Ejemplos de Haskell

Lista de ejemplos

- Traductor de expresiones
- Máximo común divisor
- Búsqueda en profundidad en grafos
- Insertion-sort – Ordenamiento de listas
- Quick-sort – Ordenamiento de listas
- Eliminar duplicados de una lista
- Criba de Eratóstenes
- Factorial
- Comprobación de equivalencias
- Derivada de una función
- Árboles binarios
- Árbol de búsqueda binaria
- Ordenamiento mediante árbol de búsqueda binaria

Traductor de expresiones

Se definen los tipos de expresión aritmética:

```
data Expr = Núm Integer
          | Sum Expr Expr
          | Pro Expr Expr
```

Esto da a lugar a expresiones del tipo:

```
Expr = Sum (Núm 3) (Pro (Núm 4) (Núm 6))
```

Se define la función:

```
muestraExpr  :: Expr -> String
muestraExpr (Núm n)
   | n < 0 = "(" ++ show n ++ ")"
   | otherwise = show n
muestraExpr (Sum a b) = muestraExpr a ++ "+" ++ muestraExpr b
muestraExpr (Pro a b) = muestraFactor a ++ "*" ++ muestraFactor b

muestraFactor :: Expr -> String
muestraFactor (Sum a b) = "(" ++ muestraExpr (Sum a b) ++ ")"
muestraFactor e = muestraExpr e

instance Show Expr where
   show = muestraExpr
```

Se llama a la función *muestraExpr Sum (Núm 3) (Pro (Núm 4) (Núm 6))*, cuyo resultado es: "3+4*6"

Máximo común divisor

Se define la función n_gcd con dos parámetros numéricos:

```
n_gcd 20 12 -> 4
```

Se presentan distintas soluciones:

Definición recursiva:

```
n_gcd1 :: Int -> Int -> Int
n_gcd1 0 0 = error "gcd 0 0 no está definido"
n_gcd1 x y = n_gcd1' (abs x) (abs y)
   where n_gcd1' x  0 = x
        n_gcd1' x y = n_gcd1' y (x `rem` y)
```

Definición con divisible y divisores:

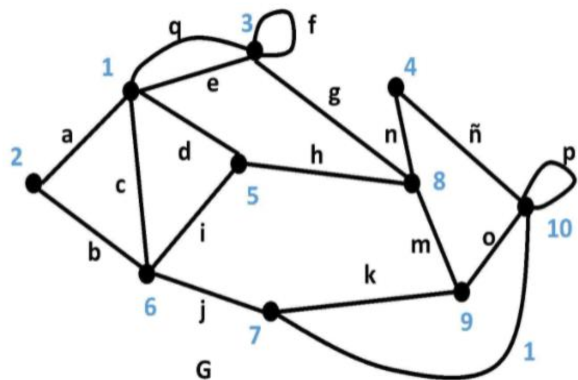
```
n_gcd2 :: Int -> Int -> Int
n_gcd2 0 0 = error "gcd 0 0 no está definido"
n_gcd2 0 y = abs y
n_gcd2 x y = last (filter (divisible y') (divisores x'))
   where x' = abs x
        y' = abs y
        divisores x = filter (divisible x) [1..x]
        divisible x y = x `rem` y == 0
```

En esta última implementación, se obtiene el último elemento de la lista construida luego de encontrar todos los divisores de "x" que son divisibles por "y".

Búsqueda en profundidad en grafos

Se trabaja sobre el siguiente grafo de ejemplo:

```
ej_grafo = G [1..5] suc
   where suc 1 = [2, 6, 5, 3]
        suc 2 = [1, 6]
        suc 3 = [1, 8]
        suc 4 = [8, 10]
        suc 5 = [1, 6, 8]
        suc 6 = [2, 1, 5, 7]
        suc 7 = [6, 9]
        suc 8 = [3, 5, 4, 9]
        suc 9 = [7, 8, 10]
        suc 10 = [4, 9, 7]
```



Se llama a la función *camino*, que establece una ruta entre el nodo de origen y el nodo de destino realizando una búsqueda en profundidad.

> Camino ej_grafo 1 10

El primer término corresponde con el grafo, el segundo con el nodo de origen, y el tercero con el nodo de destino. El resultado es el siguiente:

= [10, 4, 8, 5, 6, 2, 1]

```
camino :: Eq a => Grafo a -> a -> a -> [a]
camino g u v = head (caminosDesde g u (== v) [])
```

la función *camino* llama a la función *caminosDesde*, que establece una lista de todos los caminos del grafo que comienzan en el nodo origen y van hasta el nodo destino.

> caminosDesde ej_grafo 1 (==10) []

El primer término corresponde con el grafo, el segundo con el nodo de origen, la expresión booleana corresponde con el nodo destino que se quiere buscar, y la lista vacía es en donde se irán concatenando todos los caminos que se encuentren en la búsqueda en profundidad.

```
caminosDesde :: Eq a => Grafo a -> a -> (a -> Bool) -> [a] -> [[a]]
caminosDesde g o te vis
   | te o = [o:vis]
   | otherwise = concat [caminosDesde g o' te (o:vis)
      | o' <- suc o,
      notElem o' vis]
   where G _ suc = g
```

En caso de que o sea igual a *te*, solo se hace la concatenación del elemento, pues ya hemos llegado a nuestro destino, si no, se hace la concatenación de todos los caminos desde el *suc o* hasta nuestro nodo destino y además de ello, introducimos la *o* a nuestra lista de caminos, pues este nodo ya ha sido visitado.

Insertion-sort – Ordenamiento de listas

En primer lugar, se define la función *inserta* que nos permitirá insertar un número en una lista de números ordenados, delante del primer elemento que sea mayor o igual a ese número. Por ejemplo:

> Inserta 5 [2, 4, 7, 3, 6, 8, 10] -> [2, 4, 5, 7, 3, 6, 8, 10]

```
inserta :: Ord a => a -> [a] -> [a]
inserta e [] = [e]
inserta e (x:xs)
   | e <= x = e:x:xs
   | otherwise = x : inserta e xs
```

Luego, se define la función *insertion_sort* que recibe como parámetro una lista:

> Insertion_sort [2, 4, 3, 6, 3] -> [2, 3, 3, 4, 6]

Se presentan diferentes soluciones:

Definición recursiva:

```
insertion_sort :: Ord a => [a] -> [a]
insertion_sort [] = []
insertion_sort (x:xs) = inserta x (insertion_sort xs)
```

Si la lista está vacía, no hay nada que ordenar. Si tiene elementos, se inserta el primer elemento en la lista ordenada que se construye de manera recursiva con el resto de la lista.

Definición por plegado por la derecha:

```
insertion_sort :: Ord a => [a] -> [a]
insertion_sort = foldr inserta []
```

la función *foldr* nos permite asociar a derecha una función *f*, recorriendo la estructura de manera recursiva. Al final, termina haciendo un trabajo similar al que se hace con la primera definición que se planteó del problema.

Un seguimiento nos permite entender de mejor manera su funcionamiento:

```
> Insertion_sort [2, 4, 3, 6, 3]
= foldr inserta [2, 4, 3, 6, 3]
= Inserta 2 (foldr inserta [4, 3, 6, 3])
= Inserta 2 (inserta 4 (foldr inserta [3, 6, 3]))
= Inserta 2 (inserta 4 (inserta 3 (foldr inserta [6, 3])))
= Inserta 2 (inserta 4 (inserta 3 (inserta 6 (foldr inserta [3]))))
= Inserta 2 (inserta 4 (inserta 3 (inserta 6 (inserta 3 (foldr inserta [])))))
= Inserta 2 (inserta 4 (inserta 3 (inserta 6 (inserta 3 []))))
= Inserta 2 (inserta 4 (inserta 3 (inserta 6 [3])))
= Inserta 2 (inserta 4 (inserta 3 [3, 6]))
= Inserta 2 (inserta 4 [3, 3, 6])
= Inserta 2 [3, 3, 4, 6]
= [2, 3, 3, 4, 6]
```

Definición por plegado por la izquierda:

```
insertion_sort :: Ord a => [a] -> [a]
insertion_sort = foldl (flip inserta) []
```

la función *foldl* hace lo mismo que *foldr*, pero trabaja con una asociación por izquierda. Mientras tanto, *flip* se encarga de aplicar la función *inserta*, pero con los parámetros que recibe en el orden inverso del que llegan.

Quick-sort – Ordenamiento de listas

Se implementa la función *quick_sort* que realiza el algoritmo de ordenamiento rápido sobre una lista de números:

```
> Quick_sort [2, 19, 4, 3, 1, 5, 3] -> [1, 2, 3, 3, 4, 5, 19]
```

```
quick_sort :: Ord a => [a] -> [a]
quick_sort [] = []
quick_sort (x:xs) = quick_sort menores ++ [x] ++ quick_sort mayores
      where menores = [e | e<-xs, e<x]
           mayores = [e | e<-xs, e>=x]
```

Si la lista está vacía, no hay elementos por ordenar. Si tiene elementos, se toma el primer elemento de la lista. A su izquierda se concatena con la lista ordenada de todos los números menores que él y a su derecha la lista ordenada de todos los números mayores que él.

Eliminar duplicados de una lista

Se define la función *duplicados* que permite evaluar si en una lista hay elementos duplicados:

```
> duplicados [1, 2, 3, 4, 5] -> False
> duplicados [1, 2, 3, 2] -> True
```

```
duplicados :: Eq a => [a] -> Bool
duplicados [] = False
duplicados (x:xs) = elem x xs || duplicados xs
```

Si la lista es vacía, no hay elementos duplicados. Si tiene elementos, se evalúa si el primer elemento de la lista pertenece al resto de la lista o si el resto de la lista tiene duplicados. Luego se define la función *elimina*, que nos permite eliminar elementos de una lista.

```
elimina :: Eq a => a -> [a] -> [a]
elimina x [] = []
elimina x (y:ys) | x == y = elimina x ys
             | otherwise = y : elimina x ys
```

Finalmente, se define la función *eliminaDuplicados*, que recibe como parámetro una lista.

```
eliminaDuplicados :: Eq a => [a] -> [a]
eliminaDuplicados [] = []
eliminaDuplicados (x:xs) = x : eliminaDuplicados (elimina x xs)
```

Se elimina el primer elemento de la lista, se le eliminan luego todos los duplicados de la lista resultante y luego se concatena de nuevo, al inicio de la lista, una única coincidencia del elemento que se eliminó inicialmente. Esto asegura la eliminación de todos los duplicados de la lista.

Criba de Eratóstenes

Se escriben todos los números desde el 2 hasta algún rango en el que se quieran calcular los números primos. Se hace de manera iterativa el siguiente proceso: Se toma el primer número de la lista, que es primo, y se eliminan de ella todos los múltiplos de ese número, haciendo así, que el primer número de la lista resultante en cada paso sea también un número primo. El método se repite hasta llegar al último elemento de la lista.

Se define la función *elimina* que recibe como parámetro un entero y una lista de enteros

```
elimina :: Int -> [Int] -> [Int]
elimina n xs = [ x | x <- xs, x `mod` n /= 0 ]
```

Esta función elimina de la lista a todos los elementos que pertenezcan a ella que sean múltiplos del número que se le pasa. Luego, se define la función *criba*

```
criba :: [Int] -> [Int]
criba [] = []
criba (n:ns) = n : criba (elimina n ns)
```

Esta función deja el primer elemento de la lista y lo concatena con la criba del resto de la lista.

Factorial

Se define la función *fact* que recibe un parámetro como entrada:

```
> fact 5 -> 120
```

Se presentan diferentes soluciones:

Definición recursiva:

```
fact1 :: Integer -> Integer
fact1 n = if n == 0
      then 1
      else n*fact1 (n-1)
```

Definición recursiva – modular:

```
fact2 :: Integer -> Integer
fact2 0 = 1
fact2 n = n* fact2 (n-1)
```

Definición con product:

```
fact3 :: Integer -> Integer
fact3 n = product [1..n]
```

El tercer método resulta más rápido por tratarse de una función pura de Haskell, estas tienen métodos de optimización que hacen más rápido el cálculo.

Comprobación de equivalencias

Una manera de comprobar si dos implementaciones son equivalentes en Haskell, es el uso de la función *quickCheck*, que nos permite evaluar una propiedad de equivalencia testeándola 100 veces con casos generados de manera aleatoria.

Para el ejemplo, evaluaremos la equivalencia entre los tres métodos de solución que implementamos para el factorial. Se define la función *prop_equivalencia*:

```
prop_equivalencia :: Integer -> Property
prop_equivalencia x =
   x >= 0 ==> (fact2 x == fact1 x &&
           fact3 x == fact1 x)
```

Comprobación:

```
Main> quickCheck prop_equivalencia
OK, passed 100 tests.
```

Derivada de una función

Se define una función *derivada* que calcula el valor de la derivada de una función f en un punto x con una aproximación de 0.0001.

```
derivada :: (Float -> Float) -> Float -> Float
derivada f x = (f(x+0.0001)-f(x))/0.0001
```

Ahora, a partir de la derivada, definiremos una función *puntoCero* que sea capaz de encontrar un cero de la función *f*. Hacemos uso de la siguiente propiedad:

> Si b es una aproximación del punto cero de f, entonces b – f(b)/f'(b) es una mejor aproximación

```
puntoCero f = until aceptable mejorar 1
   where mejorar b = b - f b / derivada f b
      aceptable b = abs (f b) < 0.00001
```

Se llama a la función de la siguiente manera:

```
> puntoCero cos -> 1.570796
```

Utilizando la función de *puntoCero*, se construye una función *inversa* que permita determinar el valor de un número evaluado en la función inversa de *f*.

```
inversa g a = puntoCero f
   where f x = g x – a
```

Se llama a la función de la siguiente manera:

> > Inversa (^2) 3 -> 1.7320508

> > Inversa sqrt 3 -> 8.990

Árboles binarios

Se define un árbol binario *a* como: Una hoja *a* o Un nodo *a* que tiene como hijos dos árboles binarios *a*. La siguiente es la definición en Haskell:

```
data Árbol a = Hoja
   | Nodo a (Árbol a) (Árbol a)
   deriving Show
```

Para trabajar, utilizaremos el siguiente ejemplo de árbol binario:

```
ejÁrbol_1 = Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
                           (Nodo 3 Hoja Hoja))
                   (Nodo 6 (Nodo 5 Hoja Hoja)
                           (Nodo 7 Hoja Hoja))
```

Se define la función tamaño que calcula el número de nodos internos de un árbol.

```
tamaño :: Árbol a -> Int
tamaño Hoja = 0
tamaño (Nodo x a1 a2) = 1 + tamaño a1 + tamaño a2
```

Árbol de búsqueda binaria

Un árbol de búsqueda binaria es aquel que cumple con la condición de que todos los valores en el subárbol izquierdo de un nodo son menores al valor de este nodo, y los valores en el subárbol derecho son mayores.

Para determinar si un elemento pertenece a un árbol de búsqueda binaria, se define la función *elemÁrbol*:

> > elemÁrbol 5 ejÁrbol_1 -> True

> > elemÁrbol 9 ejÁrbol_1 -> False

```
elemÁrbol :: Ord a => a -> Árbol a -> Bool
elemÁrbol e Hoja = False
elemÁrbol e (Nodo x izq der) | e==x = True
                             | e<x = elemÁrbol e izq
                             | e>x = elemÁrbol e der
```

Ahora, se define la función insertaÁrbol, que permite insertar un elemento en el árbol de búsqueda binaria.

```
> insertaÁrbol 8 ejÁrbol_1
= Nodo 4 (Nodo 2 (Nodo 1 Hoja Hoja)
                 (Nodo 3 Hoja Hoja))
         (Nodo 6 (Nodo 5 Hoja Hoja)
                 (Nodo 7 Hoja
                         (Nodo 8 Hoja Hoja)))
```

```
insertaÁrbol :: Ord a => a -> Árbol a -> Árbol a
insertaÁrbol e Hoja = Nodo e Hoja Hoja
insertaÁrbol e (Nodo x izq der)
    | e <= x = Nodo x (insertaÁrbol e izq) der
    | e > x = Nodo x izq (insertaÁrbol e der)
```

Ordenamiento mediante un árbol de búsqueda binaria

Primero, se define la función *listaÁrbol* que convierte una lista a un árbol de búsqueda binario.

```
> listaÁrbol [3, 2, 4, 1]
= Nodo 1
      Hoja
      (Nodo 4
           (Nodo 2
                Hoja
                (Nodo 3 Hoja Hoja))
           Hoja)
```

```
listaÁrbol :: Ord a => [a] -> Árbol a
listaÁrbol = foldr insertaÁrbol Hoja
```

Luego, se define una función *aplana* que devuelva una lista a partir de un árbol de búsqueda binario:

```
> aplana ejÁrbol_1 -> [1, 2, 3, 4, 5, 6, 7]
```

```
aplana :: Árbol a -> [a]
aplana Hoja = []
aplana (Nodo x izq der) = aplana izq ++ [x] ++ aplana der
```

Finalmente, se define la función *tree_sort* que combine ambas funciones, *listaÁrbol* y *aplana*:

```
tree_sort :: Ord a => [a] -> [a]
tree_sort = aplana . listaÁrbol
```

```
> tree_sort [1, 4, 3, 7, 2] -> [1, 2, 3, 4, 7]
```

Referencias:

[1] J. A. Alonzo Jiménez, *Ejercicios de programación funcional con Haskell*, Sevilla, Universidad de Sevilla, 2007.