Informe de práctica 1 – Programación Funcional en Haskell

Integrantes:

- Samuel Valencia Montoya
- Jerónimo Echeverry

En esta primera práctica se desarrollaron tres programas en Haskell aplicando conceptos de programación funcional, recursión y restricciones en el uso de funciones (solo las fundamentales: +, -, *, /, mod, toEnum, fromEnum, fromIntegral, cos).

Primer ejercicio: Eliminar datos fuera de un intervalo

Explicación de las funciones:

Primera función deleteFunction:

deleteFunction elimina de una lista todos los valores que no estén dentro del intervalo [a,b].

Entradas: una lista de enteros, un entero a (límite inferior) y un entero b (límite superior).

Salida: una lista de enteros con sólo los valores dentro del intervalo.

Ejemplo:

```
deleteFunction :: [Int] -> Int -> Int -> [Int]

deleteFunction [] _ _ = []

deleteFunction (x:xs) a b

| inRange x a b = x : deleteFunction xs a b

| otherwise = deleteFunction xs a b
```

Segunda función inRange:

inRange determina si un número está dentro del intervalo [a, b].

Entradas: un entero x, entero a, entero b.

Salida: True si a <= x <= b, False en otro caso.

Ejemplo:

inRange :: Int -> Int -> Int -> Bool

inRange x a b = (lessEqual a x) && (lessEqual x b)

-Tercera función lessEqual:

lessEqual compara dos enteros x y y, devolviendo True si x <= y.

Entradas: dos enteros x e y.

Salida: True si x <= y, False en caso contrario.

Ejemplo:

lessEqual :: Int -> Int -> Bool

lessEqual x y = not (higher x y)

Cuarta función higher:

higher compara dos enteros x y y, devolviendo True si x > y.

Entradas: dos enteros x e y.

Salida: True si x > y, False en caso contrario.

Ejemplo:

higher :: Int -> Int -> Bool

higher x y = (substraction x y) > 0

Quinta función substraction:

substraction calcula la resta entre dos enteros.

Entradas: dos enteros x e y.

Salida: el valor x - y.

Ejemplo:

substraction :: Int -> Int -> Int

substraction x y = x - y

Problemas:
En este primer ejercicio, el principal inconveniente fue la restricción de únicamente poder

usar las funciones fundamentales (+, −, ∗, /, mod, toEnum, fromEnum, fromIntegral, cos). Esto nos impidió utilizar funciones ya existentes como filter, que permite filtrar los elementos de una lista dentro de un rango específico y eliminar los que no cumplen la condición. Como solución, fue necesario crear múltiples funciones que, mediante recursividad, replicaran el comportamiento de filter.

Segundo ejercicio: Ordenar una lista de flotantes en orden descendente

Explicación de funciones:

Primera función: orderList

orderList ordena una lista de números flotantes en orden descendente.

Entrada: una lista de flotantes.

Salida: la misma lista pero ordenada de mayor a menor.

Ejemplo:

```
orderList :: [Float] -> [Float]

orderList [] = []

orderList xs =

  let m = top xs

  in m : orderList (removeFirst m xs)
```

Segunda función: top

top devuelve el mayor elemento de una lista de flotantes.

Entrada: una lista de flotantes.

Salida: el flotante máximo en la lista.

Ejemplo:

```
top :: [Float] -> Float

top [x] = x

top (x:xs)

  | greaterThan x (top xs) = x
```

```
  | otherwise = top xs
```

Tercera función: removeFirst

removeFirst elimina la primera ocurrencia de un elemento en la lista.

Entradas: un flotante y una lista de flotantes.

Salida: la lista sin la primera ocurrencia de ese flotante.

Ejemplo:

```
removeFirst :: Float -> [Float] -> [Float]
removeFirst _ [] = []
removeFirst y (x:xs)
  | equalsF x y = xs
  | otherwise = x : removeFirst y xs
```

Cuarta función: greaterThan

greaterThan compara dos flotantes y devuelve True si el primero es mayor.

Entradas: dos flotantes x e y.

Salida: True si x > y, False en otro caso.

Ejemplo:

```
greaterThan :: Float -> Float -> Bool
greaterThan x y = (substractF x y) > 0
```

Quinta función: equalsF

equalsF compara dos flotantes y devuelve True si son iguales.

Entradas: dos flotantes x e y.

Salida: True si x == y, False en otro caso.

Ejemplo:

equalsF :: Float -> Float -> Bool

equalsF x y = not (greaterThan x y) && not (greaterThan y x)

Sexta función: substractF

substractF calcula la resta de dos flotantes.

Entradas: dos flotantes x e y.

Salida: el resultado x - y.

Ejemplo:

substractF :: Float -> Float -> Float

substractF x y = x - y

Problemas:

Tercer ejercicio: Aproximar la función exponencial, coseno(x) y ln(1 + x)

Explicación de funciones:

Primera función: factorial

Calcula el factorial de un número entero.

Entrada: Un entero n

Salida: el resultado de: n * (n-1) *.. * 1

Ejemplo:

factorial :: Int -> Int

factorial 0 = 1

factorial n = n * factorial (n-1)

Segunda función: potencia

Calcula la potencia de un double elevado a un entero positivo.

Entrada: Un double x, y un entero n.

Salida: El resultado de x ^ n.

Ejemplo:

potencia :: Double -> Int -> Double

potencia _ 0 = 1

potencia x n = x * potencia x (n-1)

Tercera función: ex_aprox

Aproxima el valor de e^x usando la serie de Taylor.

Entrada: Un double x, y el número de términos n.

Salida: Aproximación de e^x.

Ejemplo:

ex_aprox :: Double -> Int -> Double

ex_aprox x n = sumaExp x n 0

Cuarta función: sumaExp

Realiza la suma de los términos de la serie de Taylor.

Entrada: Un double x, el número de términos n y el índice k.

Salida: Suma de la sucesión e^x.

Ejemplo:

sumaCos :: Double -> Int -> Int -> Double

sumaCos x n k =

  if k > n then 0

  else (potencia x k / fromIntegral (factorial (k)) + sumaExp x n (k+1)

Quinta función: cosAprox

Aproxima el valor de coseno con serie de Taylor.

Entrada: Un double x, y un número de términos n.

Salida: Aproximación de cos(x).

Ejemplo:

```
cos_aprox :: Double -> Int -> Double

cos_aprox x n = sumaCos x n 0
```

Sexta función: sumaCos

Entrada: un double x, el número de términos n, y el índice k

Salida: término actual de la serie de Taylor de cos(x) para k + recursión

Ejemplo:

```
sumaCos :: Double -> Int -> Int -> Double

sumaCos x n k =

  if k > n then 0

   else (((potencia (-1) k) * (potencia x (2*k))) / fromIntegral (factorial (2*k))) + sumaCos x n (k+1)
```

Séptima función: lnAprox

lnAprox, aproxima ln(1+ x) con serie de Taylor.

Entrada: x ∈ (-1, 1] y número de términos n.

Salida: Aproximación de ln(1 + x).

Ejemplo:

```
ln_aprox :: Double -> Int -> Double

ln_aprox x n = sumaLn x n 1
```

Octava función: sumaLn

Calcula la suma de los términos de la serie de Taylor para ln(1 + x)-

Entrada: Entrada: un double x, el número de términos n, y el índice k.

Salida: Suma parcial de la serie ln(x + 1).

Ejemplo:

```
sumaLn :: Double -> Int -> Int -> Double

sumaLn x n k =
```

```
  if k > n then 0
  else (((potencia (-1) (k+1)) * (potencia x k)) / fromIntegral k) + sumaLn x n (k+1)
```

Cuarto ejercicio: Técnicas de procesamiento de señales discretas

Explicación de funciones:

Primera función: piVal

Es valor de pi

Salida: el valor aproximado de π (3.141592653589793)

Ejemplo:

```
piVal :: Double
piVal = 3.141592653589793
```

Segunda función: absVal

absVal Calcula el valor absoluto de un número

Entrada: un Float

Salida: x si es positivo o -x si es negativo (Float)

Ejemplo:

```
absVal :: Double -> Double
absVal x = if x >= 0 then x else (0 - x)
```

Tercera función: sqrtNewton

sqrtNewton Calcula la raíz cuadrada de un número usando el método de Newton-Raphson.

Entrada: un número flotante

Salida: la √x (Float)

Ejemplo:

```
sqrtBusca :: Double -> Double

sqrtBusca x = busca 0

  where

        paso = 0.0001   -- precisión

        busca y =

        if y*y > x

        then y

        else busca (y + paso)
```

Cuarta función: aCoef

aCoef Coeficiente de normalización para la DCT $\sqrt{\frac{1}{N}}$

Entradas: un índice k (Int), y el tamaño de la lista n (Int)

Salida: sqrt(1/n) si k=0, en otro caso sqrt(2/n).

Ejemplo:

```
aCoef :: Int -> Int -> Double

aCoef k n =

  if k == 0

  then sqrtBusca (1 / fromIntegral n)

  else sqrtBusca (2 / fromIntegral n)
```

Quinta función: nth

nth Obtiene el n-ésimo elemento de una lista (Recorrer lista)

Entradas: una lista [a] y un índice n (Int)

Salida: el elemento número n de la lista (empieza desde 0).

Ejemplo:

```
nth :: [a] -> Int -> a

nth (y:ys) n = if n == 0 then y else nth ys (n-1)
```

Sexta función: len

len Calcula la longitud de una lista

Entrada: una lista [a].

Salida: un entero con la cantidad de elementos.

Ejemplo:

```
len :: [a] -> Int

len [] = 0

len (_:xs) = 1 + len xs
```

Séptima función: sumatoria

sumatoria Suma los valores de una función desde i = 0 hasta i = n-1.

Entradas: (Int -> Double) y un entero n.

Salida: Double

Ejemplo:

```
sumatoria :: (Int -> Double) -> Int -> Double

sumatoria f n =

  if n == 0

  then 0

  else f (n-1) + sumatoria f (n-1)
```

Octava función: dctTerm

dctTerm Calcula un término individual de la DCT para un índice k

Entradas:

xs :: [Double] → lista de datos.

k :: Int → índice de frecuencia.

n :: Int → índice de posición en la lista.

bigN :: Int → longitud de la lista.

Salida: x[n] * cos(((n+0.5) * π * k) / N) (Double)

Ejemplo:

```
dctTerm :: [Double] -> Int -> Int -> Int -> Double
dctTerm xs k n bigN =
(nth xs n) * cos (((fromIntegral n + 0.5) * piVal * fromIntegral k) / fromIntegral bigN)
```

Novena función: dctK

dctK Calcula el valor X(k) de la DCT.

Entradas: lista Double, índice k (Int)

Salida: el coeficiente X(k) de la transformada (Double)

Ejemplo:

```
dctK :: [Double] -> Int -> Double
dctK xs k =
  let bigN = len xs
  ak = aCoef k bigN
  suma = sumatoria (\n -> dctTerm xs k n bigN) bigN
  in ak * suma
```

Décima función: dct

Dct Calcula la DCT completa de una lista

Entrada: lista Double

Salida: lista con todos los coeficientes de la DCT (Lista Double)

Ejemplo:

```
dct :: [Double] -> [Double]
dct xs = dctRec xs 0 (len xs)
  where
  dctRec xs k n =
  if k == n
  then []
  else dctK xs k : dctRec xs (k+1) n
```

Onceava función: round4

round4 Redondea un número flotante a 4 decimales.

Entrada: Double.

Salida: x redondeado a 4 decimales (Double)

Ejemplo:

```
round4 :: Double -> Double
round4 x =
  let escala = 10000.0
  valor = x * escala
  ajustado = if x >= 0 then valor + 0.5 else valor - 0.5
  entero = fromEnum ajustado
  in fromIntegral entero / escala
```

Doceava función: formatList

formatList aplica round4 a todos los elementos de una lista.

Entrada: Lista Double

Salida: Lista Double con cada elemento redondeado a 4 decimales

Ejemplo:

formatList :: [Double] -> [Double]

formatList [] = []

formatList (x:xs) = round4 x : formatList xs

Treceava función: dctFinal

dctFinal Calcula la DCT y luego redondea los resultados.

Entrada: lista Double

Salida: lista Double con los coeficientes X(k) redondeados a 4 decimales

Ejemplo:

dctFinal :: [Double] -> [Double]

dctFinal xs = formatList (dct xs)

Problemas: En este ejercicio fue donde más problemas tuvimos, ya que fue el más complejo por aplicar tanta fórmulas matemáticas, por ejemplo, para hacer la raíz del Coeficiente de normalización para la DCT $\sqrt{\frac{1}{N}}$ sin usar funciones ya definidas en Haskell, sin embargo, encontramos forma de hacerlo, con la función sqrtBusca, la cual empieza en 0 como valor de prueba, va aumentando el valor poco a poco, sumando 0.0001 cada vez. En cada intento, multiplica el número por sí mismo (y*y), cuando ese cuadrado es mayor que el número de entrada x, se detiene y devuelve el valor actual como aproximación de la raíz cuadrada. Otro problema que tuvimos fue al momento de hacer la función nth, la cual obtiene el n-ésimo elemento de una lista (Recorrer lista), ya que debería trabajar con diferentes tipos de datos, por lo que usamos una variable comodín, a, la cual puede trabajar como cualquier tipo de dato, ya sea Int, Float, Char, Bool, etc.