

Programación Funcional

Clases teóricas

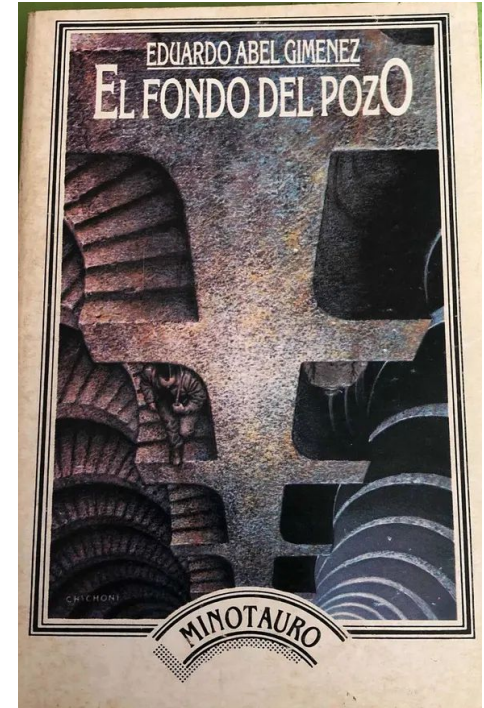
por Pablo E. “Fidel” Martínez López

6. Esquemas de recursión sobre otros tipos recursivos

“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



Repaso

Funciones de orden superior

- Funciones como valores (1)
 - Expresiones lambda
 - “Pasaje” de parámetros
 - Regla Beta
 - Convenciones de notación para sacar paréntesis
 - Currificación
 - “Hablar en francés”
 - Aplicación parcial

Funciones de orden superior

- Funciones como valores (2)
 - Funciones de orden superior
 - Funciones como parámetros
 - Funciones como datos
 - ¡Las funciones son estructuras de datos!
 - Expresiones lambda para construir, aplicación para usar
- Esquemas de funciones
 - `map`, `filter`, `elemBy`, `zipWith`, ...

Esquema de map

- Esquema de transformación elemento a elemento
- En inglés “to map” significa “asociar”, “relacionar”

```
map :: (a->b) -> [a] -> [b]
map f []          = []
map f (x:xs) = f x : tests xs

succs' = map (\n -> n+1)
uppers' = map (\c -> upper c)
tests' = map (\x -> x==0)
```

Observar el uso de
sección de operadores

```
-- map (+1)
```

```
-- map upper
```

```
-- map (==0)
```

Esquema de foldr

- Esquema de recursión estructural sobre listas
- En inglés “to fold” es “plegar” y la “r” es por “right”
- “Pliega” la lista desde la derecha, la “reduce”

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

```
sonCincos' = foldr (\n b -> n==5 && b) True
cantTotal' = foldr (\xs n -> length xs + n) 0
concat'    = foldr (\xs zs -> xs ++ zs) []
```

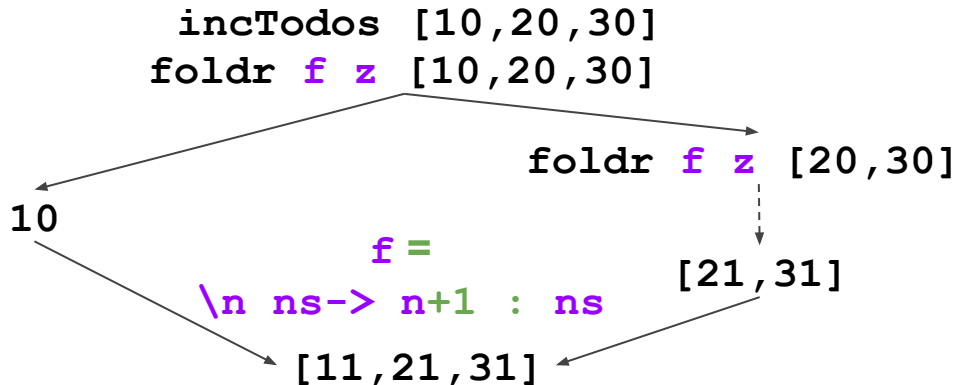
Esquema de foldr

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

■ Esquema de **foldr** = recursión estructural

■ Aplica el mismo razonam

```
incTodos :: [Int] -> [Int]  
incTodos = foldr (\n ns-> n+1 : ns) []
```



Esquema de foldr

- 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 este esquema
- Provee diversas ventajas:
 - Expresividad
 - Modularidad
 - Propiedades generales

Esquema de recr

Esquema de recursión primitiva

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

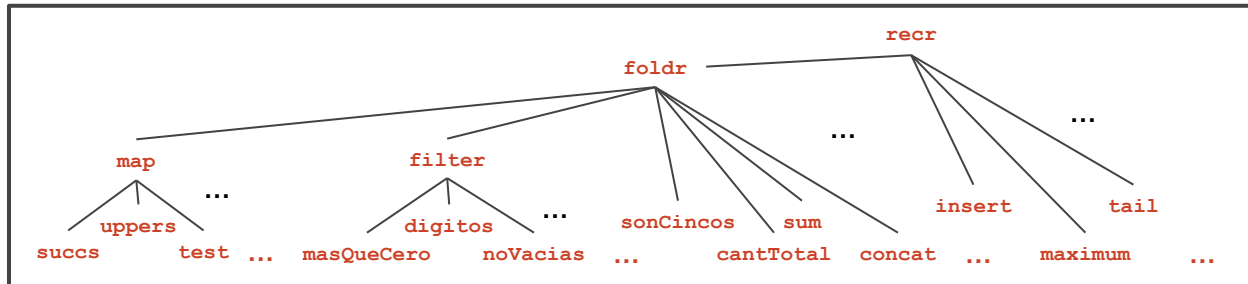
insert x = recr [x] (\y ys rs -> if x<y
                                then x:y:ys
                                else y:rs)

maximum = recr (error "") (\x xs m -> if null xs
                                       then x
                                       else max x m)
```

Acceder a la cola con otra auxiliar recursiva es mala práctica

Abstracción

- **Abstraer** es detectar similitudes (ignorando diferencias) y aprovecharlas (entendiendo lo que tienen en común cosas diferentes)
- Los parámetros son una forma sintáctica de expresar abstracción
- Es como “subir” y mirar desde mayor altura en forma vertical



Abstracción

- ❑ **Abstraer** es detectar similitudes (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)
```

- 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 :: ( -> ) -> ( -> -> -> ) -> 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

- 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) -> ( -> -> -> ) -> 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

- 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

$h :: \text{Arbol } a \rightarrow B$

$h = \text{foldA } f \ g$

where $f \ x = \dots x \dots_1$
 $g \ x \ r1 \ r2 = \dots x \dots r1 \dots r2 \dots_2$

```
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 violeta 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...

Hoja :: a -> ~~Arbol a~~^b

Nodo :: a -> ~~Arbol a~~^b -> ~~Arbol a~~^b -> ~~Arbol a~~^b

- ¿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!

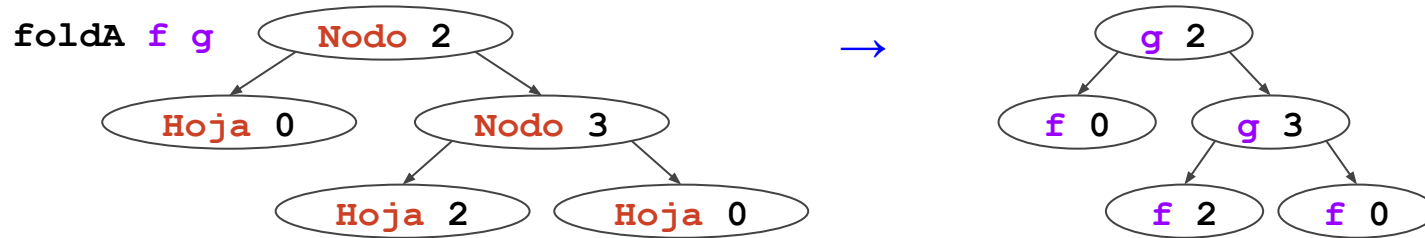
Árboles binarios

```
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
              | Nudo 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 (Nudo 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 plantea el esquema de recursión estructural
 - Observar que ahora solamente se escribe **foldA** porque esta función expresa ese esquema
 - ¿Cómo se evidencian 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 -> n) n
sumA (Nodo n t1 t2) = (\n n1 n2 -> n+n1+n2) n (sumA t1)
                                   (sumA t2)
```

Árboles binarios

```
data Arbol a = Hoja a
              | Nudo 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 (Nudo x t1 t2) =
    g x (foldA f g t1) (foldA f g t2)
```

Los árboles binarios son un tipo algebraico recursivo

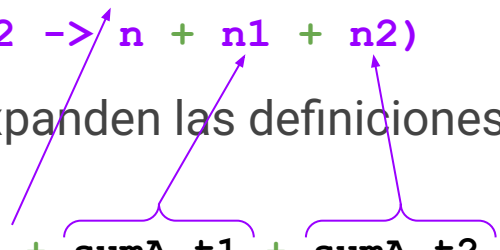
¿Cómo definir funciones usando **foldA**?

¡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 (Nudo 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         -> ...)
              (\x h1 h2 -> ... h1 ... h2 ...)

alturaA :: Arbol a -> Int   -- La altura del árbol
alturaA  = foldA (\x         -> ...)
              (\x a1 a2 -> ... a1 ... a2 ...)
```

■ Se plantea el esquema de recursión

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          -> ...)
              (\x h1 h2 -> h1 + h2)

alturaA :: Arbol a -> Int   -- La altura del árbol
alturaA = foldA (\x          -> ...)
               (\x a1 a2 -> 1 + max a1 a2)
```

■ Se resuelven los casos inductivos

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 altura del árbol
alturaA = foldA (\x          -> 0)
              (\x a1 a2 -> 1 + max a1 a2)
```

■ Se cierra con los casos base

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 altura 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)
              alturaNodo
```

```
where alturaNodo x a1 a2 = 1 + max a1 a2
```

■ La forma de escribir las expresiones puede complicar

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          -> ... n ...)
            (\n t1' t2' -> ... n ... t1' ... t2' ...)
```

■ Funciones de transformación de estructuras

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

- También hay un esquema de recursión primitiva

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

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

```
recA :: (a->b)
      -> (a->Arbol a->b->Arbol a->b->b)
      -> Arbol a -> b

recA f g (Hoja x)          = f x
recA f g (Nodo x t1 t2) = g x t1 (recA f g t1)
                           t2 (recA f g t2)
```

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

$h ::$		$ExpA \rightarrow B$
h	$(Cte\ x)$	$= \dots x \dots_1$
h	$(Suma\ e_1\ e_2)$	$= \dots h\ e_1 \dots h\ e_2 \dots_2$
h	$(Prod\ e_1\ e_2)$	$= \dots h\ e_1 \dots h\ e_2 \dots_3$

- ❏ ¿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 :: ( -> ) -> ( -> -> ) -> ( -> -> ) -> 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 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 aritméticas

```
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) = p (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 violetas proveen las conexiones

Expresiones aritméticas

```
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) = p (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 aritméticas

```
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) = p (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 aritméticas

```
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) = p (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

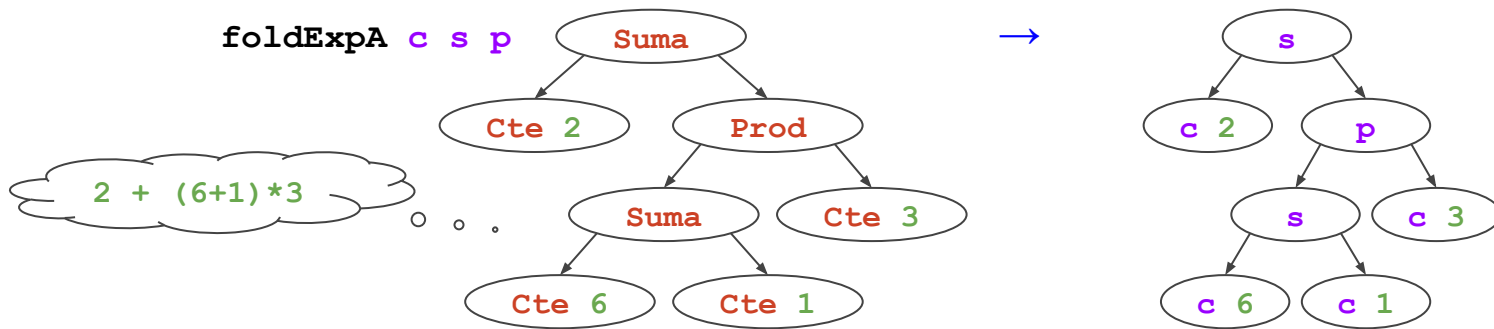
Expresiones aritméticas

```
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) = p (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 aritméticas

```
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) = p (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 -> ... n ...)
                  (\e1' e2' -> ... e1' ... e2' ...)
                  (\e1' e2' -> ... e1' ... e2' ...)
```

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

int2ta :: Int -> TA
int2ta = foldNat B A
```


Expresiones aritméticas

```
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) = p (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 TB = C TA | D TB TB
          | E TB TB

int2ta :: Int -> TA
int2ta = foldNat B A
```

Expresiones aritméticas

```
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) = p (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 TB = C TA | D TB TB
          | E TB TB

int2ta :: Int -> TA
int2ta = foldNat B A
```

Expresiones aritméticas

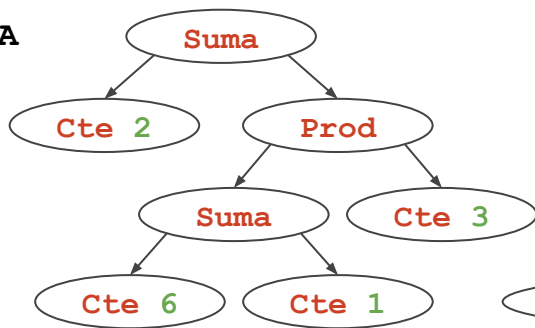
```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
foldExpA :: (Int->b) -> (b->b->b) -> (b->b->b)
                                         -> ExpA -> b
```

```
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) = p (foldExpA c s p e1)
                               (foldExpA c s p e2)
```

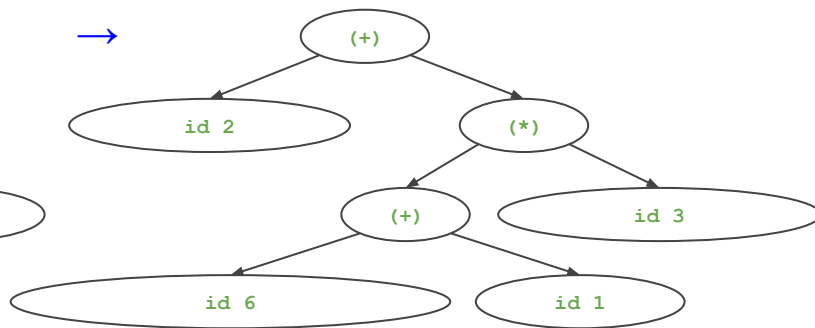
■ Otro tipo algebraico recursivo. expresiones aritméticas

■ ¿Cómo entender el tipo de los parámetros de **foldExpA**?

evalExpA



→



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

```
evalExpA (Suma (Cte 2) (Prod (Suma (Cte 6) (Cte 1)) (Cte 3)))
= (+) (id 2) ((*) ((+) (id 6) (id 1)) (id 3))
= (+) 2 ((*) ((+) 6 1) 3) = 2 + ((6 + 1) * 3) = 23
```

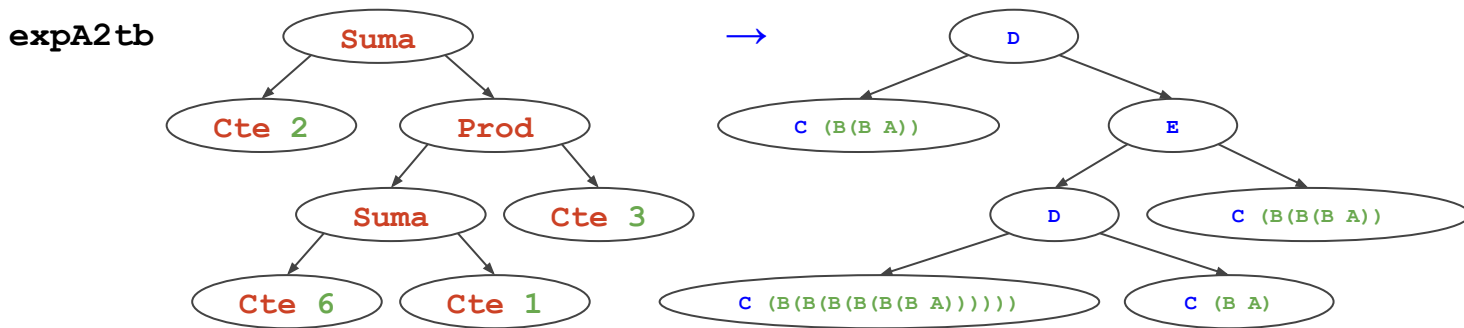
Expresiones aritméticas

```
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) = p (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 casos va a tener la definición de **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      CB                        = ...
foldTG      (CC g1 ch g2) = ...      (foldTG      g1) ch
                                         (foldTG      g2)
foldTG      (CD n g1)          = ... n (foldTG      g1)
foldTG      (CE g1 g2 g3) = ...      (foldTG      g1)
                                         (foldTG      g2)
                                         (foldTG      g3)
foldTG      (CF g ch)          = ...      (foldTG      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 :: ?? ->      ??      ->      ??
          ->      ??      ->      ??      -> 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

- 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?

```
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
```

Otros tipos recursivos

- 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
- Solamente el 2do y el último suman 1, porque tienen **Chars**

```
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
```

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 \mathbf{T} se puede expresar como función de orden superior (**foldT**)
- ❑ Para construirla
 - ❑ Se necesitan tantos parámetros como constructores tenga \mathbf{T}
 - ❑ El tipo de cada parámetro está vinculado al tipo del constructor correspondiente, cambiando \mathbf{T} por \mathbf{b}

Otros tipos recursivos

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

Otros tipos recursivos

- ❑ La recursión estructural de un tipo recursivo \mathbf{T} se puede expresar como función de orden superior (**foldT**)
- ❑ Para construirla
 - ❑ Se necesitan tantos parámetros como constructores tenga \mathbf{T}
 - ❑ El tipo de cada parámetro está vinculado al tipo del constructor correspondiente, cambiando \mathbf{T} por \mathbf{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 problemas

- Indicar si el estudiante está tomando alguno de los cursos

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

Estudiante -> Curso -> Bool



- 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 problemas

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

```
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 = snd . totalizar . 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

Resumen

Resumen

- ❑ Esquema de recursión estructural y primitiva para otros tipos recursivos
 - ❑ Árboles binarios (`foldA`, `recA`)
 - ❑ Expresiones Aritméticas (`foldExpA`, `recExpA`)
 - ❑ Otros (`TG`, `foldTG`, `recTG`)
 - ❑ Forma genérica de definir esquemas de recursión
- ❑ Uso de esquemas para programación