Programación Funcional

Clases teóricas

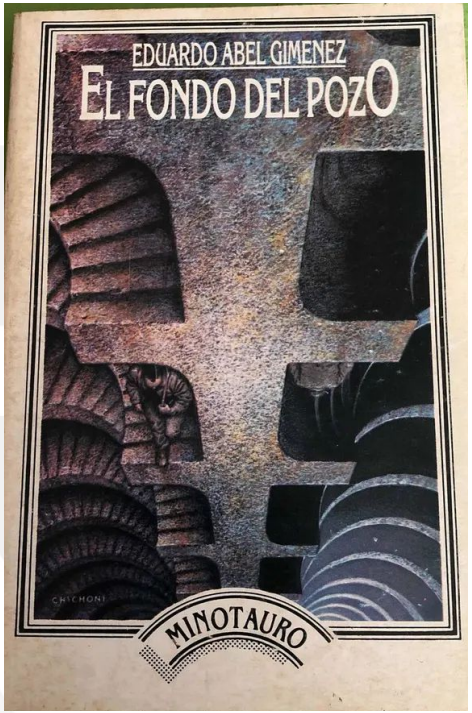
por Pablo E. “Fidel” Martínez López

12. Esquemas de funciones II

“'Todo es pasajero. La verdad depende del momento. Baje los ojos. Incline la cabeza. Cuente hasta diez. Descubrirá otra verdad.'

(Consejero, 74:96:3)”

El Fondo del Pozo
Eduardo Abel Giménez



Previously on Programación Funcional...

Esquema de recursión estructural sobre listas

- Se expresó el esquema de recursión estructural para listas como una función en Haskell

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

  - Todas las funciones recursivas estructurales sobre listas se pueden definir con él
  - Provee diversas ventajas:
    - Expresividad
    - Modularidad
    - Propiedades generales

Esquema de recursión estructural sobre listas

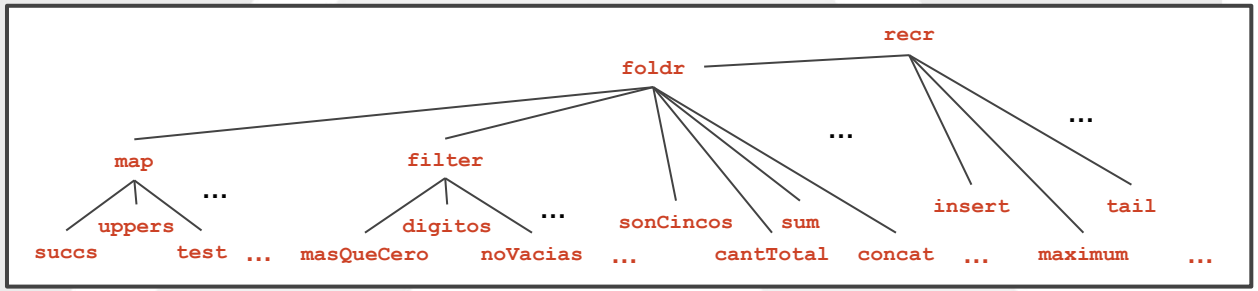
- ***Esquema de recursión estructural (foldr)***

```
foldr :: (a->b->b) -> b -> ([a] -> b)
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

  - *¡La función* **`foldr`** *ES la* ***expresión en código*** *de la noción de* ***recursión estructural sobre listas****!*
  - Permite abstraer muchas de las definiciones vistas
  - Existen otros esquemas sobre listas también

Abstracción

- ***Abstraer*** es detectar similaridades (ignorando diferencias) y aprovecharlas
- Los parámetros son una forma sintáctica de expresar abstracción
- Es como “subir” y mirar desde mayor altura en forma vertical



Abstracción

- ***Abstraer*** es detectar similaridades (ignorando diferencias) y aprovecharlas
- Los parámetros son una forma sintáctica de expresar abstracción
- Es como “subir” y mirar desde mayor altura en forma vertical
  - ¿Se podrá subir más aún? ¿Cuánto?

Esquemas de recursión en árboles

Esquemas en árboles

- La función **`foldr`** expresa el esquema de recursión estructural sobre listas como función en Haskell
- Todo tipo algebraico recursivo tiene asociado un esquema de recursión estructural

Esquemas en árboles

- La función **`foldr`** expresa el esquema de recursión estructural sobre listas como función en Haskell
- Todo tipo algebraico recursivo tiene asociado un esquema de recursión estructural
- ¿Existirá una forma de expresar cada uno de esos esquemas como funciones en Haskell?

Esquemas en árboles

- La función **`foldr`** expresa el esquema de recursión estructural sobre listas como función en Haskell
- Todo tipo algebraico recursivo tiene asociado un esquema de recursión estructural
- ¿Existirá una forma de expresar cada uno de esos esquemas como funciones en Haskell?
  - ¡Sí! Pero el sistema de tipos la va a poner algo difícil…

Esquemas en árboles binarios

Árboles binarios

- Los árboles binarios son un tipo algebraico recursivo

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
```

  - Para definir funciones recursivas se usaba un esquema

```
h :: Arbol a -> B
h (Hoja x)       = ... x ...1
h (Nodo x t1 t2) = ... x ... h t1 ... h t2 ...2
```

  - ¿Cómo expresar este esquema como función en Haskell?

Árboles binarios

- Los árboles binarios son un tipo algebraico recursivo

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
```

  - Para definir funciones recursivas se usaba un esquema

```
h ::                                  Arbol a -> B
h       (Hoja x)          = ... x ...1
h       (Nodo x t1 t2)    = ... x ... h t1 ... h t2 ...2
```

  - ¿Cómo expresar este esquema como función en Haskell?

Árboles binarios

- Los árboles binarios son un tipo algebraico recursivo

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
```

  - La función *fold* de árboles sigue el mismo esquema

```
foldA ::    ??   ->        ??      -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) = g x (foldA f g t1)
                               (foldA f g t2)
```

    - ¡Los ... se reemplazan por parámetros!
    - ¿Cuál es el tipo de esos parámetros?

Árboles binarios

- Los árboles binarios son un tipo algebraico recursivo

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
```

  - La función *fold* de árboles sigue el mismo esquema

```
foldA :: (a->b) -> (a->b->b->b) -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) = g x (foldA f g t1)
                               (foldA f g t2)
```

    - ¿Cuál es el tipo de esos parámetros?
      - Son funciones…

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - La función *fold* de árboles sigue el mismo esquema

```
h :: Arbol a -> B
h = foldA f g
  where f x          = ... x ...1
        g x r1 r2 = ... x ... r1 ... r2 ...2
```

```
h :: Arbol a -> B
h (Hoja x)       = ... x ...1
h (Nodo x t1 t2) = ... x ... h t1 ... h t2 ...2
```

    - Las partes verdes y negras se redistribuyen
    - Las partes azules proveen las conexiones

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo entender el tipo de los parámetros de **foldA**?
    - Observar el tipo de los constructores…

```
Hoja :: a -> Arbol a

Nodo :: a -> Arbol a -> Arbol a -> Arbol a
```

    - ¿Qué tipo tiene **foldA f g**?
      - **Arbol a -> b**

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                        -> Arbol a -> b
foldA f g (Hoja x)       = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo entender el tipo de los parámetros de **`foldA`**?
    - Observar el tipo de los constructores…

```
                    b
Hoja :: a -> Arbol a

                    b          b          b
Nodo :: a -> Arbol a -> Arbol a -> Arbol a
```

    - ¿Qué tipo tiene **`foldA f g`**?
      - Se transforma un **`Arbol a`** en un `b`

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo entender el tipo de los parámetros de **`foldA`**?
    - Observar el tipo de los constructores…

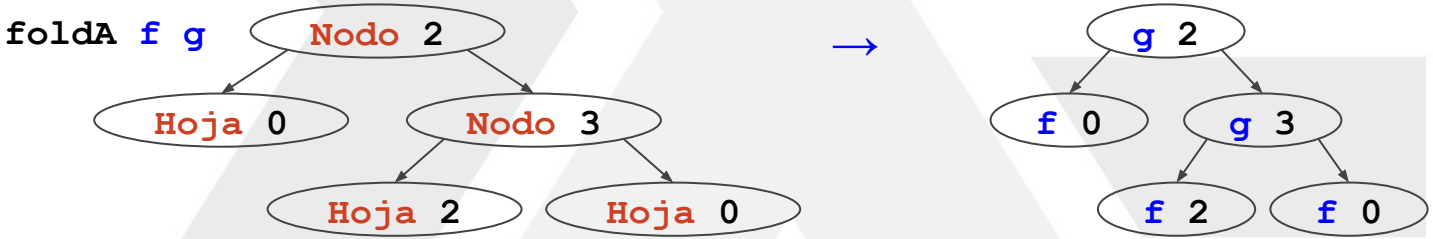
```
   f :: a ->    b
Hoja :: a -> Arbol a
   g :: a ->    b    ->    b    ->    b
Nodo :: a -> Arbol a -> Arbol a -> Arbol a
```

    - ¿Se observa la relación?
      - ¡Los parámetros reemplazan a los constructores!

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - Los parámetros reemplazan a los constructores



  - El cómputo mantiene la *estructura* de los datos

```
foldA f g (Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0)))
        = g     2 (f    0) (g     3 (f    2) (f    0))
```

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **foldA**?
    - ¡Los pasos son los mismos!

```
sumA :: Arbol Int -> Int  -- La suma de todos los elementos
sumA ...
```

      - Se decide usar recursión estructural...

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **foldA**?
    - ¡Los pasos son los mismos!

```
sumA :: Arbol Int -> Int   -- La suma de todos los elementos
sumA = foldA (\n          -> ... n ... )
             (\n n1 n2 -> ... n ... n1 ... n2 ...)
```

      - Se “escribe” el esquema de recursión estructural
        - Observar que ahora solamente se escribe **foldA** porque esta función expresa ese esquema
        - ¿Cuáles son los llamados recursivos?

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **`foldA`**?
    - ¡Los pasos son los mismos!

```
sumA :: Arbol Int -> Int   -- La suma de todos los elementos
sumA = foldA (\n          -> ... n ... )
             (\n n1 n2 -> n + n1 + n2)
```

      - Se decide el caso inductivo
        - Observar que los casos recursivos son el *resultado* de los llamados y **no** se conocen los subárboles

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **foldA**?
    - ¡Los pasos son los mismos!

```
sumA :: Arbol Int -> Int   -- La suma de todos los elementos
sumA = foldA (\n          -> n)
             (\n n1 n2 -> n + n1 + n2)
```

      - Se completa con el caso base
      - ¿Cómo queda si se expanden las definiciones?

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **`foldA`**?
    - ¡Los pasos son los mismos!

```
sumA :: Arbol Int -> Int   -- La suma de todos los elementos
sumA = foldA (\n          -> n)
             (\n n1 n2 -> n + n1 + n2)
```

    - ¿Cómo queda si se expanden las definiciones?

```
sumA (Hoja n)        = n
sumA (Nodo n t1 t2) = n + sumA t1 + sumA t2
```

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **foldA**?
    - Es más simple escribir si se sigue el esquema

```
hojasA :: Arbol a -> Int    -- La cantidad de hojas del árbol
hojasA  = foldA (\x        -> 1)
                (\x h1 h2 -> h1 + h2)

alturaA :: Arbol a -> Int   -- La cantidad de hojas del árbol
alturaA = foldA (\x        -> 0)
                (\x a1 a2 -> 1 + max a1 a2)
```

      - ¿Por qué el primer argumento es una función?

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **foldA**?
    - Es más simple escribir si se sigue el esquema

```
hojasA :: Arbol a -> Int    -- La cantidad de hojas del árbol
hojasA  = foldA (const 1)
                (const (+))

alturaA :: Arbol a -> Int   -- La cantidad de hojas del árbol
alturaA = foldA (const 0)
                (\x a1 a2 -> 1 + max a1 a2)
```

      - ¿Por qué el primer argumento es una función?

Árboles binarios

```
data Arbol a = Hoja a
             | Nodo a (Arbol a) (Arbol a)
foldA :: (a->b) -> (a->b->b->b)
                          -> Arbol a -> b
foldA f g (Hoja x)        = f x
foldA f g (Nodo x t1 t2) =
        g x (foldA f g t1) (foldA f g t2)
```

- Los árboles binarios son un tipo algebraico recursivo
  - ¿Cómo definir funciones usando **foldA**?
    - Es más simple escribir si se sigue el esquema

```
inOrderA :: Arbol a -> [a]      -- El listado in orden
inOrderA  = foldA (\x          -> [x])
                  (\x xs1 xs2 -> xs1 ++ [x] ++ xs2)

dupA :: Arbol Int -> Arbol Int -- Elementos duplicados
dupA = foldA (\n          -> Hoja (2*n))
             (\n t1' t2' -> Nodo (2*n) t1' t2')
```

      - Funciones de transformación de estructuras

Árboles binarios

- Los árboles binarios son un tipo algebraico recursivo
  - Se expresó el patrón de recursión estructural sobre este tipo como función de orden superior, **`foldA`**
    - Se vio que los parámetros se vinculan a los constructores
  - Se (re)definieron funciones usando **`foldA`**
    - Es mucho más conciso y expresivo
  - Se pueden demostrar diversas propiedades sobre **`foldA`**
    - Similares a las que se demostraron para **`foldr`**
  - ¿Qué pasará con otros tipos recursivos?

Esquemas en expresiones aritméticas

Expresiones aritméticas

- Otro tipo algebraico recursivo: expresiones aritméticas

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

  - Para definir funciones recursivas se usaba un esquema

```
h :: ExpA -> B
h (Cte x)          = ... x ...1
h (Suma e1 e2)     = ... h e1 ... h e2 ...2
h (Prod e1 e2)     = ... h e1 ... h e2 ...3
```

  - ¿Cómo expresar este esquema como función en Haskell?

Expresiones aritméticas

- Otro tipo algebraico recursivo: expresiones aritméticas

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

  - Para definir funciones recursivas se usaba un esquema

```
h ::                                                ExpA -> B
h          (Cte x)          = ... x ...1
h          (Suma e1 e2)     = ... h e1 ... h e2 ...2

h          (Prod e1 e2)     = ... h e1 ... h e2 ...3
```

  - ¿Cómo expresar este esquema como función en Haskell?

Expresiones aritméticas

- Otro tipo algebraico recursivo: expresiones aritméticas

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

  - La función fold de **ExpA** sigue ese mismo esquema

```
foldExpA ::     ??    ->     ??     ->     ??   -> 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)
```

    - ¡Los ... se reemplazan por parámetros!
    - ¿Cuál es el tipo de esos parámetros?

Expresiones aritméticas

- Otro tipo algebraico recursivo: expresiones aritméticas

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

  - La función fold de **ExpA** sigue ese mismo esquema

```
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)
```

    - ¡Los ... se reemplazan por parámetros!
      - Son funciones…

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - La función fold de **ExpA** sigue ese mismo esquema

```
h :: ExpA -> B
h = foldExpA c s p
  where c n      = ... n ...1
        s r1 r2  = ... r1 ... r2 ...2
        p r1 r2  = ... r1 ... r2 ...3
```

```
h :: ExpA -> B
h (Cte x)       = ... x ...1
h (Suma e1 e2)  = ... h e1 ... h e2 ...2
h (Prod e1 e2)  = ... h e1 ... h e2 ...3
```

    - Las partes verdes y negras se redistribuyen
    - Las partes azules proveen las conexiones

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo entender el tipo de los parámetros de **foldExpA**?

```
Cte  :: Int -> ExpA

Suma :: ExpA -> ExpA -> ExpA

Prod :: ExpA -> ExpA -> ExpA
```

  - ¿Qué tipo tiene **foldExpA c s p**?
    - **ExpA -> b**

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo entender el tipo de los parámetros de **foldExpA**?

```
                    b
Cte   :: Int -> ExpA

          b         b         b
Suma :: ExpA -> ExpA -> ExpA

          b         b         b
Prod :: ExpA -> ExpA -> ExpA
```

  - ¿Qué tipo tiene **foldExpA c s p**?
    - Se transforma una **ExpA** en un **b**

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo entender el tipo de los parámetros de **`foldExpA`**?

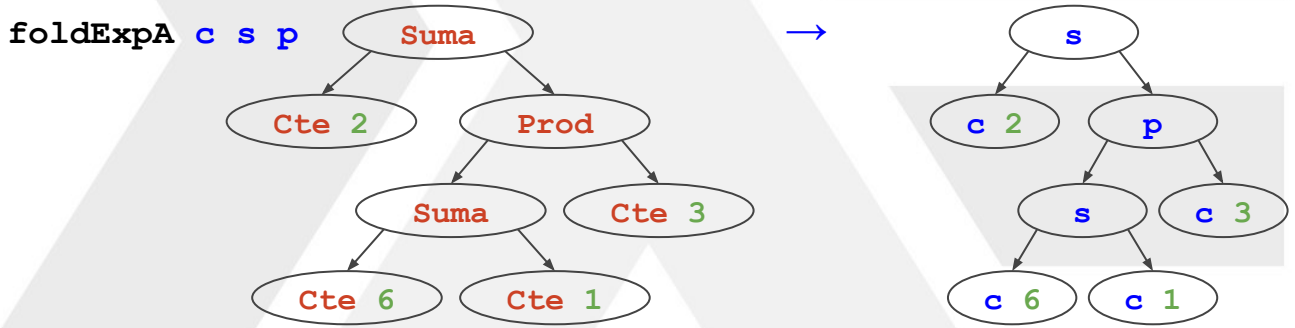
```
  c   :: Int ->   b
Cte   :: Int -> ExpA

  s   ::   b  ->   b  ->   b
Suma  :: ExpA -> ExpA -> ExpA

  p   ::   b  ->   b  ->   b
Prod  :: ExpA -> ExpA -> ExpA
```

    - Cada parámetro corresponde con un constructor
      - Nuevamente, los parámetros reemplazan a los constructores

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo entender el tipo de los parámetros de **`foldExpA`**?



  - Otra vez, el cómputo mantiene la estructura de los datos

```
foldExpA c s p (Suma (Cte 2) (Prod (Suma (Cte 6) (Cte 1)) (Cte 3))
             = s     (c   2) (p    (s    (c   6) (c   1)) (c   3))
```

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo definir funciones usando **foldExpA**?
    - Igual que antes, salvo que se escribe diferente

```
evalExpA :: ExpA -> Int
evalExpA = foldExpA (\n -> n)
                    (\n1 n2 -> n1 + n2)
                    (\n1 n2 -> n1 * n2)

expA2tb :: ExpA -> TB
expA2tb = foldExpA (\n -> C (int2ta n))
                   (\e1' e2' -> D e1' e2')
                   (\e1' e2' -> E e1' e2')
```

```
data TA = A | B TA
data T = C TA | D TB TB
               | E TB TB

int2ta :: Int -> A
int2ta = foldN B A
```

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo definir funciones usando **foldExpA**?
    - Igual que antes, salvo que se escribe diferente

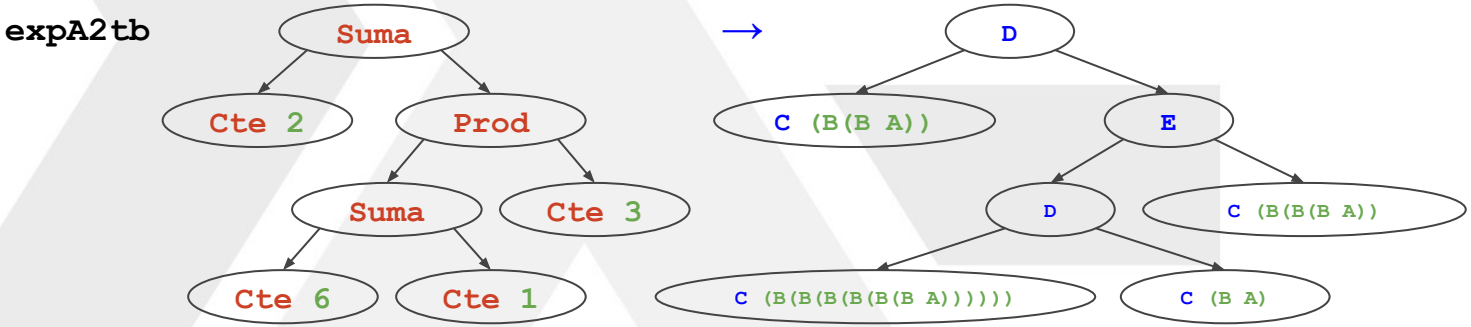
```
evalExpA :: ExpA -> Int
evalExpA = foldExpA id (+) (*)


expA2tb :: ExpA -> TB
expA2tb = foldExpA (C . int2ta) D E
```

```
data TA = A | B TA
data T = C TA | D TB TB
               | E TB TB

int2ta :: Int -> A
int2ta = foldN B A
```

Expresiones aritméti

```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
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) = s (foldExpA c s p e1)
                                (foldExpA c s p e2)
```

- Otro tipo algebraico recursivo: expresiones aritméticas
  - ¿Cómo entender el tipo de los parámetros de **`foldExpA`**?



  - Otra vez, el cómputo mantiene la estructura de los datos

```
expA2tb (Suma (Cte 2)      (Prod (Suma (Cte 6)                    (Cte 1)) (Cte 3))
      = D     (C (B (B A))) (E    (D    (C (B(B(B(B(B A)))))) (C (B A))) (C   (B(B(B A)))))
```

Expresiones aritméticas

- El tipo **`ExpA`** es un tipo algebraico recursivo
  - Se expresó el patrón de recursión estructural sobre este tipo como función de orden superior, **`foldExpA`**
    - Se vio que los parámetros se vinculan a los constructores
  - Se (re)definieron funciones usando **`foldExpA`**
    - Es mucho más conciso y expresivo
  - Se pueden demostrar propiedades sobre **`foldExpA`**
    - Similares a las que se demostraron para **`foldr`** y **`foldA`**
  - ¿Es generalizable esta secuencia de trabajo?

Esquemas en otros tipos recursivos

Otros tipos recursivos

- Un tipo algebraico recursivo genérico

```
data TG = CB | CC TG Char TG | CD Int TG
        | CE TG TG   TG | CF TG Char
```

  - Este tipo no tiene una intención de significado premeditada
  - Pero la función **`foldTG`** se construye con la misma secuencia

```
foldTG ::   ??  -> TG -> b
foldTG …
```

    - ¿Cuántos parámetros adicionales debe haber?
    - ¿De qué tipos deben ser?

Otros tipos recursivos

```
data TG = CB
        | CC TG  Char TG
        | CD Int TG
        | CE TG  TG   TG
        | CF TG  Char
```

- Un tipo algebraico recursivo genérico
  - La función **foldTG** se construye con la misma secuencia

```
foldTG :: ?? ->       ??       ->     ??
          ->       ??       ->     ??       -> TG -> b
foldTG b c d e f  CB            = b
foldTG b c d e f (CC g1 ch g2) = c   (foldTG b c d e f g1) ch
                                     (foldTG b c d e f g2)
foldTG b c d e f (CD n g1)     = d n (foldTG b c d e f g1)
foldTG b c d e f (CE g1 g2 g3) = e   (foldTG b c d e f g1)
                                     (foldTG b c d e f g2)
                                     (foldTG b c d e f g3)
foldTG b c d e f (CF g ch)     = f   (foldTG b c d e f g1) ch
```

  - Un parámetro por cada constructor, con tipos similares

Otros tipos recursivos

```
data TG = CB
        | CC TG  Char TG
        | CD Int TG
        | CE TG  TG   TG
        | CF TG  Char
```

- Un tipo algebraico recursivo genérico
  - La función **foldTG** se construye con la misma secuencia

```
foldTG :: b -> (b->Char->b->b) -> (Int->b->b)
           -> (b->b->b->b)     -> (b->Char->b) -> TG -> b
foldTG b c d e f  CB            = b
foldTG b c d e f (CC g1 ch g2) = c   (foldTG b c d e f g1) ch
                                     (foldTG b c d e f g2)
foldTG b c d e f (CD n g1)     = d n (foldTG b c d e f g1)
foldTG b c d e f (CE g1 g2 g3) = e   (foldTG b c d e f g1)
                                     (foldTG b c d e f g2)
                                     (foldTG b c d e f g3)
foldTG b c d e f (CF g ch)     = f   (foldTG b c d e f g1) ch
```

  - Un parámetro por cada constructor, con tipos similares

Otros tipos recursivos

```
data TG = CB
        | CC TG  Char TG
        | CD Int TG
        | CE TG  TG   TG
        | CF TG  Char
```

```
foldTG :: b -> (b->Char->b->b)
           -> (Int->b->b)
           -> (b->b->b->b)
           -> (b->Char->b)
           -> TG -> b
foldTG b c d e f CB           = b
foldTG b c d e f (CC g1 ch g2) =
    c   (foldTG b c d e f g1) ch
        (foldTG b c d e f g2)
foldTG b c d e f (CD n g1)     =
    d n (foldTG b c d e f g1)
foldTG b c d e f (CE g1 g2 g3) =
    e   (foldTG b c d e f g1)
        (foldTG b c d e f g2)
        (foldTG b c d e f g3)
foldTG b c d e f (CF g ch)     =
    f   (foldTG b c d e f g1) ch
```

- Un tipo algebraico recursivo genérico
  - La definición de funciones usando **`foldTG`** provee los argumentos, como antes

```
cantCharTG :: TG -> Int
cantCharTG =
   foldTG ... (\n1 c n2 -> ... n1 ... c ... n2 ...)
              (\m  n -> ... m ... n ...)
              (\n1 n2 n3 -> ... n1 ... n2 ... n3 ...)
              (\n c -> ... n ... c ...)
```

  - ¿Cuáles son los llamados recursivos, y qué significan?

Otros tipos recursivos

```
data TG = CB
        | CC TG  Char TG
        | CD Int TG
        | CE TG  TG   TG
        | CF TG  Char
```

```
foldTG :: b -> (b->Char->b->b)
           -> (Int->b->b)
           -> (b->b->b->b)
           -> (b->Char->b)
           -> TG -> b
foldTG b c d e f CB           = b
foldTG b c d e f (CC g1 ch g2) =
    c   (foldTG b c d e f g1) ch
        (foldTG b c d e f g2)
foldTG b c d e f (CD n g1)    =
    d n (foldTG b c d e f g1)
foldTG b c d e f (CE g1 g2 g3) =
    e   (foldTG b c d e f g1)
        (foldTG b c d e f g2)
        (foldTG b c d e f g3)
foldTG b c d e f (CF g ch)    =
    f   (foldTG b c d e f g1) ch
```

- Un tipo algebraico recursivo genérico
  - La definición de funciones usando **foldTG** provee los argumentos, como antes

```
cantCharTG :: TG -> Int
cantCharTG =
   foldTG 0 (\n1 c n2 -> n1 + 1 + n2)
            (\m  n -> n)
            (\n1 n2 n3 -> n1 + n2 + n3)
            (\n c -> n + 1)
```

  - La transformación es estructural
  - Solo el 2do y el último suman 1, porque tienen Chars

Otros tipos recursivos

- La recursión estructural de un tipo recursivo **`T`** se puede expresar como función de orden superior (**`foldT`**)

Otros tipos recursivos

- La recursión estructural de un tipo recursivo `T` se puede expresar como función de orden superior (`foldT`)
- Para construirla
  - Se necesitan tantos parámetros como constructores
  - El tipo de cada parámetro está vinculado al del constructor correspondiente, cambiando `T` por `b`

Otros tipos recursivos

- La recursión estructural de un tipo recursivo `T` se puede expresar como función de orden superior (`foldT`)
- Para construirla
  - Se necesitan tantos parámetros como constructores
  - El tipo de cada parámetro está vinculado al del constructor correspondiente, cambiando `T` por `b`
- Se pueden demostrar propiedades generales para cada `foldT`

Otros tipos recursivos

- La recursión estructural de un tipo recursivo `T` se puede expresar como función de orden superior (`foldT`)
- Para construirla
  - Se necesitan tantos parámetros como constructores
  - El tipo de cada parámetro está vinculado al del constructor correspondiente, cambiando `T` por `b`
- Se pueden demostrar propiedades generales para cada `foldT`
- ¿Se puede definir una función genérica que exprese todos?

Otros tipos recursivos

- La recursión estructural de un tipo recursivo `T` se puede expresar como función de orden superior (`foldT`)
- Para construirla
  - Se necesitan tantos parámetros como constructores
  - El tipo de cada parámetro está vinculado al del constructor correspondiente, cambiando `T` por `b`
- Se pueden demostrar propiedades generales para cada `foldT`
- ¿Se puede definir una función genérica que exprese todos?
  - No en el sistema de tipos H-M (no hay forma de darle tipo)
  - Hay extensiones de Haskell donde sí es posible

Uso de esquemas para programación

Uso de esquemas

- Se pueden definir gran cantidad de esquemas útiles
- ¿Cómo usarlos para mejorar la práctica de programar?
  - Se los puede usar como subtareas parametrizadas
  - También sirven para combinar otras subtareas

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
or :: [Bool] -> Bool
or = foldr (||) False
any :: (a->Bool) -> [a] -> Bool
any p = or . map p
```

- Gran expresividad para resolver problemas
  - Indicar si el estudiante está tomando alguno de los cursos

    ```
    estaCursandoAlguno :: Estudiante -> [Curso] -> Bool
    estaCursandoAlguno ...
    ```

  - Calcular el promedio sin aplazos de cada estudiante

    ```
    promsSinAplazos :: [Estudiante] -> [Int]
    promsSinAplazos ...
    ```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
or :: [Bool] -> Bool
or = foldr (||) False
any :: (a->Bool) -> [a] -> Bool
any p = or . map p
```

- Gran expresividad para resolver probl

```
Estudiante -> Curso -> Bool
```

  - Indicar si el estudiante está tomando alguno de los cursos

```
estaCursandoAlguno :: Estudiante -> [Curso] -> Bool
estaCursandoAlguno e cs = any (estaInscriptoEn e) cs
```

  - Calcular el promedio sin aplazos de cada estudiante

```
promsSinAplazos :: [Estudiante] -> [Int]
promsSinAplazos ...
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
or :: [Bool] -> Bool
or = foldr (||) False
any :: (a->Bool) -> [a] -> Bool
any p = or . map p
```

- Gran expresividad para resolver probl

  Estudiante -> Curso -> Bool

  - Indicar si el estudiante está tomando alguno de los cursos

```
estaCursandoAlguno :: Estudiante -> [Curso] -> Bool
estaCursandoAlguno e cs = any (estaInscriptoEn e) cs
```

  - Calcular el promedio sin aplazos de cada estudiante

```
promsSinAplazos :: [Estudiante] -> [Int]
promsSinAplazos es = map promedio
                         (map (filter (>=4))
                            (map notas es))
```

Estudiante -> [Int]

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
or :: [Bool] -> Bool
or = foldr (||) False
any :: (a->Bool) -> [a] -> Bool
any p = or . map p
```

- Gran expresividad para resolver problemas
  - Indicar si el estudiante está tomando alguno de los cursos

```
estaCursandoAlguno :: Estudiante -> [Curso] -> Bool
estaCursandoAlguno e cs = any (estaInscriptoEn e) cs
```

  - Calcular el promedio sin aplazos de cada estudiante

```
promsSinAplazos :: [Estudiante] -> [Int]
promsSinAplazos es = map promedio
                         (map (filter (>=4))
                              (map notas es))
```

    - Mediante propiedades se puede hacer más conciso

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
or :: [Bool] -> Bool
or = foldr (||) False
any :: (a->Bool) -> [a] -> Bool
any p = or . map p
```

- Gran expresividad para resolver problemas
  - Indicar si el estudiante está tomando alguno de los cursos

```
estaCursandoAlguno :: Estudiante -> [Curso] -> Bool
estaCursandoAlguno = any . estaInscriptoEn
```

  - Calcular el promedio sin aplazos de cada estudiante

```
promsSinAplazos :: [Estudiante] -> [Int]
promsSinAplazos = map (promedio . filter (>=4) . notas)
```

    - Es más fácil que hacer recursiones explícitas cada vez
      - Requiere práctica y familiaridad con esquemas

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
and :: [Bool] -> Bool
and = foldr (&&) True
all :: (a->Bool) -> [a] -> Bool
all p = and . map p
```

- Gran expresividad para resolver problemas
  - La función ambos verifica que las dos condiciones se cumplen

```
ambas :: (a -> Bool) -> (a -> Bool) -> a -> Bool
ambas p1 p2 x = ...
```

  - Indicar si el estudiante cumple condiciones de promoción

```
promociona :: Estudiante -> Bool
promociona e = ...
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
and :: [Bool] -> Bool
and = foldr (&&) True
all :: (a->Bool) -> [a] -> Bool
all p = and . map p
```

- Gran expresividad para resolver problemas
  - La función ambos verifica que las dos condiciones se cumplen

```
ambas :: (a -> Bool) -> (a -> Bool) -> a -> Bool
ambas p1 p2 x = p1 x && p2 x
```

  - Indicar si el estudiante cumple condiciones de promoción

```
promociona :: Estudiante -> Bool
promociona e = ...
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
and :: [Bool] -> Bool
and = foldr (&&) True
all :: (a->Bool) -> [a] -> Bool
all p = and . map p
```

- Gran expresividad para resolver problemas
  - La función ambos verifica que las dos condiciones se cumplen

```
ambas :: (a -> Bool) -> (a -> Bool) -> a -> Bool
ambas p1 p2 x = p1 x && p2 x
```

  - Indicar si el estudiante cumple condiciones de promoción

```
promociona :: Estudiante -> Bool
promociona e = ambas (all (>=6))
                     (\ns -> promedio ns >= 7)) (notas e)
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
and :: [Bool] -> Bool
and = foldr (&&) True
all :: (a->Bool) -> [a] -> Bool
all p = and . map p
```

- Gran expresividad para resolver problemas
  - La función ambos verifica que las dos condiciones se cumplen

```
ambas :: (a -> Bool) -> (a -> Bool) -> a -> Bool
ambas p1 p2 x = p1 x && p2 x
```

  - Indicar si el estudiante cumple condiciones de promoción

```
promociona :: Estudiante -> Bool
promociona e = ambas (all (>=6))
                     ((>=7) . promedio) (notas e)
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
and :: [Bool] -> Bool
and = foldr (&&) True
all :: (a->Bool) -> [a] -> Bool
all p = and . map p
```

- Gran expresividad para resolver problemas
  - La función ambos verifica que las dos condiciones se cumplen

```
ambas :: (a -> Bool) -> (a -> Bool) -> a -> Bool
ambas p1 p2 x = p1 x && p2 x
```

  - Indicar si el estudiante cumple condiciones de promoción

```
promociona :: Estudiante -> Bool
promociona = ambas (all (>=6))
                   ((>=7) . promedio) . notas
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
and :: [Bool] -> Bool
and = foldr (&&) True
all :: (a->Bool) -> [a] -> Bool
all p = and . map p
```

- Gran expresividad para resolver problemas
  - La función ambos verifica que las dos condiciones se cumplen

```
ambas :: (a -> Bool) -> (a -> Bool) -> a -> Bool
ambas p1 p2 x = p1 x && p2 x
```

  - Indicar si el estudiante cumple condiciones de promoción

```
promociona :: Estudiante -> Bool
promociona = ambas (all (>=6))
                   ((>=7) . promedio) . notas
```

    - Es posible definir otras funciones de orden superior útiles

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
appFork :: (a->b,a->c) -> a -> (b,c)
appFork (f,g) x = (f x, g x)
data TPizza = Muzza | Fugazetta | Especial
            | Roque | Panceta | Anana | …
```

- Gran expresividad para resolver problemas
  - Dado un tipo de pizzas y una lista de pedidos de porciones, indicar cuántas porciones en total se pidieron de ese tipo

    ```
    cuantasDe :: TPizza -> [(TPizza, Int)] -> Int
    cuantasDe ...
    ```

  - Dada una lista de pares no vacía donde todos los 1ros elementos son iguales, dar el par (ese elemento, el total de todos los 2dos)

    ```
    totalizar :: [(a, Int)] -> (a, Int)
    -- PRECOND: la lista no vacía, todos los a iguales
    totalizar ...
    ```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
appFork :: (a->b,a->c) -> a -> (b,c)
appFork (f,g) x = (f x, g x)
data TPizza = Muzza | Fugazetta | Especial
            | Roque | Panceta | Anana | …
```

- Gran expresividad para resolver problemas
  - Dado un tipo de pizzas y una lista de pedidos de porciones, indicar cuántas porciones en total se pidieron de ese tipo

```
cuantasDe :: TPizza -> [(TPizza, Int)] -> Int
cuantasDe …
```

```
cuantasDe Muzza [(Muzza,3), (Roque,1), (Panceta,2)
                ,(Anana,2), (Muzza,4), (Roque,1)  ] = 7
```

  - Dada una lista de pares no vacía donde todos los 1ros elementos son iguales, dar el par (ese elemento, el total de todos los 2dos)

```
totalizar :: [(a, Int)] -> (a, Int)
-- PRECOND: la lista no vacía, todos los a iguales
totalizar …
```

```
totalizar [(Roque,3), (Roque, 4)
          ,(Roque,2), (Roque, 3)] = (Roque, 12)
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
appFork :: (a->b,a->c) -> a -> (b,c)
appFork (f,g) x = (f x, g x)
data TPizza = Muzza | Fugazetta | Especial
            | Roque | Panceta | Anana | …
```

- Gran expresividad para resolver problemas
  - Dado un tipo de pizzas y una lista de pedidos de porciones, indicar cuántas porciones en total se pidieron de ese tipo

```
cuantasDe :: TPizza -> [(TPizza, Int)] -> Int
cuantasDe tp cps = sum (map snd (filter ((==tp) . fst) cps))
```

  - Dada una lista de pares no vacía donde todos los 1ros elementos son iguales, dar el par (ese elemento, el total de todos los 2dos)

```
totalizar :: [(a, Int)] -> (a, Int)
-- PRECOND: la lista no vacía, todos los a iguales
totalizar xns = ( fst (head xns) , sum (map snd xns) )
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
appFork :: (a->b,a->c) -> a -> (b,c)
appFork (f,g) x = (f x, g x)
data TPizza = Muzza | Fugazetta | Especial
            | Roque | Panceta | Anana | …
```

- Gran expresividad para resolver problemas
  - Dado un tipo de pizzas y una lista de pedidos de porciones, indicar cuántas porciones en total se pidieron de ese tipo

```
cuantasDe :: TPizza -> [(TPizza, Int)] -> Int
cuantasDe tp = sum . map snd . filter ((==tp) . fst)
```

  - Dada una lista de pares no vacía donde todos los 1ros elementos son iguales, dar el par (ese elemento, el total de todos los 2dos)

```
totalizar :: [(a, Int)] -> (a, Int)
-- PRECOND: la lista no vacía, todos los a iguales
totalizar xns = ( (fst . head) xns , (sum . map snd) xns)
```

Uso de esquemas

```
map :: (a->b) -> [a] -> [b]
filter :: (a->Bool) -> [a] -> [a]
appFork :: (a->b,a->c) -> a -> (b,c)
appFork (f,g) x = (f x, g x)
data TPizza = Muzza | Fugazetta | Especial
            | Roque | Panceta | Anana | …
```

- Gran expresividad para resolver problemas
  - Dado un tipo de pizzas y una lista de pedidos de porciones, indicar cuántas porciones en total se pidieron de ese tipo

```
cuantasDe :: TPizza -> [(TPizza, Int)] -> Int
cuantasDe tp = sum . map snd . filter ((==tp) . fst)
```

  - Dada una lista de pares no vacía donde todos los 1ros elementos son iguales, dar el par (ese elemento, el total de todos los 2dos)

```
totalizar :: [(a, Int)] -> (a, Int)
-- PRECOND: la lista no vacía, todos los a iguales
totalizar = appFork (fst . head, sum . map snd)
```

Uso de esquemas

- Se pueden definir gran cantidad de esquemas útiles
- ¿Cómo usarlos para mejorar la práctica de programar?
  - Se los puede usar como subtareas parametrizadas
  - También sirven para combinar otras subtareas
- Trabajar en forma denotacional es mucho más expresivo y eficaz

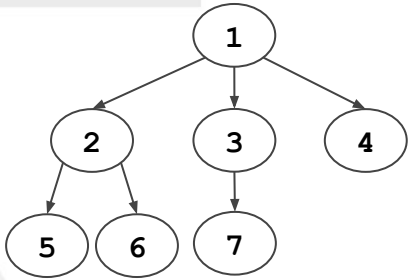
Árboles generales

Árboles generales

- ¿Cómo definir un árbol con cantidad variable de hijos?
  - No es opción tener un constructor por cada cantidad

```
data WT a = Cero a
          | Uno  a (WT a)
          | Dos  a (WT a) (WT a)
          | Tres a (WT a) (WT a) (WT a)
          ...
```

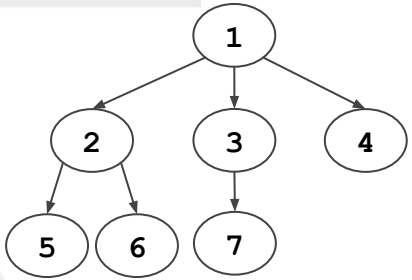
```
wt = Tres 1 (Dos  2 (Cero 5)
                    (Cero 6))
            (Uno  3 (Cero 7))
            (Cero 4)
```



  - ¿Cuántos más habría que poner?
    ¿Y si hay que agregar o sacar hijos?
  - ¿Cómo definirlo, entonces?

Árboles generales

- ¿Cómo definir un árbol con cantidad variable de hijos?
  - ¡Con una lista de hijos!

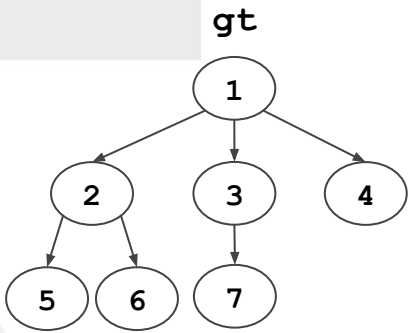


Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¡Con una lista de hijos!

```
data GTree a = GNode a [GTree a]
gt = ...
```

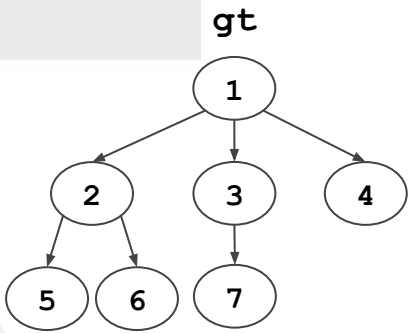
  - En la lista se pueden agregar o sacar hijos
  - ¿Cuál es el caso base?



Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¡Con una lista de hijos!

```
data GTree a = GNode a [GTree a]
gt = GNode 1 [ GNode 2 [ GNode 5 []
                       , GNode 6 [] ]
             , GNode 3 [ GNode 7 [] ]
             , GNode 4 [] ]
```



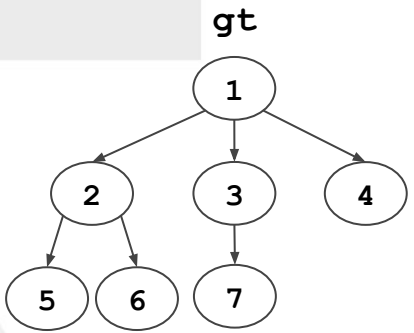
  - En la lista se pueden agregar o sacar hijos
  - ¿Cuál es el caso base?

```
data GenTree a = GNode a [GenTree a]
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - Su estructura está entretejida con la estructura de listas
  - ¿Cómo definir funciones sobre **`GTree`**s?

```
sumGT :: GTree Int -> Int
sumGT (GNode x ts) = ...
```



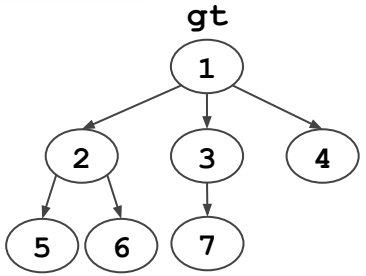
```
data GenTree a = GNode a [GenTree a]
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - Su estructura está entretejida con la estructura de listas
  - ¿Cómo definir funciones sobre **`GTree`**s?
    - ¡Usando funciones sobre listas!

```
sumGT :: GTree Int -> Int
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

    - La importancia de los esquemas de listas
    - ¿Y esto es estructural?



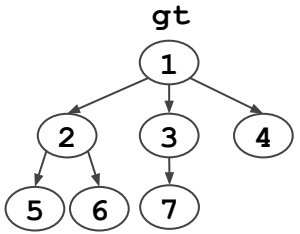
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Cómo funciona **`sumGT`**?

```
sumGT gt
→ 1 + sum (map sumGT [ GNode 2 [ GNode 5 [], GNode 6 [] ]
                     , GNode 3 [ GNode 7 [] ]
                     , GNode 4 [] ])
→ 1 + sum [ sumGT (GNode 2 [ GNode 5 [], GNode 6 [] ])
          , sumGT (GNode 3 [ GNode 7 [] ])
          , sumGT (GNode 4 []) ]
```

  - El **`map`** aplica la recursión a todos,
    y el `sum` realiza la suma de los resultados



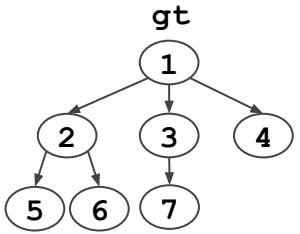
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Y si no hubiese orden superior? ¡Subtareas!

```
sumGTPO :: GTree Int -> Int
sumGTPO (GNode x ts) = x + aRaTyS ts

aRaTyS :: [GTree Int] -> Int -- Aplicar recursión a todos y sumar
aRaTyS ...
```

  - La subtarea trabaja sobre la lista
    - Si no hay esquemas, recursión explícita…



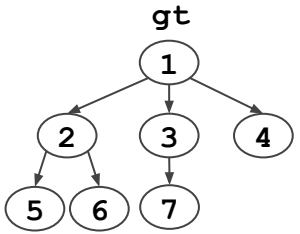
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Y si no hubiese orden superior? ¡Subtareas!

```
sumGTPO :: GTree Int -> Int
sumGTPO (GNode x ts) = x + aRaTyS ts

aRaTyS :: [GTree Int] -> Int -- Aplicar recursión a todos y sumar
aRaTyS []     = ...
aRaTyS (t:ts) = ... t ... aRaTyS ts ...
```

  - La subtarea trabaja sobre la lista
    - Si no hay esquemas, recursión explícita…



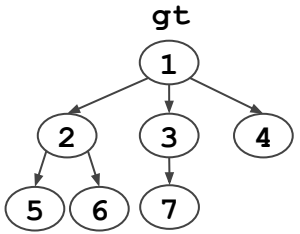
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Y si no hubiese orden superior? ¡Subtareas!

```
sumGTPO :: GTree Int -> Int
sumGTPO (GNode x ts) = x + aRaTyS ts

aRaTyS :: [GTree Int] -> Int -- Aplicar recursión a todos y sumar
aRaTyS []     = 0
aRaTyS (t:ts) = sumGTPO t + aRaTyS ts
```

  - La subtarea trabaja sobre la lista
    - Si no hay esquemas, recursión explícita…
  - Forma nueva de recursión: ¡recursión mutua!



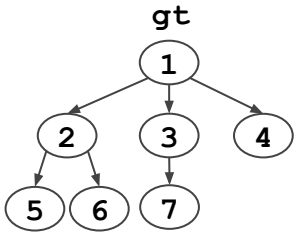
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Cómo funciona **`sumGT`**?

```
sumGTPO gt
→ 1 + aRaTyS [ GNode 2 [ GNode 5 [], GNode 6 [] ]
                       , GNode 3 [ GNode 7 [] ]
                       , GNode 4 [] ]
→ 1 + (sumGTPO (GNode 2 [ GNode 5 [], GNode 6 [] ])
    + aRaTyS [ GNode 3 [ GNode 7 [] ], GNode 4 [] ])
```

  - La función auxiliar aplica la recursión a todos y realiza la suma
  - Observar la recursión mutua



Árboles generales

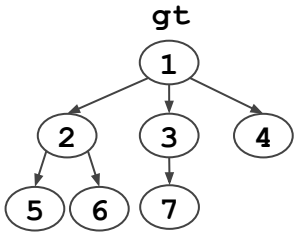
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

- Árboles generales (o *Rose Trees*)
  - ¿Cómo funciona **`sumGT`**?

```
sumGTPO gt
→ 1 + aRaTyS [ GNode 2 [ GNode 5 [], GNode 6 [] ]
                       , GNode 3 [ GNode 7 [] ]
                       , GNode 4 [] ]
→ 1 + (sumGTPO (GNode 2 [ GNode 5 [], GNode 6 [] ])
    + aRaTyS [ GNode 3 [ GNode 7 [] ], GNode 4 [] ])
```

```
sumGTPO ...
→
aRaTyS ...
→
sumGTPO ...
→
aRaTyS ...
...
```

  - La función auxiliar aplica la recursión a todos y realiza la suma
  - Observar la recursión mutua



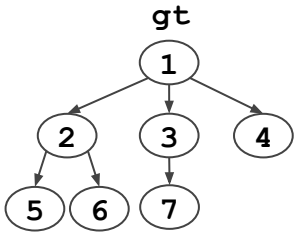
```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Cómo NO se debe pensar? Sin seguir el esquema…

```
sumGTFeo :: GTree Int -> Int
sumGTFeo (GNode x [])     = x
sumGTFeo (GNode x (t:ts)) = x + sumGTFeo t + ??  ts
```

    - Mezcla dos tareas en una sola
    - Las respuestas adecuadas la incógnita se dieron antes
      - Esquemas de listas
      - Recursión mutua
    - Otras respuestas posibles no son convenientes



Árboles generales

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?
    - Si se sigue la secuencia vista para otros tipos…

```
foldGT0 :: (a->[b]->b) -> GTree a -> b
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
```

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

Árboles generales

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?
    - Si se sigue la secuencia vista para otros tipos…

```
foldGT0 :: (a->[b]->b) -> GTree a -> b
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
```

    - La función **sumGT** se expresa como

```
sumGT0 = foldGT0 (\x ns -> x + sum ns)
```

Árboles generales

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
```

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?
    - Si se sigue la secuencia vista para otros tipos…

```
foldGT0 :: (a->[b]->b) -> GTree a -> b
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
```

    - La función **sumGT** se expresa como

```
sumGT0 = foldGT0 (\x ns -> x + sum ns)
```

    - ¡Pero esto NO es recursión estructural!
      - No hay garantías de tratamiento estructural para la lista

Árboles generales

```
data GenTree a = GNode a [GenTree a]
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
```

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?
    - Una versión completamente estructural…

```
foldGT1 :: (a->c->b)
                -> (b->c->c) -> c -> GTree a -> b
foldGT1 g f z (GNode x ts) =
      g x (foldr f z (map (foldGT1 g f z) ts))
```

Árboles generales

```
data GenTree a = GNode a [GenTree a]
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
```

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?
    - Una versión completamente estructural…

```
foldGT1 :: (a->c->b)
                -> (b->c->c) -> c -> GTree a -> b
foldGT1 g f z (GNode x ts) =
      g x (foldr f z (map (foldGT1 g f z) ts))
```

    - La función **sumGT** se expresa como

```
sumGT1 = foldGT1 (\x n -> x+n) (\n1 nr -> n1+nr) 0
```

    - Esto es recursión estructural, ¡pero es muy complicado!

Árboles generales

```
data GenTree a = GNode a [GenTree a]
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
```

- Árboles generales (o *Rose Trees*)
  - ¿Cómo sería la función *fold* para árboles generales?
    - Una versión completamente estructural…

```
foldGT1 :: (a->c->b)
                -> (b->c->c) -> c -> GTree a -> b
foldGT1 g f z (GNode x ts) =
      g x (foldr f z (map (foldGT1 g f z) ts))
```

    - La función **sumGT** se expresa como

```
sumGT1 = foldGT1 (+) (+) 0
```

    - Esto es recursión estructural, ¡pero es muy complicado!

Árboles generales

```
data GenTree a = GNode a [GenTree a]
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
foldGT1 g f z (GNode x ts) =
        g x (foldr f z (map (foldGT1 g f z) ts))
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT1 = foldGT1 (+) (+) 0
```

- Árboles generales (o *Ro*
  - ¿Cómo sería la función *fold* para árboles generales?
    - Una versión intermedia, más satisfactoria

```
foldGT :: (a->c->b) -> ([b]->c) -> GTree a -> b
foldGT g k (GNode x ts) =
      g x (k (map (foldGT g k) ts))
```

Árboles generales

```
data GenTree a = GNode a [GenTree a]
foldGT0 h (GNode x ts) = h x (map (foldGT0 h) ts)
foldGT1 g f z (GNode x ts) =
        g x (foldr f z (map (foldGT1 g f z) ts))
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT1 = foldGT1 (+) (+) 0
```

- Árboles generales (o *Ro*
  - ¿Cómo sería la función *fold* para árboles generales?
    - Una versión intermedia, más satisfactoria

```
foldGT :: (a->c->b) -> ([b]->c) -> GTree a -> b
foldGT g k (GNode x ts) =
      g x (k (map (foldGT g k) ts))
```

    - La función **sumGT** se expresa como

```
sumGT' = foldGT (+) sum
```

    - Es fácil poner condiciones para que sea estructural
      - Si la función **k** es estructural, este **foldGT** también

Árboles generales

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT1 = foldGT1 (+) (+) 0
sumGT' = foldGT (+) sum
```

- Árboles generales (o *Rose Trees*)
  - Comparar las 3 versiones

```
foldGT0 :: (a->[b]->b)                    -> GTree a -> b
foldGT1 :: (a->c->b) -> (b->c->c) -> c    -> GTree a -> b
foldGT  :: (a->c->b) ->     ([b]->c)      -> GTree a -> b
```

    - Las tareas están repartidas diferente entre los parámetros
      - El parámetro de la primera hace todo
      - Los parámetros de la segunda delegan la recursión de listas
      - Los parámetros de la tercera se reparten el trabajo

Árboles generales

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT1 = foldGT1 (+) (+) 0
sumGT' = foldGT (+) sum
```

- Árboles generales (o *Rose Trees*)
  - Comparar las 3 versiones

```
foldGT0 h     (GNode x ts) =
        h x             (map (foldGT0 h)     ts)
foldGT  g k   (GNode x ts) =
        g x (     k     (map (foldGT g k)    ts))
foldGT1 g f z (GNode x ts) =
        g x (foldr f z (map (foldGT1 g f z) ts))
```

    - Las tareas están repartidas diferente entre los parámetros
    - Observar que `h x = g x . foldr f z = g x . k`

Árboles generales

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT1 = foldGT1 (+) (+) 0
sumGT' = foldGT (+) sum
```

- Árboles generales (o *Rose Trees*)
  - Comparar las 3 versiones

```
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT  = foldGT  (+) sum
sumGT1 = foldGT1 (\x n -> x + n)
                 (\nt nr -> nt + nr) 0   -- Argumentos del foldr
```

    - La 1ra tiene todo junto (`(+)` y `sum` en la misma función)
    - La 2da separa de forma intermedia (`(+)` y `sum` por su lado)
    - La 3ra tiene todo separado (`(+)` y las partes del `sum`)
      - Recordar que `sum = foldr (+) 0`

Árboles generales

```
data GenTree a = GNode a [GenTree a]
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT0 = foldGT0 (\x ns -> x + sum ns)
sumGT1 = foldGT1 (+) (+) 0
sumGT' = foldGT (+) sum
```

- Árboles generales (o *Rose Trees*)
  - Comparar las 3 versiones

```
sumGT0 = foldGT0 (flip ((+) . sum))
sumGT  = foldGT  (+) sum
sumGT1 = foldGT1 (+)
                 (+) 0   -- Argumentos del foldr
```

    - La 1ra tiene todo junto (`(+)` y `sum` en la misma función)
    - La 2da separa de forma intermedia (`(+)` y `sum` por su lado)
    - La 3ra tiene todo separado (`(+)` y las partes del `sum`)
      - Recordar que `sum = foldr (+) 0`

Árboles generales

```
data GenTree a = GNode a [GenTree a]
foldGT g k (GNode x ts) =
                    g x (k (map (foldGT g k) ts)
sumGT (GNode x ts) = x + sum (map sumGT ts)
sumGT' = foldGT (+) sum
```

- Árboles generales (o *Rose Trees*)
  - Otras funciones sobre árboles generales

```
mirrorGT = foldGT GNode reverse
depthGT = foldGT (\x d -> 1+d) (maxOr 0)
            where maxOr x [] = x
                  maxOr _ xs = maximum xs
```

    - La primera de las funciones es considerada difícil…
      - …cuando no se sigue el esquema
    - Como **`maximum`** es parcial, se le agrega un neutro
      - Posible porque las profundidades son mayor o igual que cero

Transformación de información

- Las técnicas vistas en el curso se trabajaron solamente para el aspecto de transformación de la información
  - Se consiguieron mejoras sustanciales de expresividad
  - Se pudieron demostrar propiedades no triviales
  - Se mejoró la comprensión de muchos conceptos
  - Dominamos el Tesseract (poder ilimitado ;))

Transformación de información

- Las técnicas vistas en el curso se trabajaron solamente para el aspecto de transformación de la información
- ¿Y se pueden utilizar para interacción con el medio?
  - Como es usual, hay que esperar a la próxima clase…

Resumen

Resumen

- Expresión como función del esquema de recursión estructural (*fold*) de tipos recursivos
  - Un parámetro por constructor (si la recursión es directa)
  - Otras variantes (si la recursión no es directa)
  - Se pueden mostrar propiedades generales
- Límites del sistema H-M: no es posible un *fold* general
  - Pero hay extensiones que sí lo permiten