Lab 02

David Rivera Morales -- 320176876

Israel Rivera -- XXXXXXX

La función potencia de un número

Dentro de la función potencia hay un caso base donde se evalúa si el exponente es igual a 0, en caso de serlo, se retorna 1, de lo contrario se retorna el número multiplicado por la función potencia con el número y el exponente - 1.

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

La función suma_pares

En la funcion suma_pares se evalúa en el caso semilla si el número es 0 , en caso de serlo se retorna 0, de lo contrario se evalúa si el número es par, en caso de serlo se retorna el número más la función suma_pares con el número - 1, de lo contrario se retorna la función suma_pares con el número - 1.

```
suma_pares :: Int -> Int
suma_pares 0 = 0
suma_pares n = if mod n 2 == 0 then n + suma_pares (n-1) else suma_pares
(n-1)
```

La función triangular

En la función triangular se evalúa en el caso semilla si el número es 0, en caso de serlo se retorna 0, de lo contrario se retorna el número más la función triangular con el número - 1 por lo que se va sumando el número con los anteriores.

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

La función fibo

Para esta función se utilizó un caso base donde se evalúa si el número es 0 ya que en la serie de fibonacci el primer número es 0, en caso de serlo se retorna 0, de lo contrario se evalúa si el número es 1, en caso de

serlo se retorna 1 porque es el segundo número de la serie y también forma parte de un caso base, de lo contrario se retorna la función fibo con el número - 1 más la función fibo con el número - 2 ya que la serie de fibonacci es la suma de los dos números anteriores.

```
fibo :: Int -> Int
fibo 0 = 0
fibo n = if n == 1 then 1 else fibo (n -1) + fibo (n -2)
```

La función ultimo

Esta función se evalúa en el caso semilla si la lista es vacía, en caso de serlo se retorna error porque no hay ningún elemento en la lista, de lo contrario se evalúa si la lista es de un solo elemento, en caso de serlo se retorna el primer elemento de la lista porque el primero es el último, de lo contrario se retorna la función ultimo con la cola de la lista con un elemento menos hasta que la lista sea de un solo elemento.

```
ultimo :: [a] -> a
ultimo [] = error "No hay elementos"
ultimo [x] = x
ultimo (x : xs) = ultimo xs
```

La función reversa

En la función reversa vemos si la lista es vacía, si es así se regresa un error porque no hay elementos en la lista, de lo contrario se evalúa si la lista es de un solo elemento, en caso de serlo se retorna la lista porque es el último elemento, de lo contrario se retorna la función reversa con la cola de la lista más el primer elemento de la lista por lo que se va agregando el último elemento de la lista al principio de la lista hasta que la lista esté al revés.

```
reversa :: [a] -> [a]
reversa [] = error "No hay elementos"
reversa [x] = [x]
reversa (x : xs) = reversa xs ++ [x]
```

La función elemento

Lo que hace esta función es evaluar si la lista es vacía, en caso de serlo se retorna falso porque no hay ningún elemento en la lista, de lo contrario se evalúa si el primer elemento de la lista es igual al elemento que se está buscando, en caso de serlo se retorna verdadero porque se encontró el elemento, de lo contrario se retorna la función elemento con la cola de la lista y el elemento que se está buscando para evaluar si el siguiente elemento es igual al elemento que se está buscando.

```
elemento :: Eq a => a -> [a] -> Bool
elemento elm [] = False
```

```
elemento elm (x : xs) = if elm == x then True else elemento elm xs
```

La función nicomaco

En la función nicomaco se evalúa si el número es 0 como caso base, en caso de serlo se regresa Deficiente ya que no es ni perfecto ni abundante, de lo contrario se evalúa si el número es igual a la suma de los divisores del número, en caso de serlo se retorna Perfecto ya que la suma de los divisores del número es igual al número, de lo contrario se evalúa si el número es mayor a la suma de los divisores del número, en caso de serlo se retorna Abundante ya que la suma de los divisores del número es menor al número, de lo contrario se retorna Deficiente ya que la suma de los divisores del número es mayor al número. Se hace uso de la función suma_divisores para obtener la suma de los divisores del número donde se ve si el modulo del número

```
data Categoria = Perfecto | Deficiente | Abundante

nicomaco :: Int -> Categoria
nicomaco 0 = Deficiente
nicomaco n = if suma_divisores n == n then Perfecto else if suma_divisores
n < n then Deficiente else Abundante

suma_divisores :: Int -> Int
suma_divisores 0 = 0
suma_divisores n = if mod n 2 == 0 then n + suma_divisores (n -1) else
suma_divisores (n -1)
```

```
data EA
  = N Int
  | Positivo EA
  | Negativo EA
  | Suma EA EA
  | Resta EA EA
  | Mult EA EA
  | Div EA EA
  | Mod EA EA
  | Pot EA EA

instance Show EA where
  show (N x) = if x >= 0 then show x else "(" ++ show x ++ ")"
  show (Positivo (N (x))) = show x
  show (Negativo (N (x))) = show (- x)
  show (Suma x y) = show x ++ " + " ++ show y
  show (Resta x y) = show x ++ " - " ++ show y
  show (Mult x y) = show x ++ " * " ++ show y
  show (Div x y) = show x ++ " / " ++ show y
  show (Mod x y) = show x ++ " % " ++ show y
  show (Pot x y) = show x ++ " ^ " ++ show y

creaSumaEA :: Int -> Int -> EA
```

```
creaSumaEA x y = Suma (N x) (N y)

creaRestaEA :: Int -> Int -> EA
creaRestaEA x y = Resta (N x) (N y)

creaMultEA :: Int -> Int -> EA
creaMultEA x y = Mult (N x) (N y)

creaDivEA :: Int -> Int -> EA
creaDivEA x y = Div (N x) (N y)

creaModEA :: Int -> Int -> EA
creaModEA x y = Mod (N x) (N y)

creaPotEA :: Int -> Int -> EA
creaPotEA x y = Pot (N x) (N y)

menorque :: EA -> EA -> Bool
menorque (N x) (N y) = x < y
menorque (N x) (Positivo (N y)) = x < y
menorque (N x) (Negativo (N y)) = x < y

mayorque :: EA -> EA -> Bool
mayorque (N x) (N y) = x > y
mayorque (N x) (Positivo (N y)) = x > y
mayorque (N x) (Negativo (N y)) = x > y
```