

Estructuras de datos

Clases teóricas

por Pablo E. “Fidel” Martínez López

2. Listas y recursión estructural

Listas

Listas

- Las listas son estructuras de datos predefinidas
 - Son similares a las listas de Gobstones
 - Pero hay algunas diferencias
 - La notación `[1,2,3]` es abreviatura de `1:2:3:[]`
 - Las operaciones de `(:)` y `[]` son constructores
 - La operación de agregar `(++)` no es predefinida
 - Así, son equivalentes

<code>[1,2,3]</code>	<code>1:2:3:[]</code>	<code>1:[2,3]</code>
<code>[1]++[2,3]</code>	<code>(1:[])++(2:[3])</code>	<code>(1:[])++(2:3:[])</code>

Listas

- Las listas son estructuras de datos predefinidas
 - $(:)$:: $a \rightarrow [a] \rightarrow [a]$
 - Se lee *cons* (o *agregar adelante*)
 - Toma un elemento de algún tipo y una lista del mismo tipo, y describe la lista que resulta de agregar el elemento dado adelante de la lista dada
 - $[]$:: $[a]$
 - Se lee *nil* (o *lista vacía*)
 - Es una lista de algún tipo a instanciar

Listas

- Los constructores pueden usarse para *pattern matching*

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

```
null :: [a] -> Bool
```

```
null [] = True
```

```
null _ = False
```

```
noEsVacía :: [a] -> Bool
```

```
noEsVacía (_:_) = True
```

```
noEsVacía _ = False
```

- Las variables de lista se suelen terminar con la letra s
- Permite indicar que son muchos elementos de ese tipo

Listas

- ❏ Los constructores pueden usarse para *pattern matching*
- ❏ Recordar: **head** y **tail** son operaciones parciales
 - ❏ No están definidas para la lista vacía
 - ❏ Podrían definirse indicando explícitamente el error

```
head :: [a] -> a
head (x:_) = x
head _     = error "head no se puede usar con []"
```

Listas

- ❑ Los constructores pueden usarse para *pattern matching*
- ❑ Pero la función de agregar (++) NO es constructora
 - ❑ Significa que NO se puede hacer *pattern matching* con ella
 - ❑ O sea, NO se puede hacer esto:

f (xs++ys) = ...

- ❑ Es un error de construcción

Listas

- ❑ Los constructores pueden usarse para *pattern matching*
- ❑ Pero la función de agregar (++) NO es constructora
 - ❑ Significa que NO se puede hacer *pattern matching* con ella
 - ❑ O sea, NO se puede hacer esto:

`f (xs++ys) = ...`

¡¡MAL!!

- ❑ Es un error de construcción

Listas

- ❏ Los constructores pueden usarse para *pattern matching*
- ❏ Más ejemplos de *pattern matching* válidos sobre listas:

```
esSingular :: [a] -> Bool
esSingular (_:[]) = True
esSingular _      = False

segundo :: [a] -> a
segundo (_:x:_) = x
segundo _       = error "No hay 2 elementos"
```

Listas y recursión

- ❑ ¿Cómo hacer recorridos con listas?
 - ❑ En Gobstones se realizan recorridos con repetición condicional (**while**) o repetición indexada (**foreach**)
 - ❑ Se considera de a un elemento por vez
 - ❑ Se repite hasta que la lista está vacía
 - ❑ Pero ambos son comandos
 - ❑ Y Haskell no tiene comandos...
 - ❑ Se hace necesario otra forma de realizar el recorrido

Listas y recursión

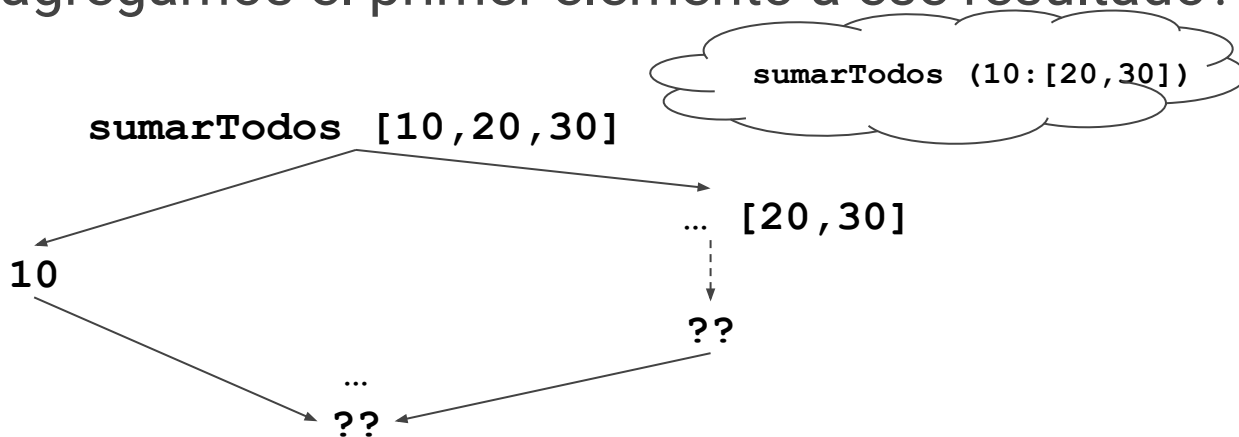
- ❏ ¿Cómo hacer recorridos con listas?
- ❏ En Haskell se utiliza **recursión (estructural)**
- ❏ ¿En qué consiste la recursión?
 - ❏ Empecemos con los dos casos de listas...

```
sumarTodos :: [Int] -> Int
sumarTodos []      = ...
sumarTodos (n:ns) = ... n ... ns ...
```

Listas y recursión

```
sumarTodos :: [Int] -> Int
sumarTodos []      = ...
sumarTodos (n:ns) = ... n ... ns ...
```

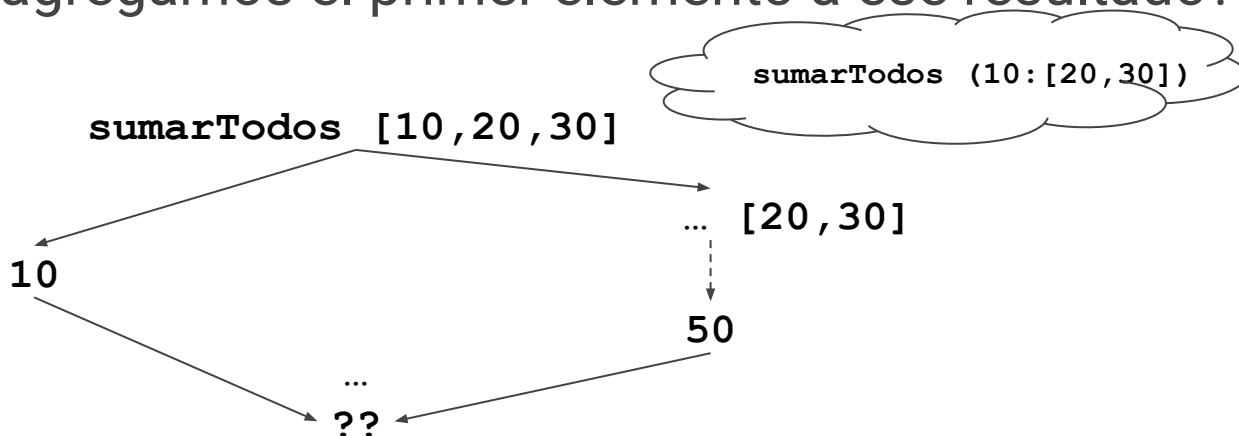
- ¿Cómo pensar la recursión?
- ¡NO intentar ejecutarla!
- Si podemos encontrar el resultado de la cola...
- ...¿cómo agregamos el primer elemento a ese resultado?



Listas y recursión

```
sumarTodos :: [Int] -> Int
sumarTodos []      = ...
sumarTodos (n:ns) = ... n ... ns ...
```

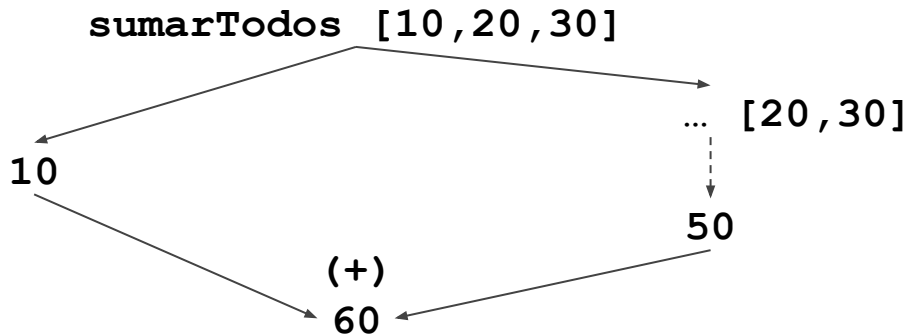
- ¿Cómo pensar la recursión?
- ¡NO intentar ejecutarla!
- Si podemos encontrar el resultado de la cola...
- ...¿cómo agregamos el primer elemento a ese resultado?



Listas y recursión

```
sumarTodos :: [Int] -> Int
sumarTodos []      = ...
sumarTodos (n:ns) = n + ... ns
```

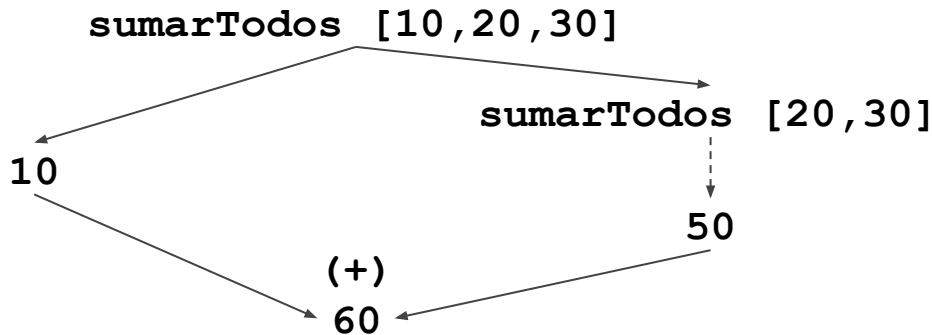
- ❏ ¿Cómo pensar la recursión?
- ❏ En este caso hay que sumar el primero
- ❏ ¿Y cómo obtenemos el resultado de la cola?



Listas y recursión

```
sumarTodos :: [Int] -> Int
sumarTodos []      = ...
sumarTodos (n:ns) = n + sumarTodos ns
```

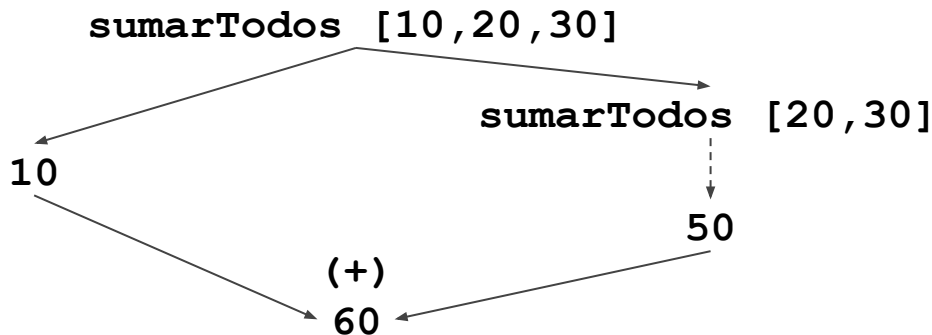
- ¿Cómo pensar la recursión?
- En este caso hay que sumar el primero
- ¿Y cómo obtenemos el resultado de la cola?
- ¡Con la misma función!



Listas y recursión

```
sumarTodos :: [Int] -> Int
sumarTodos []      = 0
sumarTodos (n:ns) = n + sumarTodos ns
```

- ¿Cómo pensar la recursión?
- ¡Solamente es necesario pensar cómo agregar el primer elemento al resultado final!
- Y no hay que olvidar el caso base, para []
 - ¿Por qué es 0?



Recursión estructural sobre listas

- ❏ Los recorridos sobre listas con recursión estructural
 - ❏ Siempre tienen 2 casos: caso base y caso recursivo
 - ❏ En el caso recursivo se utiliza **la misma función** que se define **sobre una parte** de la lista original

```
sumarTodos :: [Int] -> Int
sumarTodos []      = 0
sumarTodos (n:ns) = n + sumarTodos ns
```

- ❏ Se llama **recursión estructural** sobre listas porque sigue el mismo esquema que la estructura de listas

Recursión estructural sobre listas

- Recursión estructural sobre listas
 - Se reemplazan los constructores por operaciones

```
sumarTodos [10,20,30]
=
sumarTodos (10 : (20 : (30 : [])))
=
      10 + (20 + (30 + 0))
=
      10 + 50
=
      60
```

¡La estructura de la cuenta es similar a la estructura de la lista!

Recursión estructural

```
sumarTodos :: [Int] -> Int
sumarTodos []      = 0
sumarTodos (n:ns) = n + sumarTodos ns
```

■ Recursión estructural sobre listas

■ Compararlo con Gobstones

```
function sumarTodos(números) {
  sumaHastaAhora := 0
  foreach número in números {
    sumaHastaAhora := número + sumaHastaAhora
  }
  return(sumaHastaAhora)
}
```

Recursión estructural

```
sumarTodos :: [Int] -> Int
sumarTodos []      = 0
sumarTodos (n:ns) = n + sumarTodos ns
```

■ Recursión estructural sobre listas

■ Si Gobstones tuviese recursión, se podría hacer

```
function sumarTodos(números) {
  return(choose 0                when isEmpty(números)
              primero(números)
              + sumarTodos(resto(números))
              otherwise)
}
```

■ Pero Gobstones no tiene recursión :(

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente

`longitud :: ...`

`agregar :: ...`

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo

```
longitud :: ...  
longitud []      = ...  
longitud (x:xs) = ...
```

```
agregar :: ...  
agregar []      ys = ...  
agregar (x:xs) ys = ...
```

Recursión estructural sobre listas

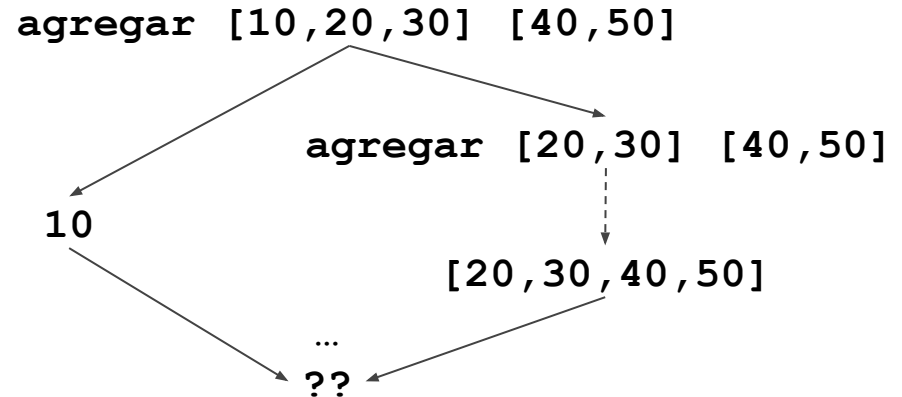
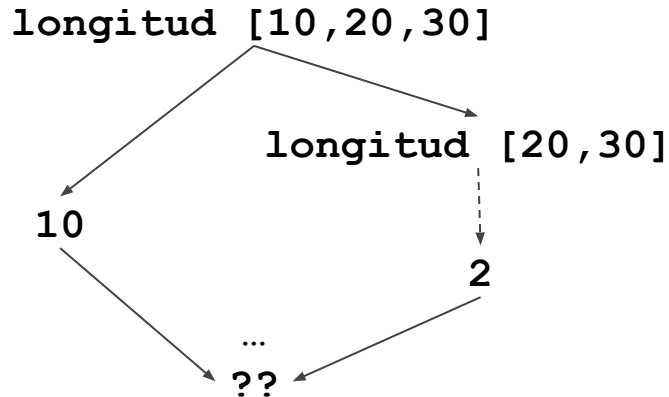
- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: ...  
longitud []      = ...  
longitud (x:xs) = ... x ... longitud xs ...
```

```
agregar :: ...  
agregar []      ys = ...  
agregar (x:xs) ys = ... x ... agregar xs ys ...
```

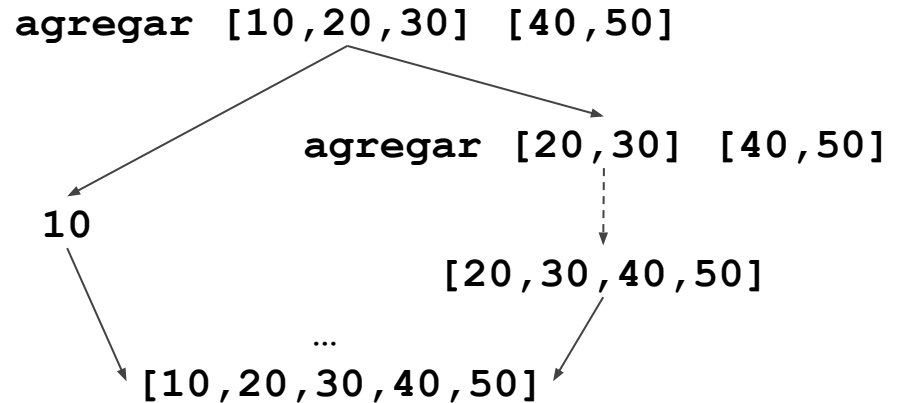
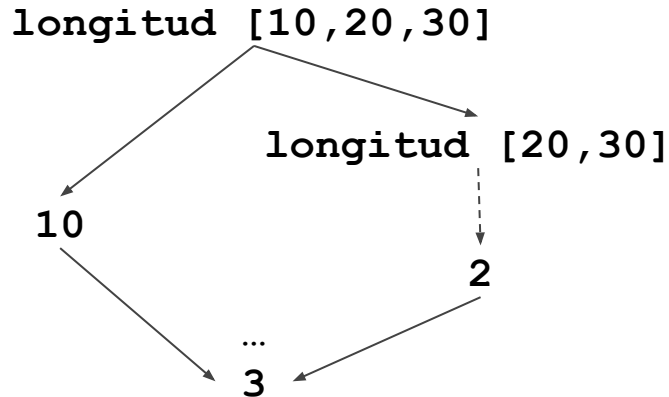
Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola



Lista Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola



Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: ...  
longitud []      = ...  
longitud (x:xs) = 1 + longitud xs
```

```
agregar :: ...  
agregar []      ys = ...  
agregar (x:xs) ys = ... x ... agregar xs ys ...
```

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: ...  
longitud []      = 0  
longitud (x:xs) = 1 + longitud xs
```

```
agregar :: ...  
agregar []      ys = ...  
agregar (x:xs) ys = ... x ... agregar xs ys ...
```

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: [ ... ] -> Int
longitud []      = 0
longitud (x:xs) = 1 + longitud xs
```

```
agregar :: ...
agregar []      ys = ...
agregar (x:xs) ys = ... x ... agregar xs ys ...
```

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: [ a ] -> Int
longitud []      = 0
longitud (x:xs) = 1 + longitud xs
```

```
agregar :: ...
agregar []      ys = ...
agregar (x:xs) ys = ... x ... agregar xs ys ...
```

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: [ a ] -> Int
longitud []      = 0
longitud (x:xs) = 1 + longitud xs

agregar :: ...
agregar []      ys = ...
agregar (x:xs)  ys = x : agregar xs ys
```

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: [ a ] -> Int
longitud []      = 0
longitud (x:xs) = 1 + longitud xs

agregar :: ...
agregar []      ys = ys
agregar (x:xs)  ys = x : agregar xs ys
```

Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: [ a ] -> Int
longitud []      = 0
longitud (x:xs) = 1 + longitud xs

agregar :: [ ... ] -> [ ... ] -> [ ... ]
agregar []      ys = ys
agregar (x:xs)  ys = x : agregar xs ys
```


Recursión estructural sobre listas

- Más funciones que recorren listas recursivamente
 - Recordar usar dos casos: base y recursivo
 - En el recursivo, usar la misma función en la cola

```
longitud :: [ a ] -> Int
longitud []      = 0
longitud (x:xs) = 1 + longitud xs

agregar :: [ a ] -> [ a ] -> [ a ]
agregar []      ys = ys
agregar (x:xs)  ys = x : agregar xs ys
```

Recursión estructural sobre listas

■ Más funciones que recorren listas recursivamente

```
hayAlMenosUnCinco :: [Int] -> Bool
  -- hayAlMenosUnCinco [10,20,30] = False
  -- hayAlMenosUnCinco [10,5,30] = True

hayAlMenosUn :: Int -> [Int] -> Bool
  -- Similar, pero toma el número por parámetro

soloLosMayoresQue :: Int -> [Int] -> [Int]
  -- soloLosMayoresQue 25 [10,30,20,40] = [30,40]

iniciales :: [Dir] -> [Char]
  -- iniciales [Norte,Sur] = ['N','S']

zip :: [a] -> [b] -> [(a,b)]
  -- zip [10,20,30] ['N','S'] = [(10,'N'),(20,'S')]
```

Recursión estructural sobre listas

- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
```

Recursión estructural sobre listas

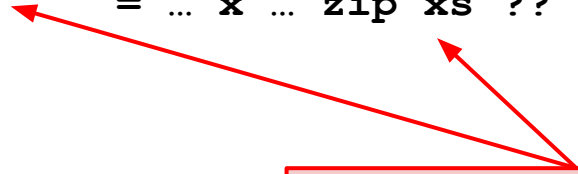
- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
zip []      _      = ...
zip (x:xs) ??    = ... x ... zip xs ?? ...
```

Recursión estructural sobre listas

- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]  
zip []      _      = ...  
zip (x:xs) ??    = ... x ... zip xs ?? ...
```

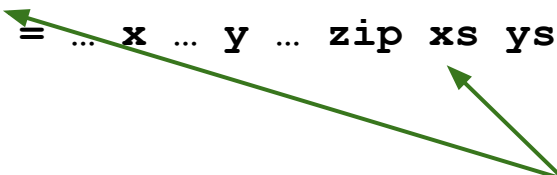


Se precisa más información
sobre el segundo argumento

Recursión estructural sobre listas

- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = ...
zip (x:xs) [] = ...
zip (x:xs) (y:ys) = ... x ... y ... zip xs ys ...
```



¡Una recursión dentro de otra!

Recursión estructural sobre listas

- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
zip []      _      = ...
zip (x:xs) []      = ...
zip (x:xs) (y:ys) = (x,y) ... zip xs ys
```

Recursión estructural sobre listas

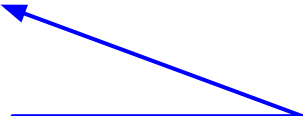
- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _ = ...
zip (x:xs) [] = ...
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```


Recursión estructural sobre listas

- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
zip [] _          = []
zip (x:xs) []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

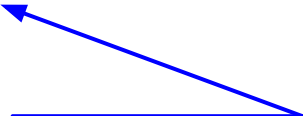


¡Las recursiones NO son simultáneas, sino anidadas!

Recursión estructural sobre listas

- Recursiones adentro de recursiones
 - Hay funciones recursivas que requieren más análisis
 - Usualmente involucran 2 listas a juntar elemento a elemento

```
zip :: [a] -> [b] -> [(a,b)]
zip []      _      = []
zip _      []      = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```



¡Las recursiones NO son simultáneas, sino anidadas!

Recursión estructural sobre listas

- ❑ ¿Todas las funciones sobre listas son recursivas?
 - ❑ Hay funciones recursivas que no son estructurales
 - ❑ Hay funciones que NO se pueden hacer por recursión
 - ❑ Intentemos...

```
last :: [ a ] -> a
```

```
promedio :: [ Int ] -> Int
```

Recursión estructural sobre listas

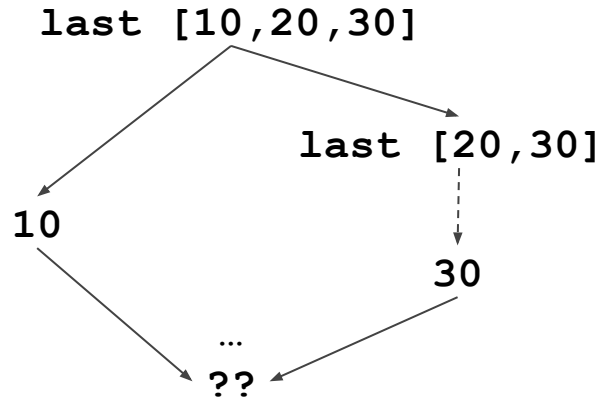
- ¿Todas las funciones sobre listas son recursivas?
 - Hay funciones recursivas que no son estructurales
 - Hay funciones que NO se pueden hacer por recursión
 - Intentemos...

```
last :: [ a ] -> a
last []      = ...
last (x:xs) = ... x ... last xs ...
```

```
promedio :: [ Int ] -> Int
promedio []      = ...
promedio (n:ns) = ... n ... promedio ns ...
```

Recursión estructural sobre listas

- ¿Todas las funciones sobre listas son recursivas?
 - Caso 1:** recursivas no exactamente estructurales



Recursión estructural sobre listas

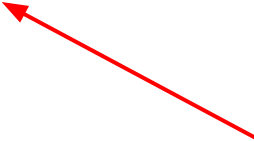
- ❑ ¿Todas las funciones sobre listas son recursivas?
- ❑ **Caso 1:** recursivas no exactamente estructurales
 - ❑ ¿Qué pasa con el caso base?

```
last :: [ a ] -> a
last []      = ...
last (x:xs) = last xs
```

Recursión estructural sobre listas

- ❑ ¿Todas las funciones sobre listas son recursivas?
- ❑ **Caso 1:** recursivas no exactamente estructurales
 - ❑ ¿Qué pasa con el caso base?
 - ❑ ¿Se puede usar el caso inductivo así no más?

```
last :: [ a ] -> a    -- PRECOND: la lista no es vacía  
last []              = error "No hay elementos"  
last (x:xs) = last xs
```



¿Qué pasa si **xs** es []?

Recursión estructural sobre listas

- ❑ ¿Todas las funciones sobre listas son recursivas?

- ❑ **Caso 1:** recursivas no exactamente estructurales

- ❑ ¿Se puede usar el caso inductivo así no más?

- ❑ No. Requiere ajustarse, para evitar la parcialidad

```
last :: [ a ] -> a    -- PRECOND: la lista no es vacía
last []              = error "No hay elementos"
last (x:xs) = if null xs
               then ...
               else last xs
```


Recursión estructural sobre listas

❑ ¿Todas las funciones sobre listas son recursivas?

❑ **Caso 1:** recursivas no exactamente estructurales

❑ ¿Se puede usar el caso inductivo así no más?

❑ No. Requiere ajustarse, para evitar la parcialidad

```
last :: [ a ] -> a    -- PRECOND: la lista no es vacía
last []              = error "No hay elementos"
last (x:xs) = if null xs
               then x
               else last xs
```

Recursión estructural sobre listas

- ❑ ¿Todas las funciones sobre listas son recursivas?

- ❑ **Caso 1:** recursivas no exactamente estructurales

- ❑ Requiere ajustarse, para evitar la parcialidad

- ❑ ¡Este ajuste puede quedar escondido en la sintaxis!

```
last :: [ a ] -> a    -- PRECOND: la lista no es vacía
last []              = error "No hay elementos"
last (x:[])          = x
last (_:xs)          = last xs
```

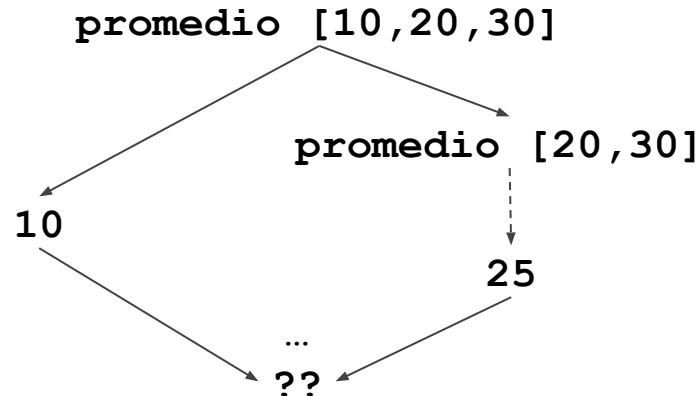
Recursión estructural sobre listas

- ❑ ¿Todas las funciones sobre listas son recursivas?
- ❑ **Caso 2:** no recursivas

```
promedio :: [ Int ] -> Int
promedio []      = ...
promedio (n:ns) = ... n ... promedio ns ...
```

Recursión estructural sobre listas

- ❏ ¿Todas las funciones sobre listas son recursivas?
- ❏ **Caso 2:** no recursivas
 - ❏ ¿Cómo recuperar la cantidad de elementos desde el promedio de la cola?



Recursión estructural sobre listas

❑ ¿Todas las funciones sobre listas son recursivas?

❑ **Caso 2:** no recursivas

❑ ¿Cómo recuperar la cantidad de elementos desde el promedio de la cola?

```
promedio :: [ Int ] -> Int  
promedio []          = ...  
promedio (n:ns) = ... n ... promedio ns ...
```

Recursión estructural sobre listas

- ❑ ¿Todas las funciones sobre listas son recursivas?

- ❑ **Caso 2:** no recursivas

- ❑ No hay una solución recursiva razonable

- ❑ ¡Es necesario usar funciones auxiliares!

```
promedio :: [ Int ] -> Int
-- PRECOND: la lista no es vacía
promedio ns = div (sumarTodos ns) (longitud ns)
```

Recursión sobre números

Recursión sobre números

- ¿Cómo se realizan funciones recursivas sobre números?
 - Se sigue el mismo principio: recursión “estructural”
 - La “parte recursiva” de un número es su anterior
 - El caso base es el 0
 - No se aplica a números negativos

`f :: Int -> b`

`f 0 = ...`

`f n = ... f (n-1) ...`

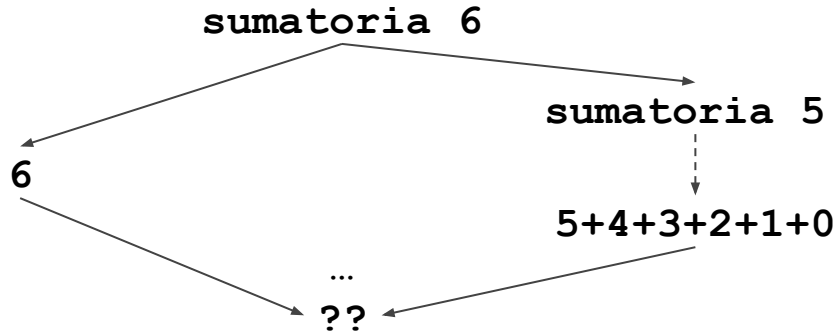
Recursión sobre números

- Recursión estructural sobre números
 - **Ejemplo:** sumatoria de 0 a un número dado
 - Primero se plantea el esquema recursivo

```
sumatoria :: Int -> Int
sumatoria 0 = ...
sumatoria n = ... sumatoria (n-1) ...
```

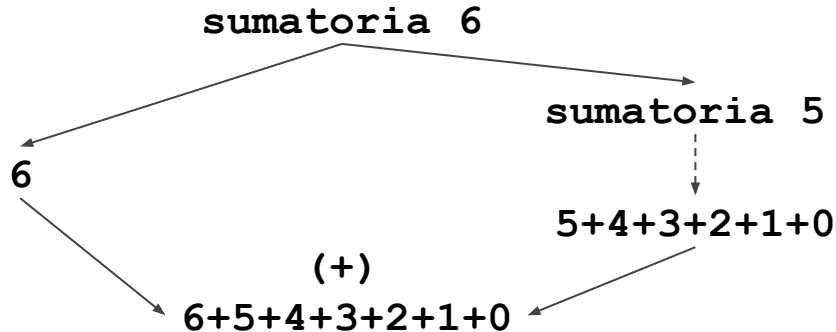
Recursión sobre números

- Recursión estructural sobre números
 - Ejemplo:** sumatoria de 0 a un número dado
 - Primero se plantea el esquema recursivo



Recursión sobre números

- Recursión estructural sobre números
 - Ejemplo:** sumatoria de 0 a un número dado
 - Luego se resuelve el caso inductivo



Recursión sobre números

- Recursión estructural sobre números
 - **Ejemplo:** sumatoria de 0 a un número dado
 - Luego se resuelve el caso inductivo

```
sumatoria :: Int -> Int
sumatoria 0 = ...
sumatoria n = n + sumatoria (n-1)
```

Recursión sobre números

- Recursión estructural sobre números
 - **Ejemplo:** sumatoria de 0 a un número dado
 - Finalmente se resuelve el caso base

```
sumatoria :: Int -> Int
sumatoria 0 = 0
sumatoria n = n + sumatoria (n-1)
```

Recursión estructural sobre listas

■ Más funciones recursivas sobre números

```
replicar :: Int -> a -> [a]
  -- replicar 2 'K' = ['K','K']

cuentaRegresivaDesde :: Int -> [Int]
  -- cuentaRegresivaDesde 5 = [5,4,3,2,1,0]

losPrimerosN :: Int -> [a] -> [a]
  -- losPrimerosN 7 [10..50] = [10,11,12,13,14,15,16]
```

Recursión sobre números

- Recursión estructural sobre números
 - Otra recursión anidada (de diferentes estructuras)
 - Primero se plantea el esquema recursivo de números

```
losPrimerosN :: Int -> [a] -> [a]
losPrimerosN 0 xs = ...
losPrimerosN n xs = ... losPrimerosN (n-1) ?? ...
```

Recursión sobre números

- ❑ Recursión estructural sobre números
 - ❑ Otra recursión anidada (de diferentes estructuras)
 - ❑ Y se observa que en el 2do caso hace falta otra recursión

```
losPrimerosN :: Int -> [a] -> [a]
losPrimerosN 0 xs = ...
losPrimerosN n xs = if null xs
                    then ...
                    else ... losPrimerosN (n-1) (tail xs) ...
```


Recursión sobre números

- Recursión estructural sobre números
 - Otra recursión anidada (de diferentes estructuras)
 - Se reescribe por comodidad

```
losPrimerosN :: Int -> [a] -> [a]
losPrimerosN 0 _          = ...
losPrimerosN _ []         = ...
losPrimerosN n (x:xs) = ... x ... losPrimerosN (n-1) xs ...
```

Recursión sobre números

- Recursión estructural sobre números
 - Otra recursión anidada (de diferentes estructuras)
 - Se completa el caso inductivo

```
losPrimerosN :: Int -> [a] -> [a]
losPrimerosN 0 _          = ...
losPrimerosN _ []         = ...
losPrimerosN n (x:xs) = x : losPrimerosN (n-1) xs
```

Recursión sobre números

- Recursión estructural sobre números
 - Otra recursión anidada (de diferentes estructuras)
 - Y finalmente los casos base

```
losPrimerosN :: Int -> [a] -> [a]
losPrimerosN 0 _          = []
losPrimerosN _ []         = []
losPrimerosN n (x:xs) = x : losPrimerosN (n-1) xs
```

Resumen

Resumen

- ❑ Listas
 - ❑ Son un tipo algebraico con constructores: `[]` y `(:)` (nil y cons)
- ❑ Recursión estructural sobre listas
 - ❑ Sigue la estructura de las listas (dos casos)
 - ❑ El caso recursivo usa *la misma función que se está definiendo*, pero sobre la parte recursiva
- ❑ Casos donde no alcanza la recursión estructural

Resumen

- Recursión estructural sobre números
 - Dos casos: 0 y mayor que 0 (no para negativos)
 - El caso recursivo usa *la misma función que se está definiendo*, pero sobre el *número anterior*