Práctica de ejercicios #6 - Esquemas de recursión sobre otros tipos recursivos

Programación Funcional, Universidad Nacional de Hurlingham

10 de mayo de 2025

Aclaraciones:

- ***Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados.*** *No se saltee ejercicios sin consultar antes a un docente.*
- ***Recuerde que puede aprovechar en todo momento las funciones que ha definido****, tanto las de esta misma práctica como las de prácticas anteriores.*
- ***Pruebe todas sus implementaciones****, al menos en una consola interactiva.*
- *Es sumamente aconsejable* ***resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en videos publicados o clases presenciales****, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*

ACLARACIÓN GENERAL: Cuando se pide *"definir sin utilizar recursión en forma explícita"*, quiere decir que solamente se puede utilizar la versión de **`fold`** o **`rec`** que corresponda al tipo sobre el que se trabaja en conjunto con algunos combinadores (como por ejemplo **`flip`**). En particular, las definiciones NO deben tomar el valor del tipo recursivo de forma explícita (o sea, la función que transforma el argumento en el resultado DEBE ser el resultado de un **`fold`** con solamente los argumentos que corresponden a los casos de análisis, quizás combinada con otras funciones).

PARTE 1

Considerar la definición de **`ExpA`**:

```
data ExpA = Cte Int
          | Suma ExpA ExpA
          | Prod ExpA ExpA
```

Ejercicio 1) Dar el tipo y definir **`foldExpA`**, que expresa el esquema de recursión estructural para la estructura **`ExpA`**.

Ejercicio 2) Resolver las siguientes funciones sin utilizar recursión explícita:

a. **`cantidadDeCeros :: ExpA -> Int`**, que describe la cantidad de ceros explícitos en la expresión dada.
   OBSERVACIÓN: Una cuenta cuyo resultado dé cero, NO es un cero explícito.
b. **`noTieneNegativosExplicitosExpA :: ExpA -> Bool`**, que describe si la expresión dada no tiene números negativos de manera explícita.
c. **`simplificarExpA :: ExpA -> ExpA`**, que describe una expresión con el mismo significado que la dada, pero que no tiene sumas del número 0 ni multiplicaciones por 1 o por 0. La resolución debe ser exclusivamente *simbólica*.

d. **`evalExpA :: ExpA -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
e. **`showExpA :: ExpA -> String`**, que describe el string sin espacios y con paréntesis correspondiente a la expresión dada.
RECORDATORIO: los String son listas de caracteres.
AYUDA: puede utilizarse la función **`show :: Int -> String`** para construir el string que corresponde a un número.

Ejercicio 3) Dar el tipo y definir **`recExpA`**, que expresa el esquema de recursión primitiva para la estructura **`ExpA`**.

Ejercicio 4) Resolver las siguientes funciones sin utilizar recursión explícita:

a. **`cantDeSumaCeros :: ExpA -> Int`**, que describe la cantidad de constructores de suma con al menos uno de sus hijos constante cero.
b. **`cantDeProdUnos :: ExpA -> Int`**, que describe la cantidad de constructores de producto con al menos uno de sus hijos constante uno.

PARTE 2

Considerar la definición de **`EA`**:

```
data EA = Const Int | BOp BinOp EA EA
data BinOp = Sum | Mul
```

Ejercicio 5) Dar el tipo y definir **`foldEA`**, que expresa el esquema de recursión estructural para la estructura **`EA`**.

Ejercicio 6) Resolver las siguientes funciones sin utilizar recursión explícita:

a. **`noTieneNegativosExplicitosEA :: EA -> Bool`**, que describe si la expresión dada no tiene números negativos de manera explícita.
b. **`simplificarEA :: EA -> EA`**, que describe una expresión con el mismo significado que la dada, pero que no tiene sumas del número 0 ni multiplicaciones por 1 o por 0. La resolución debe ser exclusivamente *simbólica*.
c. **`evalEA :: EA -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
d. **`showEA :: EA -> String`**, que describe el string sin espacios y con paréntesis correspondiente a la expresión dada.
e. **`ea2ExpA :: EA -> ExpA`**, que describe una expresión aritmética representada con el tipo ExpA, cuyo significado es el mismo que la dada.
f. **`ea2Arbol :: EA -> ABTree BinOp Int`**, que describe la representación como elemento del tipo **`ABTree BinOp Int`** de la expresión aritmética dada.

Ejercicio 7) Dar el tipo y definir **`recEA`**, que expresa el esquema de recursión primitiva para la estructura **`EA`**.

Ejercicio 8) Resolver las siguientes funciones sin utilizar recursión explícita:

a. **`cantDeSumaCeros :: EA -> Int`**, que describe la cantidad de constructores de suma con al menos uno de sus hijos constante cero.
b. **`cantDeProdUnos :: ExpA -> Int`**, que describe la cantidad de constructores de producto con al menos uno de sus hijos constante uno.

PARTE 3

Considerar la definición de **`Tree`**:

```
data Tree a = ET | NT a (Tree a) (Tree a)
```

Para los ejemplos, suponer también la siguiente definición

```
tej = NT 1 (NT 2 (NT 4 (NT 8 ET ET)
                       (NT 9 (NT 18 ET ET)
                             ET))
                 (NT 5 (NT 10 ET ET)
                       ET)))
           (NT 3 (NT 6 ET
                       (NT 13 ET ET))
                 (NT 7 (NT 14 ET ET)
                       (NT 15 ET ET)))
```

Ejercicio 9) Dar el tipo y definir la función **`foldT`**, que expresa el esquema de recursión estructural para la estructura **`Tree`**.

Ejercicio 10) Definir las siguientes funciones sin utilizar recursión explícita:

a. **`mapT :: (a -> b) -> Tree a -> Tree b`**, que describe el árbol con la misma estructura que el dado, pero con cada elemento modificado con la función dada.
b. **`sumT :: Tree Int -> Int`**, que describe la suma de todos los elementos del árbol.
c. **`sizeT :: Tree a -> Int`**, que describe la cantidad de elementos del árbol.
d. **`heightT :: Tree a -> Int`**, que describe la altura del árbol
e. **`preOrder :: Tree a -> [a]`**, que describe la lista de elementos del árbol dado, en preorder (o sea, la raíz antes que los elementos de los hijos).
f. **`inOrder :: Tree a -> [a]`**, que describe la lista de elementos del árbol dado, en inorder (o sea, la raíz entre los elementos del hijo izquierdo y los elementos del hijo derecho).
g. **`postOrder :: Tree a -> [a]`**, que describe la lista de elementos del árbol dado, en postorder (o sea, la raíz después que los elementos de los hijos).
h. **`mirrorT :: Tree a -> Tree a`**, que describe el árbol que es la versión espejo del árbol dado (todo lo que estaba a la izquierda, está a la derecha, y viceversa).
i. **`countByT :: (a -> Bool) -> Tree a -> Int`**, que describe la cantidad de elementos del árbol dado que cumplen con el predicado dado.
j. **`partitionT :: (a -> Bool) -> Tree a -> ([a], [a])`**, que describe un par de listas tal que la primera contiene todos los elementos que cumplen el predicado en preorden, y la 2da los elementos que no lo cumplen, también en preorden.

k. **`zipWithT :: (a->b->c) -> Tree a -> Tree b -> Tree c`**, que describe el árbol que resulta de combinar los elementos que están en posiciones correspondientes de los árboles dados utilizando la función dada. Los elementos que no tienen correspondiente se pierden.

l. **`caminoMasLargo :: Tree a -> [a]`**, que describe la lista de elementos del árbol dado que conduce desde la raíz hasta la hoja más profunda. Por ejemplo

```
caminoMasLargo tej = [1,2,4,9,18]
```

m. **`todosLosCaminos :: Tree a -> [[a]]`**, que describe la lista de todos los caminos del árbol dado que van desde la raíz hasta cada uno de los elementos. Por ejemplo

```
todosLosCaminos tej =
  [ [1], [1,2], [1,2,4], [1,2,4,8], [1,2,4,9], [1,2,4,9,18],
    [1,2,5], [1,2,5,10], [1,3], [1,3,6], [1,3,6,13],
    [1,3,7], [1,3,7,14], [1,3,7,15]
  ]
```

n. **`todosLosNiveles :: Tree a -> [[a]]`**, que describe la lista de todos los niveles del árbol dado, donde un nivel son todos los elementos a una profundidad dada. Por ejemplo

```
todosLosNiveles tej =
  [ [1], [2,3], [4,5,6,7], [8,9,10,13,14,15], [18] ]
```

o. **`nivelN :: Tree a -> Int -> [a]`**, que describe la lista de elementos del árbol dado que están en el nivel del número dado. Por ejemplo:

```
nivelN tej 3 = [4,5,6,7]
```

Ejercicio 11) Dar el tipo y definir la función **`recT`**, que expresa el esquema de recursión primitiva para la estructura **`Tree`**.

Ejercicio 12) Definir las siguientes funciones sin utilizar recursión explícita:

a. **`insertT :: a -> Tree a -> Tree a`**, que describe el árbol resultante de insertar el elemento dado en el árbol dado, teniendo en cuenta invariantes de BST.

b. **`caminoHasta :: Eq a => a -> Tree a -> [a]`**, que describe el camino hasta el elemento dado en el árbol dado, suponiendo que existe el elemento en el árbol (falla si el elemento no existe). Por ejemplo:

```
caminoHasta 13 tej = [1,3,6,13]
caminoHasta 11 tej = error "No existe ese elemento"
```

SUGERENCIA: definir una función auxiliar que indique si el elemento existe o no, y en caso que exista, describa la lista dada, y usarla para definir la función pedida.

PARTE 4

Considerar la definición de **`Dep`**:

```
data Dep a = SinSalida
           | Almacen [a]
           | Siga (Dep a)
           | Bif (Dep a) (Dep a)
           | Hub (Dep a) (Dep a) (Dep a) (Dep a)
```

y del siguiente tipo para indicar direcciones dentro de un depósito:

```
type Guia = [Indicacion]
data Indicacion = Seguir | IrAIzq | IrADer | UsarPuerta Int
```

Para los ejemplos, suponer también la siguiente definición

```
dej = Hub (Bif  (Siga (Almacen [2,3]))
                (Bif  (Siga SinSalida)
                      (Almacen [17])))
          (Siga (Siga (Siga SinSalida)))
          (Siga (Bif  SinSalida
                      (Almacen [99,8])))
          (Bif  (Hub  SinSalida
                      (Almacen [42])
                      (Siga SinSalida)
                      (Siga (Almacen [])))
                (Siga SinSalida))
```

Ejercicio 13) Dar el tipo y definir la función **`foldD`**, que expresa el esquema de recursión estructural para la estructura **`Dep`**.

Ejercicio 14) Definir las siguientes funciones sin utilizar recursión explícita:

a. **`saquearAlmacenes :: Dep a -> [a]`**, que describe la lista de todos los elementos que se encuentren en todos los almacenes del depósito.
b. **`contarHubs :: Dep a -> Int`**, que describe la cantidad de hubs que tiene el depósito.
c. **`caminosSinSalida :: Dep a -> [Guia]`**, que describe la lista de todas las guías que llevan a lugares dentro del depósito que son caminos sin salida. Por ejemplo:

```
caminosSinSalida dej =
  [ [UsarPuerta 1, IrADer, IrAIzq, Seguir],
    [UsarPuerta 2, Seguir, Seguir, Seguir],
    [UsarPuerta 3, Seguir, IrAIzq],
    [UsarPuerta 4, IrAIzq, UsarPuerta 1],
    [UsarPuerta 4, IrADer, Seguir]
  ]
```

d. **`caminosAAlmacenes :: Dep a -> [Guia]`**, que describe la lista de todas las guías que llevan a lugares dentro del depósito que son almacenes. Por ejemplo:

```
caminosAAlmacenes dej =
  [ [UsarPuerta 1, IrAIzq, Seguir],
    [UsarPuerta 1, IrADer, IrADer],
    [UsarPuerta 3, Seguir, IrADer],
    [UsarPuerta 4, IrAIzq, UsarPuerta 2],
    [UsarPuerta 4, IrAIzq, UsarPuerta 4, Seguir]
  ]
```

Ejercicio 15) Dar el tipo y definir la función **`recD`**, que expresa el esquema de recursión primitiva para la estructura **`Dep`**.

Ejercicio 16) Definir las siguientes funciones sin utilizar recursión explícita:

a. **`caminosHasta :: (Dep a -> Bool) -> Dep a -> [Guia]`**, que describe la lista de todas las guías que llevan a lugares dentro del depósito dado que cumplen con el predicado dado. Por ejemplo:

```
caminosHasta esSinSalida dej =
  [ [UsarPuerta 1, IrADer, IrAIzq, Seguir],
    [UsarPuerta 2, Seguir, Seguir, Seguir],
    [UsarPuerta 3, Seguir, IrAIzq],
    [UsarPuerta 4, IrAIzq, UsarPuerta 1],
    [UsarPuerta 4, IrADer, Seguir]
  ]
caminosHasta esAlmacen dej =
  [ [UsarPuerta 1, IrAIzq, Seguir],
    [UsarPuerta 1, IrADer, IrADer],
    [UsarPuerta 3, Seguir, IrADer],
    [UsarPuerta 4, IrAIzq, UsarPuerta 2],
    [UsarPuerta 4, IrAIzq, UsarPuerta 4, Seguir]
  ]
```

Ejercicio 1)

```
foldExpA :: (Int -> b)
            -> (b -> b -> b)
            -> (b -> b -> b)
            -> ExpA -> b
foldExpA c s p (Cte n) = c n
foldExpA c s p (Suma e1 e2) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
foldExpA c s p (Prod e1 e2) = p (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

Ejercicio 2)

```
cantidadDeCeros = foldExpA (\n -> delta (n==0)) (+) (+)

noTieneNegativosExplicitosExpA = foldExpA (\n -> n>=0) (&&) (&&)

simplificarExpA = foldExpA Cte armarSuma armarProd

armarSuma (Cte 0) e2 = e2
armarSuma e1 (Cte 0) = e1
armarSuma e1 e2      = Suma e1 e2

armarProd (Cte 1) e2 = e2
armarProd e1 (Cte 1) = e1
armarProd e1 e2      = Prod e1 e2

evalExpA = foldExpA id (+) (*)

showExpA = foldExpA (\n -> "Cte " ++ show n)
                    (\s1 s2 -> "Suma (" ++ s1 ++ ") (" ++ s2 ++ ")")
                    (\s1 s2 -> "Prod (" ++ s1 ++ ") (" ++ s2 ++ ")")
```

Ejercicio 3)

```
recExpA :: (Int -> b)
          -> (b -> ExpA -> b -> ExpA ->b)
          -> (b -> ExpA -> b -> ExpA ->b)
          -> ExpA -> b
recExpA c s p (Cte n) = c n
recExpA c s p (Suma e1 e2) = s (recExpA  c s p e1)
                               (recExpA  c s p e2)
recExpA c s p (Prod e1 e2) = p (recExpA  c s p e1)
                               (recExpA  c s p e2)
```

Ejercicio 4)

```
cantDeSumaCeros =
   recExpA (\n -> 0)
```

```
           (\n1 e1 n2 e2 -> delta (esCte 0 e1
                                  || esCte 0 e2) + n1 + n2)
           (\n1 e1 n2 e2 -> n1 + n2)

esCte :: Int -> ExpA -> Bool
esCte n (Cte n') = n==n'
esCte _ _        = False

cantDeProdUnos =
   recExpA (\n -> 0)
           (\n1 e1 n2 e2 -> n1 + n2)
           (\n1 e1 n2 e2 -> delta (esCte 1 e1
                                  || esCte 1 e2) + n1 + n2)
```

Ejercicio 5)

```
foldEA :: (Int -> b)
             -> (BinOp -> b -> b -> b)
             -> EA -> b
foldEA c bp (Const n)      = c n
foldEA c bp (BOp op e1 e2) = bp op (foldEA c bp e1)
                                   (foldEA c bp e2)
```

Ejercicio 6)

```
s = foldExpA
```

Ejercicio 7)

```
recEA :: (Int -> b)
             -> (BinOp -> b -> EA -> b -> EA -> b)
             -> EA -> b
recEA c bp (Const n)      = c n
recEA c bp (BOp op e1 e2) = bp op (recEA c bp e1) e1
                                  (recEA c bp e2) e2
```

Ejercicio 8)

```
s = foldExpA
```

Ejercicio 9)

```
foldT :: b -> (a -> b -> b -> b)
           -> Tree a -> b
foldT empty node ET           = empty
foldT empty node (NT x t1 t2) = node x (foldT empty node t1)
                                       (foldT empty node t2)
```

Ejercicio 10)

```
s = foldExpA
```

Ejercicio 11)

```
recT :: b -> (a -> b -> Tree a -> b -> Tree a -> b)
          -> Tree a -> b
recT empty node ET           = empty
recT empty node (NT x t1 t2) = node x (recT  empty node t1) t1
                                      (recT  empty node t2) t2
```

Ejercicio 12)

```
s = foldExpA
```

Ejercicio 13)

```
foldD :: b -> ([a] -> b)
          -> (b -> b)
          -> (b -> b -> b)
          -> (b -> b -> b -> b -> b)
          -> Dep a -> b
foldD ss al sg bf hb SinSalida    = ss
foldD ss al sg bf hb (Almacen xs) = al xs
foldD ss al sg bf hb (Siga d)     = sg (foldD ss al sg bf hb d)
foldD ss al sg bf hb (Bif d1 d2)  = bf (foldD ss al sg bf hb d1)
                                       (foldD ss al sg bf hb d2)
foldD ss al sg bf hb (Hub d1 d2 d3 d4) =
    hb (foldD ss al sg bf hb d1)
       (foldD ss al sg bf hb d2)
       (foldD ss al sg bf hb d3)
       (foldD ss al sg bf hb d4)
```

Ejercicio 14)

```
saquearAlmacenes =
    foldD [] id id (++)
          (\xs1 xs2 xs3 xs4 -> xs1 ++ xs2 ++ xs3 ++ xs4)

contarHubs =
    foldD 0 (\xs -> 0) (\n -> n) (\n1 n2 -> n1 + n2)
          (\n1 n2 n3 n4 -> n1 + n2 + n3 + n4)

caminosSinSalida p =
  foldD [[]]
        []
        (\gs -> map (Seguir:) gs)
        (\gs1 gs2 -> map (IrAIzq:) gs1 ++ map (IrADer:) gs2)
        (\gs1 gs2 gs3 gs4 ->
```

```
            map (UsarPuerta 1:) gs1
          ++ map (UsarPuerta 2:) gs2
          ++ map (UsarPuerta 3:) gs3
          ++ map (UsarPuerta 4:) gs4)

caminosAAlmacenes p =
  foldD []
        [[]]
        (\gs -> map (Seguir:) gs)
        (\gs1 gs2 -> map (IrAIzq:) gs1 ++ map (IrADer:) gs2)
        (\gs1 gs2 gs3 gs4 ->
              map (UsarPuerta 1:) gs1
          ++ map (UsarPuerta 2:) gs2
          ++ map (UsarPuerta 3:) gs3
          ++ map (UsarPuerta 4:) gs4)
```

Ejercicio 15)

```
recD :: b -> ([a] -> b)
          -> (b -> Dep a -> b)
          -> (b -> Dep a -> b -> Dep a -> b)
          -> (b -> Dep a -> b -> Dep a
           -> b -> Dep a -> b -> Dep a - >b)
          -> Dep a -> b
recD ss al sg bf hb SinSalida    = ss
recD ss al sg bf hb (Almacen xs) = al xs
recD ss al sg bf hb (Siga d)     = sg (recD  ss al sg bf hb d) d
recD ss al sg bf hb (Bif d1 d2)  = bf (recD  ss al sg bf hb d1) d1
                                      (recD  ss al sg bf hb d2) d2
recD ss al sg bf hb (Hub d1 d2 d3 d4) =
    hb (recD  ss al sg bf hb d1) d1
       (recD  ss al sg bf hb d2) d2
       (recD  ss al sg bf hb d3) d3
       (recD  ss al sg bf hb d4) d4
```

Ejercicio 16)

```
caminosHasta p =
   recD (singularSi (p SinSalida) [])
        (\xs -> singularSi (p (Almacen xs)) [])
        (\gs d -> singularSi (p (Seguir d)) [] ++ map (Seguir:) gs)
        (\gs1 d1 gs2 d2 ->
              singularSi (p (Bif d1 d2)) []
          ++ map (IrAIzq:) gs1
          ++ map (IrADer:) gs2)
```

```
(\gs1 d1 gs2 d2 gs3 d3 gs4 d4 ->
      singularSi (p (Hub d1 d2 d3 d4)) []
   ++ map (UsarPuerta 1:) gs1
   ++ map (UsarPuerta 2:) gs2
   ++ map (UsarPuerta 3:) gs3
   ++ map (UsarPuerta 4:) gs4)
```