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 ya que 0 tiene divisores infinitos por lo cual se retorna un error, de lo contrario se evalúa si la suma de los divisores es mayor que el número, en caso de serlo se retorna abudante , si ninguno de los casos anteriores se cumple se concluye que el número es deficiente y se retorna deficiente. Para calcular los divisores se utiliza la función suma_alquota que toma un caso base donde 0 tiene 0 divisores , si no es así se hace la suma de los divisores con sum que forma parte de la librería predefinida de Haskell , en sum lo que es hacer una lista con los divisores a partir de 1 hasta el número - 1 y se evalúa si el número es divisible entre el divisor, en caso de serlo se agrega el divisor a la lista de divisores, de lo contrario se agrega 0 a la lista de divisores.

```
data Categoria = Perfecto | Deficiente | Abundante deriving (Show, Eq)

nicomaco :: Int -> Categoria
nicomaco 0 = error "Es imposible calcular ya que 0 no es un número
enteramente positivo"
nicomaco n = if suma_aliquota n == n then Perfecto else if suma_aliquota n
> n then Abundante else Deficiente

suma_aliquota :: Int -> Int
suma_aliquota 0 = 0
suma_aliquota n = sum [x | x <- [1 .. n -1], mod n x == 0]
```

La función luhn

Dentro de esta hay dos métodos auxiliares que son lunhAuxDouble y luhnAuxSum. La función luhnAuxDouble recibe una lista de enteros y regresa una lista de enteros donde se duplica cada elemento de la lista en caso de que la posición del elemento sea par, de lo contrario se regresa el elemento. La función luhnAuxSum recibe una lista de enteros y regresa un entero donde se suman todos los elementos de la lista. La función luhn recibe una lista de enteros y regresa un booleano donde se evalúa si la suma de los elementos de la lista es divisible entre 10, en caso de serlo se regresa verdadero, de lo contrario se regresa falso.

```
luhn :: [Int] -> Bool
luhn [] = error "No es válida"
luhn (x : xs) = if mod (luhnAuxSum (luhnAuxDouble (x : xs))) 10 == 0 then
True else False

luhnAuxDouble :: [Int] -> [Int]
luhnAuxDouble [] = []
luhnAuxDouble (x : xs) = if mod (length (x : xs)) 2 == 0 then x * 2 :
luhnAuxDouble xs else x : luhnAuxDouble xs

luhnAuxSum :: [Int] -> Int
luhnAuxSum [] = 0
```

```
luhnAuxSum (x : xs) = if x > 9 then x -9 + luhnAuxSum xs else x +
luhnAuxSum xs
```

La función pasosCollatz

En esta función se implementa el algoritmo de Collatz, el cual consiste en tomar un número entero positivo y si es par se divide entre 2, si es impar se multiplica por 3 y se le suma 1, se repite el proceso hasta que el número sea 1. La función pasosCollatz recibe un entero positivo y regresa un entero donde se evalúa si el número es 1, en caso de serlo se regresa 0, de lo contrario se evalúa si el número es par, en caso de serlo se regresa 1 + pasosCollatz (div n 2), de lo contrario se regresa 1 + pasosCollatz (n * 3 + 1) , el 1 es para contar el paso que se está haciendo.

```
pasosCollatz :: Int -> Int
pasosCollatz 1 = 0
pasosCollatz x = if mod x 2 == 0 && x >= 0 then 1 + pasosCollatz (div x 2)
else 1 + pasosCollatz (3 * x + 1)
```

La función listaCollatz

Dentro de esta función se implementa nuevamente el algoritmo de Collatz, la diferencia es que en lugar de regresar un entero se regresa una lista de enteros donde se van agregando los pasos que se van haciendo. La función listaCollatz recibe un entero positivo y regresa una lista de enteros donde se evalúa si el número es 1, en caso de serlo se regresa la lista con el 1 porque la lista ya terminó, de lo contrario se evalúa si el número es par, en caso de serlo se regresa la lista con el número y se llama recursivamente a la función listaCollatz con el número dividido entre 2, de lo contrario se regresa la lista con el número y se llama recursivamente a la función listaCollatz con el número multiplicado por 3 y se le suma 1 hasta que el número sea 1.

```
listaCollatz :: Int -> [Int]
listaCollatz 1 = [0]
listaCollatz x = if mod x 2 == 0 && x >= 0 then x : listaCollatz (div x 2)
else x : listaCollatz (3 * x + 1)
```

Expresiones aritméticas.

Dentro de este inciso lo importante es hacer la instancia de la clase Show para poder imprimir las expresiones aritméticas, para esto se utiliza la función show que toma una expresión aritmética y regresa un string. Para poder hacer la instancia de la clase Show se utiliza la palabra reservada instance seguida de la clase Show y el tipo de dato que se quiere hacer la instancia, en este caso EA. En la función show se evalúa el tipo de dato que es la expresión aritmética y se regresa el string correspondiente .Por ejemplo si Positivo (N (x)) es el tipo de dato , se regresa el string "x" ya que es un número positivo. Aunado a esto , para implementar las demás funciones se deben utilizar tipos de dato EA ya que se están implementando funciones que reciben expresiones aritméticas y regresan expresiones aritméticas por lo que también se debe pasar tipos de dato EA a la hora de llamar a las funciones.

```haskell
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
```