en una lista de tuplas

```
numbers1 = [1..4]
numbers2 = [10,20..40]
zip numbers1 numbers2
```

```
[(1,10),(2,20),(3,30),(4,40)]
```

Ejemplo práctico: Supongamos que tenemos las notas del primer parcial, y del trabajo práctico. Queremos conocer el mayor promedio del curso.

```
npp = [4, 8, 7, 9, 2]
ntp = [5, 8, 9, 10, 6]

promedio (x, y) = (x + y)/2

promedios = map promedio ( zip npp ntp )
maximum promedios
```

map: Es una función que aplica una operación a cada elemento de una lista y devuelve una nueva lista con los resultados

```
numbers = [10, 20, 30]
map (*2) numbers
```

```
[20,40,60]
```

fold (foldl / foldr): Combina los elementos de una lista en un solo valor

```
numbers = [1, 2, 3]
foldl (^) 2 numbers
```

```
64
```

```
foldl f z [x1, x2, x3] == ((z `f` x1) `f` x2) `f` x3
```

filter: Filtra una lista por una condición dada

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
filter even numbers
```

```
[2,4,6,8,10]
```

elem: Es una función que toma un elemento y una lista, y devuelve un valor booleano que indica si el elemento dado está presente en la lista.

```
lst = [1, 2, 3, 4, 5]
elem 3 lst
elem 99 lst
```

Parte 2: La moderada

Pattern Matching

Es una técnica que permite descomponer y comparar patrones en datos estructurados. Se utiliza para realizar diferentes acciones o tomar decisiones según los patrones encontrados en los valores de entrada.

```
quitaTres :: [a] -> [a]
quitaTres (_:_:_:xs) = xs
quitaTres _          = []

quitaTres [1,2,3,4,5]
quitaTres [1,2]
```

```
[4,5]
[]
```

Dos operadores notables

- El operador ++ sirve para concatenar dos listas
- El operador : , en cambio, sirve para agregar elementos antes de las listas

```
x = 1
xs = [2, 3]

y = 9
ys = [8, 7]
{-
x:xs    -- válido
x++xs   -- no válido

xs:ys   -- no válido
xs++ys  -- válido
-}
x:xs
xs++ys
```

```
[1,2,3]
[2,3,8,7]
```

Ejercicios: Cola y Pila

```
-- cola
enqueue :: a -> [a] -> [a]
enqueue x xs = xs++[x]

enqueue 9 [1, 2, 3]
enqueue 4 []
```

```
[1,2,3,9]
[4]
```

```
-- cola
dequeue :: [a] -> (a, [a])
dequeue (x:xs) = (x, xs)
dequeue [] = error "cola vacia"

dequeue [1, 2, 3, 4]
--dequeue []
```

```
(1,[2,3,4])
```

```haskell
-- pila
push :: a -> [a] -> [a]
push x xs = x:xs

push 9 [1, 2, 3]
push 3 []
```

```
[9,1,2,3]
[3]
```

```haskell
-- pila
pop :: [a] -> (a, [a])
pop (x:xs) = (x, xs)
pop [] = error "pila vacia"
```

(1,[2,3,4])

Parte 3: La difícil

Currying

Es el proceso de transformar una función de múltiples argumentos en una secuencia de funciones que toman un solo argumento. Esto permite la aplicación parcial de argumentos y la creación de funciones más genéricas y flexibles.

```haskell
-- function :: Int -> Int -> Int
function x y = x + y

-- fun :: Int -> Int
-- fun y = function 3 y
fun = function 3

fun 2
```

Currying: Un ejemplo práctico

```haskell
sumar x y = x + y
incrementar = sumar 1

incrementar 10
```

Parte 4: Ejercicios

Alfa

--1. Dado dos números enteros A y B, implemente una función que retorne la división entera de ambos por el método de las restas sucesivas

```haskell
divRestas :: Int -> Int -> Int
divRestas a b = if a < b
  then 0
  else 1 + divRestas (a - b) b
```

--2. Escribir una función para hallar la potencia de un número

```haskell
potencia :: Int -> Int -> Int
potencia a 0 = 1
potencia a 1 = a
potencia a b = a * potencia a (b-1)
```

--3. Definir una función menor que devuelve el menor de sus dos argumentos enteros

```haskell
menor :: Int -> Int -> Int
menor a b = if a < b
  then a
  else b
```

--4. Definir una función maximoDeTres que devuelve el máximo de sus argumentos enteros

```haskell
mayor :: Int -> Int -> Int
mayor a b = if a > b
```

```
  then a
  else b

maximoDeTres :: Int -> Int -> Int -> Int
maximoDeTres a b c = mayor (mayor a b) c
```

Omega

```
--1. Escribir una función que sume dos números enteros.
sumar :: Int -> Int -> Int
sumar a b = a + b

--2. Implementar una función que calcule el área de un círculo dado su radio.
areaCirc :: Float -> Float
areaCirc a = pi * a * a

--3. Definir una función que determine si un número es par o impar.
esPar :: Int -> Bool
esPar a = mod a 2 == 0

--4. Escribir una función que calcule el factorial de un número.
factorial :: Int -> Int
factorial 0 = 1
factorial a = a * factorial (a-1)

--5. Implementar una función que invierta una lista.
invertir :: [a] -> [a]
invertir [] = []
invertir (x:xs) = invertir (xs) ++ [x]

--6. Definir una función que determine si una lista está ordenada de forma ascendente.
esOrdAsc :: [Int] -> Bool
esOrdAsc [a] = True
esOrdAsc (x:y:xs) =  x <= y && esOrdAsc (y:xs)

--7. Escribir una función que cuente la cantidad de elementos en una lista.
cantElem :: [a] -> Int
cantElem [] = 0
cantElem [a] = 1
cantElem (x:xs) = 1 + cantElem xs

--8. Implementar una función que obtenga los elementos en posiciones pares de una lista.
elemPosPares :: [Int] -> [Int]
elemPosPares [] = []
elemPosPares [a] = [a]
elemPosPares (x:y:xs) = x:elemPosPares xs

--8.1. Implementar una función que obtenga los elementos pares de una lista.
esPar :: Int -> Bool
esPar a = mod a 2 == 0

elemPares :: [Int] -> [Int]
elemPares [] = []
elemPares (x:xs) = if esPar x
  then x:elemPares (xs)
  else elemPares xs

--9. Definir una función que calcule el máximo común divisor de dos números.
mcd :: Int -> Int -> Int
mcd a 0 = a
mcd a b = mcd b (mod a b)

--11. Implementar una función que calcule la suma de los dígitos de un número entero.
sumaDigitos :: Int -> Int
```

```
sumaDigitos 0 = 0
sumaDigitos n = mod n 10 + sumaDigitos (div n 10)

--12. Definir una función que encuentre el elemento mínimo en una lista.
menor :: Int -> Int -> Int
menor a b = if a < b
  then a
  else b

elemMinimo :: [Int] -> Int
elemMinimo [a] = a
elemMinimo (x:xs) = menor x (elemMinimo xs)

--13. Escribir una función que obtenga el enésimo número de la secuencia de Fibonacci.
fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n-2) + fibonacci (n-1)

--14. Implementar una función que verifique si una cadena de texto es un palíndromo.
esPalindromo :: String -> Bool
esPalindromo s = s == reverse s

esPalindromo2 :: String -> Bool
esPalindromo2 [] = True
esPalindromo2 [x] = True
esPalindromo2 (x:xs) = x == last xs && esPalindromo2 (init xs)

--15. Definir una función que elimine los duplicados de una lista.
elimDupli :: [Int] -> [Int]
elimDupli [] = []
elimDupli [a] = [a]
elimDupli (x:xs) if elem x xs
  then elimDupli xs
  else x : elimDupli xs

--16. Implementar una función que obtenga el producto de todos los elementos de una lista.
prodLst :: [Int] -> Int
prodLst [a] = a
prodLst (x:xs) = a * prodLst xs
```