Práctica Nº 1 - Programación Funcional

Para resolver esta práctica, recomendamos usar el "Hugs 98", de distribución gratuita, que puede bajarse de `http://www.haskell.org/hugs/`.

Los ejercicios marcados con el símbolo ★ constituyen un subconjunto mínimo de ejercitación. Sin embargo, aconsejamos fuertemente hacer todos los ejercicios.

Definición de Tipos y Currificación

Ejercicio 1 ★

Dado el siguiente programa:

```
xs = [1,2,3]::[Float]
ys = map (+) xs
```

¿Cuál es el tipo de ys? (La idea acá es que lo calculen "a ojo".)

Ejercicio 2 ★

I. Reescribir la expresión:

```
map f (map g xs)
```

para que se utilice un sólo llamado a la función `map` (se necesitará la función de composición, `(.)`).

II. Redifinir la siguiente función:

```
f xs = map (\x -> (x+1)/2) xs
```

de modo que utilice composición de operadores de manera similar al ítem anterior.

Ejercicio 3 ★

I. Definir la función `curry`, que dada una función de dos argumentos, devuelve su equivalente currificada.

II. Definir la función `uncurry`, que dada una función currificada de dos argumentos, devuelve su versión no currificada equivalente. Es la inversa de la anterior.

III. ¿Se podría definir una función `curryN`, que tome una función de un número arbitrario de argumentos y devuelva su versión currificada?

Listas por Comprensión

Ejercicio 4

¿Cuál es el valor de esta expresión?

```
[ x | x <- [1..4], y <- [x..5], (x+y) `mod` 2 == 0 ]
```

Ejercicio 5 ★

Una tripla pitagórica es una tripla (a,b,c) de enteros positivos tal que $a^2 + b^2 = c^2$.

La siguiente es una definición de una lista (infinita) de triplas pitagóricas:

```
pitagorica :: [(Integer,Integer,Integer)]
pitagorica = [(a,b,c) | a <- [1..], b <-[1..], c <- [1..], a^2 + b^2 == c^2]
```

Explicar porqué esta definición no es muy útil. Dar una definición mejor.

Ejercicio 6 ★

Redefinir la siguiente función `g`, sin utilizar listas por comprensión ni recursión explícita, y usando un subconjunto de las siguientes funciones: `map`, `filter` y `foldr`. Además, pueden utilizarse las funciones `even`, `odd`, `length` , `reverse` y `(++)`.

```
g xs = [ y ++ reverse x | x <- xs, odd (length x), y <- xs, even (length y)]
```

Modos de Evaluación

Salvo que se indique explícitamente, suponer que el modo de evaluación es *lazy*, que es el modo que utiliza Haskell.

Ejercicio 7 ★

Se tienen definidas las siguientes funciones:

```
cuadrado x = x*x

ciclo = [1, 2] ++ ciclo

or True x = True
or False x = x

orNoCurry (True,True) = True
orNoCurry (True,False) = True
orNoCurry (False,True) = True
orNoCurry (False,False) = False

orNoCurry2 (True,x) = True
orNoCurry2 (False,x) = x
```

Evaluar las siguientes expresiones (usando las definiciones correspondientes) utilizando evaluación *eager* por un lado y evaluación *lazy* por otro, explicando los criterios y anomalías presentes en cada caso. Las funciones `head`, `tail` y `elem` son las definidas en el prelude de Haskell.

```
  i)  elem 4 [4, 5, 2]
 ii)  cuadrado (4+5*9)
iii)  ciclo
 iv)  take 5 ciclo
  v)  foldr (+) 0 [5, 6/0, 8]
 vi)  foldr (curry orNoCurry) True [not False, head (tail [True])]
vii)  foldr or True [not False, head (tail [True])]
viii)  foldr (curry orNoCurry2) True [head (tail [True]), not False]
```

Ejercicio 8

Generar la lista de los primeros mil números perfectos. Un número natural `n` es perfecto sii la suma de sus divisores menores estrictos que él es igual a `n`. Ver de qué forma se puede implementar usando evaluación *lazy* y funciones de orden superior.

Alto Orden y Esquemas de recursión

Ejercicio 9 ★

I. Definir la función `genLista`, que genera una lista de una cantidad dada de elementos, a partir de un elemento inicial y de una función de incremento entre los elementos de la lista. Dicha función de incremento, dado un elemento de la lista, devuelve el elemento siguiente.

II. Usando `genLista`, definir la función `dh`, que dado un par de números (el primero menor que el segundo), devuelve una lista de números consecutivos desde el primero hasta el segundo.

Ejercicio 10 ★

I. Definir y dar el tipo del esquema de recursión `foldNat` sobre los naturales. Utilizar el tipo `Integer` de Haskell.

II. Utilizando `foldNat`, definir la función `potencia`.

Ejercicio 11

I. Definir la función `paraCada`, que recibe dos números (un número inicial `i` y otro final `f`), un valor `v` y una función `g` que dado un valor y un número devuelve un valor. Dados esos parámetros, `paraCada` devuelve la aplicación sucesiva de `g` desde el número final hasta el inicial: en cada paso, `g` se aplica al resultado de la evaluación anterior y al predecesor del número anterior. Esto se realiza hasta llegar a `i`. Inicialmente, `g` se aplica a `v` y a `f`.

   Ejemplo: `paraCada 1 3 [] (flip (:))` devuelve `[1, 2, 3]`

II. Definir la función `todos`, que recibe una lista de elementos y una condición sobre los elementos, y determina si todos los elementos de la lista cumplen con dicha condición. Usar `paraCada`, `length` y `(!!)`.

III. Definir la función `ninguno`, que recibe una lista de elementos y una condición sobre los elementos, y determina si ninguno de los elementos de la lista cumple con dicha condición. Usar `todos`.

Ejercicio 12 ★

I. Definir la función `mapo`, una versión de `map` que toma una función de dos argumentos y una lista de pares de valores, y devuelve la lista de aplicaciones de la función a cada par.

II. Definir la función `mapo2`, una versión de `mapo` que toma una función currificada de dos argumentos y dos listas (de igual longitud), y devuelve una lista de aplicaciones de la función a cada elemento correspondiente a las dos listas. Esta función en Haskell se llama `zipWith`.

Ejercicio 13

I. Definir la función `paresConsec`, que dada una lista de elementos devuelve una lista de pares de elementos, formada por todos los elementos de la lista original junto a su inmediato sucesor.

   Ejemplo: `paresConsec [7,3,2,5]` → `[(7,3),(3,2),(2,5)]`

II. Definir la función `pascal`, que devuelve en forma de listas el triángulo de Pascal hasta la altura pedida. No se permite el uso de números combinatorios. Usar `last`, `paresConsec` y `map`.

   Ejemplo: `pascal 4` → `[ [1], [1,1], [1,2,1], [1,3,3,1], [1,4,6,4,1] ]`

Ejercicio 14 ★

I. Usando `map` y la función `p x y = y x`, definir una función `ap` que cumpla con lo siguiente, e indicar su tipo:

   `ap [f1, f2, ..., fn] x` → `[f1 x, f2 x, ..., fn x]`

II. Definir una función `apl` que cumpla con lo siguiente, e indicar su tipo:

```
apl [f1, f2, ..., fn] [x1, x2, ..., xm] →
    [[f1 x1, f2 x1, ..., fn x1], ..., [f1 xm, f2 xm, ..., fn xm]]
```

Ejercicio 15 ★

I. Definir usando `foldr` las funciones `suma`, `elem`, `append`, `filter` y `map`.

II. Definir la función `sumaAlt`, que realiza la suma alternada de los elementos de una lista. Es decir, da como resultado: el primer elemento, menos el segundo, más el tercero, menos el cuarto, etc. Usar `foldl`.

Ejercicio 16

Definir la función `esCerrada`, que recibe una lista y una función `f`. Dicha función `f` toma un par de elementos del tipo de la lista y devuelve un elemento del mismo tipo. A partir de esos parámetros, `esCerrada` determina si `f` es cerrada con respecto a la lista de entrada. Es decir, tomando como dominio de `f` a cada par posible de elementos de la lista, `esCerrada` dice si la imagen de `f` está incluida en dicho dominio.

Ejercicio 17

I. Definir la función `partes`, que recibe una lista `L` y devuelve la lista de todas las listas formadas por los mismos elementos de `L`, en su mismo orden de aparición.

   Ejemplo: `partes [5,1,2]` → `[[], [5], [1], [2], [5,1], [5,2], [1,2], [5,1,2]]`
   (no necesariamente en ese orden).

II. Definir la función `prefijos`, que dada una lista, devuelve todos sus prefijos.

   Ejemplo: `prefijos [5,1,2]` → `[[], [5], [5,1], [5,1,2]]`

III. Definir la función `sublistas`, que dada una lista, devuelve todas sus sublistas (listas de elementos consecutivos que conforman la lista original).

   Ejemplo: `sublistas [5,1,2]` → `[[], [5], [1], [2], [5,1], [1,2], [5,1,2]]`
   (no necesariamente en ese orden).

Ejercicio 18

I. Definir la función `sacarUna::Eq a=>a->[a]->[a]`, que dados un elemento y una lista devuelve el resultado de eliminar de la lista la primera aparición del elemento (si está presente). Sugerencia: usar `break` o `recr`.

II. Utilizar listas por comprensión para definir la función `perms`, que dada una lista, devuelve todas sus permutaciones. Para esto se permite usar recursión explícita.

Ejercicio 19 ★

Definir el tipo de datos `ArbolNV` de árboles no vacíos, donde cada nodo tiene una cantidad indeterminada de hijos, las hojas contienen rótulos de un tipo y los nodos intermedios contienen rótulos de eventualmente otro tipo.

Ejercicio 20 ★

I. Definir la función:

   ```
   foldalt :: (a -> b -> b) -> (a -> b -> b) -> b -> [a] -> b
   ```

   Esta función es una versión modificada de `foldr`, que realiza un fold sobre la lista de entrada pero aplicando una función `f` a los elementos en posiciones pares y una función `g` a los elementos en posiciones impares. Considerar que la primera posición de la lista es la número 1.

II. Usando `foldalt`, escriba la función `sumaaltdoble :: [Int] -> Int`, que calcula la suma de los números de las posiciones impares y el doble de los números de las posiciones pares.
   Ejemplo: `sumaaltdoble [2,5,3,7,7,3] = 2+10+3+14+7+6 = 42`

III. Usando `foldalt`, escriba la función `numsalt :: [[Int]] -> Int` que calcula el producto de la longitud de las listas de las posiciones pares y la suma de los elementos de las listas de las posiciones impares.
   Ejemplo: `numsalt [ [1,2], [1,2], [2,3], [2,3] ] = 2*(3*(2*5)) = 60`

Ejercicio 21

I. Escribir la función `sumaMat`, que representa la suma de matrices, usando `zipWith`. Representaremos una matriz como la lista de sus filas. Esto quiere decir que cada matriz será una lista finita de listas finitas, todas de la misma longitud, con elementos enteros. Recordamos que la suma de matrices se define como la suma celda a celda. Asumir que las dos matrices a sumar están bien formadas y tienen las mismas dimensiones.

   ```
   sumaMat :: [[Int]] -> [[Int]] -> [[Int]]
   ```

II. Escribir la función `trasponer`, que dada una matriz como las del ítem I, devuelve su traspuesta; es decir, devuelve en la posición i, j del resultado el contenido de la posición j, i de la matriz original. Notar que si la entrada es una lista de N listas, todas de longitud M, entonces el resultado debe tener M listas, todas de longitud N.

   ```
   trasponer :: [[Int]] -> [[Int]]
   ```

III. Escribir la función `zipWithList`, que dada una lista de listas, un caso base y una función combinadora, hace un zipWith de los elementos correspondientes de todas las listas. Es decir:

```
zipWithList :: (a -> b -> b) -> b -> [[a]] -> [b]

zipWithList func base [[a1,...,an],[b1,...,bn],...,[z1,...,zn]] reduce a
  [foldr func base [a1,...,z1], foldr func base [a2,...,z2], ...,
    foldr func base [an,...,zn]]
```

Se puede asumir que ninguna de las listas es infinita.

Nota: se puede hacer el ítem II usando la función del ítem III o viceversa (si `trasponer` es lo suficientemente general para aplicarse a listas de listas arbitrarias). Obviamente, no sería correcto hacer las dos cosas a la vez.

Ejercicio 22 ★

Consideremos el siguiente tipo de datos:

```
data AHD tInterno tHoja = Hoja tHoja
                        | Rama tInterno (AHD tInterno tHoja)
                        | Bin (AHD tInterno tHoja) tInterno (AHD tInterno tHoja)
```

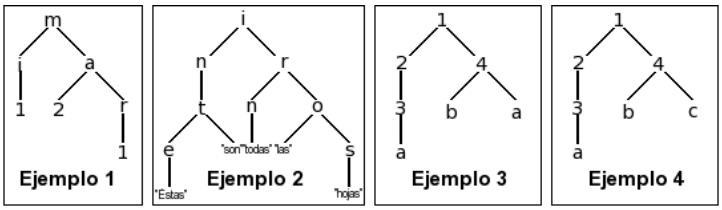
que representa un árbol binario no vacío cuyos nodos internos pueden tener datos de un tipo diferente al de sus hojas. (AHD = árbol con hojas distinguidas).

Por ejemplo:

`Bin (Hoja ‘‘hola’’) ‘b’ (Rama ‘c’ (Hoja ‘‘chau’’))` tiene tipo `AHD Char String`

`Rama 1 (Bin(Hoja True)(-2)(Hoja False))` tiene tipo `AHD Int Bool`

A continuación mostramos algunos ejemplos de forma más gráfica:



I. Escribir el esquema de recursión estructural (`fold`) para este tipo de datos.

```
foldAHD :: (tHoja->b)->(tInterno->b->b)->(b->tInterno->b->b)->AHD tInterno tHoja->b
```

Para este inciso está permitido utilizar recursión explícita.

II. Escribir, usando `foldAHD`, la función `mapAHD::(a->b)->(c->d)->AHD a c->AHD b d`, que actúa de manera análoga al `map` de listas, aplicando la primera función a los nodos internos y la segunda a las hojas. Por ejemplo:

```
mapAHD (+1) not (Bin(Rama 1 (Hoja False))2(Bin(Hoja False)3(Rama 5 (Hoja True))))
```

devuelve `Bin (Rama 2 (Hoja True)) 3 (Bin (Hoja True) 4 (Rama 6 (Hoja False)))`.

III. Escribir una función que analice un `AHD` de la siguiente manera: si el árbol posee al menos una hoja repetida, se debe devolver una función que, dado un elemento del tipo de las hojas, indique su cantidad de apariciones como hoja del árbol. En caso contrario (si no hay hojas repetidas), se debe devolver el recorrido DFS de los nodos internos del árbol. (Recordar que el recorrido DFS comienza por la raíz y explora cada rama en profundidad, pasando una sola vez por cada nodo).

Para esto utilizaremos el tipo `Either` del Prelude. De esta manera, el tipo de la función pedida es:

```
analizar::Eq tHoja=>AHD tInterno tHoja->Either (tHoja->Int) [tInterno]
```

Ejemplos (para los árboles dibujados arriba):

`analizar ejemplo2` devuelve `Right "internos"`.

`analizar ejemplo3` devulelve `Left repeticionesEj3`, donde `repeticionesEj3` es una función con el siguiente comportamiento:

repeticionesEj3 ‘a’ = 2
repeticionesEj3 ‘b’ = 1
repeticionesEj3 x = 0 *para todo* carácter x *distinto de* ‘a’ *y* ‘b’.

`analizar ejemplo4` devuelve `Right [1,2,3,4]`

Ejercicio 23

Definimos el siguiente tipo:

```
data Agenda p t = Vacía | Teléfonos p [t] (Agenda p t)
```

Este tipo modela una agenda de teléfonos. A una agenda se le puede agregar una nueva entrada, donde se registra para una persona una lista de teléfonos. Una misma persona puede aparecer en varias entradas. La lista de teléfonos de una entrada puede contener repetidos.

Ejemplo:

```
miAgenda =  Teléfonos "Letincho" [42079999,43834567]
            (Teléfonos "Javi" [47779830] (Teléfonos "Letincho" [42079999] Vacía))
```

Sea `foldrAgenda` el siguiente esquema de recursión genérico para agendas:

```
foldrAgenda f b Vacía = b
foldrAgenda f b (Teléfonos p ts ag) = f p ts (foldrAgenda f b ag)
```

En la resolución de este ejercicio, no usar recursión explícita.

I. Decir cuál es el tipo de la función `foldrAgenda`.

II. Definir una función `dameTeléfonosDe::Eq p =>p ->Agenda p t ->[t]`, que devuelva todos los teléfonos que aparecen en la agenda para una misma persona.

Ejemplo:

`dameTeléfonosDe "Letincho" miAgenda`

devuelve `[42079999, 42079999, 43834567]`

III. Definir el esquema de recursión:

`foldrPersona::Eq p =>(p ->c ->c) ->c ->Agenda p t ->c`

que funcione de manera similar a `foldrAgenda` pero sólo aplique la función pasada como parámetro a las personas de la agenda. Si la agenda tiene personas repetidas, sólo debe aplicar la función a la aparición correspondiente al redex más interno de la persona en la agenda.

Ejemplo:

`foldrPersona f b miAgenda`

debe aplicar la función `f` primero a "`Letincho`" con el caso base pasado como parámetro y luego aplicarla a "`Javi`" con el valor obtenido en el paso anterior.

IV. Definir una función

`personasSinRepeticiones:: Eq p => Agenda p t -> [p]`

que devuelva la lista de personas que aparecen en la agenda, sin repeticiones.

Ejemplo:

`personasSinRepeticiones miAgenda`

devuelve `["Javi","Letincho"]`

V. Definir una función

`compactar::Eq p => Agenda p t -> Agenda p t`

que a partir de una agenda, devuelva otra donde aparecen todas las personas de la primer agenda, pero sin repeticiones. Cada entrada de la agenda resultado, debe contener todos los teléfonos que la persona tenía en la agenda original.

Ejemplo:

`compactar miAgenda`

devuelve

`Teléfonos "Javi"[47779830] (Teléfonos "Letincho"[42079999,42079999,43834567] Vacía)`