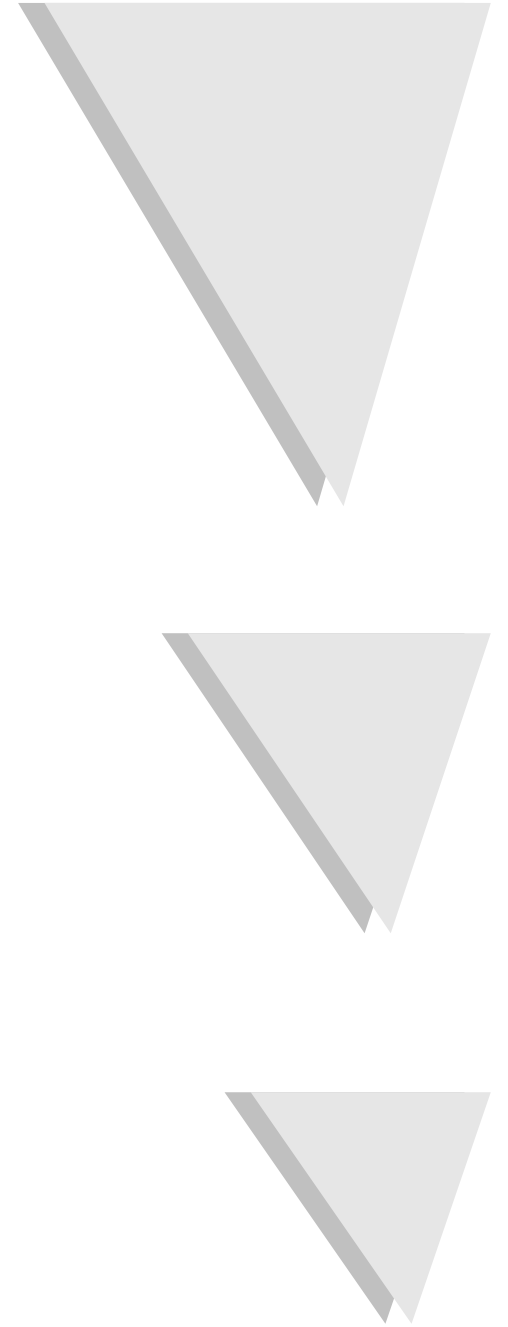




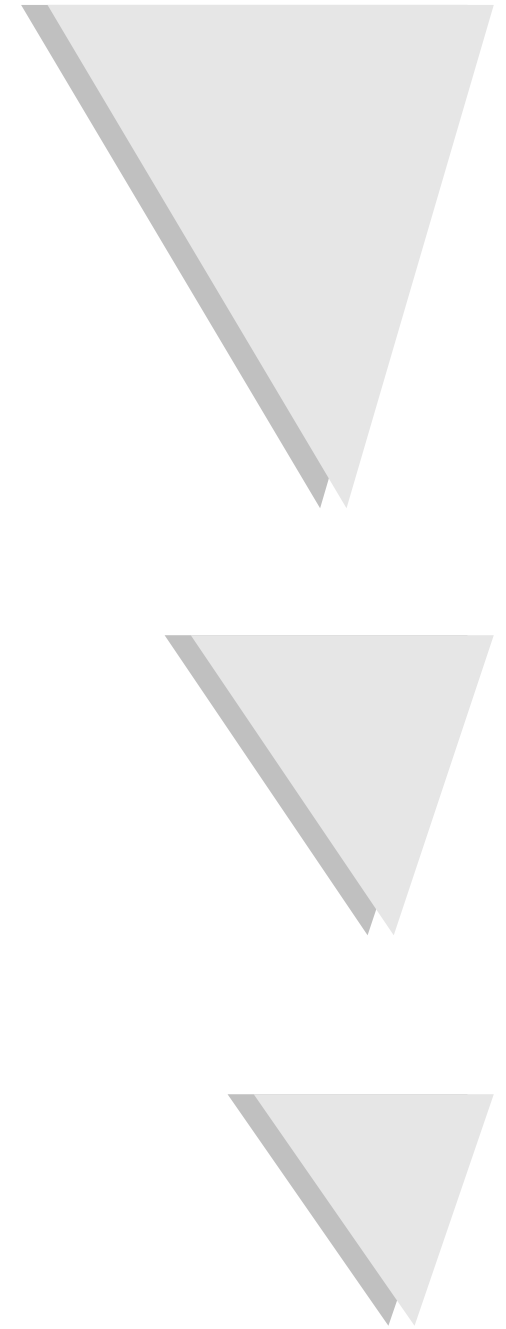
# **PROGRAMACIÓN FUNCIONAL**

## **Tipos de Datos: Tipos Recursivos**



# Tipos de Datos

- ◆ Tipos algebraicos recursivos
- ◆ Listas
- ◆ Árboles
- ◆ Funciones sobre tipos recursivos
- ◆ Lenguaje Imperativo Simple (LIS)
- ◆ Evaluación de un programa LIS



# Tipos de Datos Recursivos

- ◆ Un tipo algebraico recursivo
  - ◆ tiene al menos uno de los constructores con el tipo que se define como argumento
  - ◆ es la concreción en Haskell de un conjunto definido inductivamente
- ◆ Ejemplos:
  - `data N = Z | S N`
  - `data BE = TT | FF | AA BE BE | NN BE`
- ◆ ¿Qué elementos tienen estos tipos?

# Tipos de Datos Recursivos

- ◆ Cada constructor define un caso de una definición inductiva de un conjunto.
  - ◆ Si tiene al tipo definido como argumento, es un *caso inductivo*, si no, es un *caso base*.
- ◆ El pattern matching
  - ◆ provee análisis de casos
  - ◆ permite acceder a los elementos inductivos que forman a un elemento dado
- ◆ Por ello, se pueden definir funciones recursivas

# Tipos de Datos Recursivos

◆ Ejemplo:  $\text{data } N = Z \mid S \ N$

$\text{size} :: N \rightarrow \text{Int}$

$\text{size } Z = 0$

$\text{size } (S \ x) = 1 + \text{size } x$

$\text{addN} :: N \rightarrow N \rightarrow N$

$\text{addN } Z \ m = m$

$\text{addN } (S \ n) \ m = S \ (\text{addN } n \ m)$

◆ ¿Puede probar la siguiente propiedad?

Sean  $n, m :: N$  finitos, cualesquiera; entonces

$\text{size } (\text{addN } n \ m) = \text{size } n + \text{size } m$

# Tipos de Datos Recursivos

- ◆ Se pueden probar propiedades por inducción estructural (pues representan conjuntos inductivos)
- ◆ **Demostración:** por inducción en la estructura de  $n$ 
  - ◆ Caso base:  $n = Z$ 
    - ◆ Usar `addN.1`, `0` neutro de `(+)` y `size.1`
  - ◆ Caso inductivo:  $n = S\ n'$
  - ◆ HI:  $\text{size}(\text{addN}\ n'\ m) = \text{size}\ n' + \text{size}\ m$ 
    - ◆ Usar `addN.2`, `size.2`, HI, asociatividad de `(+)`, y `size.2`

# Listas

- Una definición equivalente a la de listas  
 $\text{data List } a = \text{Nil} \mid \text{Cons } a \text{ (List } a)$
- La sintaxis de listas es equivalente a la de esta definición:
  - $[]$  es equivalente a Nil
  - $(x:xs)$  es equivalente a  $(\text{Cons } x \text{ xs})$
- Sin embargo,  $(\text{List } a)$  y  $[a]$  son tipos distintos

# Listas

- Considerar las definiciones

$(++) :: [a] \rightarrow [a] \rightarrow [a]$

$[] ++ ys = ys$

$(x:xs) ++ ys = x : (xs ++ ys)$

$sum :: [Int] \rightarrow Int$

$sum [] = 0$

$sum (n:ns) = n + sum ns$

- Demostrar que para todo par  $xs, ys$  de listas finitas, vale que:

$sum (xs ++ ys) = sum xs + sum ys$



# Listas

- ◆ **Propiedad:** para todo par de  $xs$ ,  $ys$  de listas finitas  
 $\text{sum } (xs ++ ys) = \text{sum } xs + \text{sum } ys$
- ◆ **Demostración:** por inducción en  $xs$ 
  - ◆ Caso base:  $xs = []$ 
    - ◆ Usar  $(++)$ .1, 0 neutro de  $(+)$  y  $\text{sum}$ .1
  - ◆ Caso inductivo:  $xs = (x:xs')$
  - ◆ HI:  $\text{sum } (xs' ++ ys) = \text{sum } xs' + \text{sum } ys$ 
    - ◆ Usar  $(++)$ .2,  $\text{sum}$ .2, HI, asoc. de  $(+)$  y  $\text{sum}$ .2

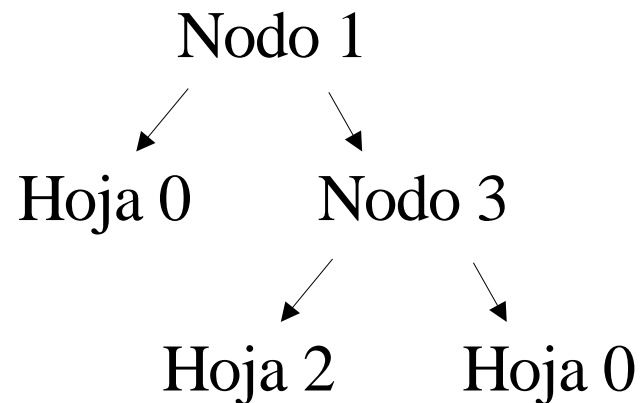
# Árboles

- Un árbol es un tipo algebraico tal que al menos un elemento compuesto tiene dos componentes inductivas
- Se pueden usar TODAS las técnicas vistas para tipos algebraicos y recursivos
- Ejemplo:  
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
- ¿Qué elementos tiene el tipo (Arbol Int)?

# Árboles

- Si representamos elementos de tipo Arbol Int mediante diagramas jerárquicos

aej = Nodo 1 (Hoja 0)  
(Nodo 3 (Hoja 2) (Hoja 0))



# Árboles

- ◆ ¿Cuántas hojas tiene un (Arbol a)?

hojas :: Arbol a -> Int

hojas (Hoja x) = 1

hojas (Nodo x t1 t2) = hojas t1 + hojas t2

- ◆ ¿Y cuál es la altura de un (Arbol a)?

altura :: Arbol a -> Int

altura (Hoja x) = 0

altura (Nodo x t1 t2) = 1 + (altura t1 `max` altura t2)

- ◆ ¿Puede mostrar que para todo árbol finito a,  
hojas a  $\leq 2^{(altura\ a)}$ ? ¿Cómo?

# Árboles

- ◆ ¿Cómo reemplazamos una hoja?
- ◆ Ej: Cambiar los 2 en las hojas por 3.  
cambiar2 :: Arbol Int -> Arbol Int  
cambiar2 (Hoja n) = if n==2  
                          then Hoja 3  
                          else Hoja n  
cambiar2 (Nodo n t1 t2) =  
                          Nodo n (cambiar2 t1) (cambiar2 t2)
- ◆ ¿Cómo trabaja cambiar2? Reducir (cambiar2 aej)

# Árboles

- ◆ Más funciones sobre árboles

`duplA :: Arbol Int -> Arbol Int`

`duplA (Hoja n) = Hoja (n*2)`

`duplA (Nodo n t1 t2) =  
    Nodo (n*2) (duplA t1) (duplA t2)`

`sumA :: Arbol Int -> Int`

`sumA (Hoja n) = n`

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

- ◆ ¿Cómo evalúa la expresión `(duplA aej)`?

- ◆ ¿Y `(sumA aej)`?

# Árboles

## ➤ Recorridos de árboles

`inOrder, preOrder :: Arbol a -> [ a ]`

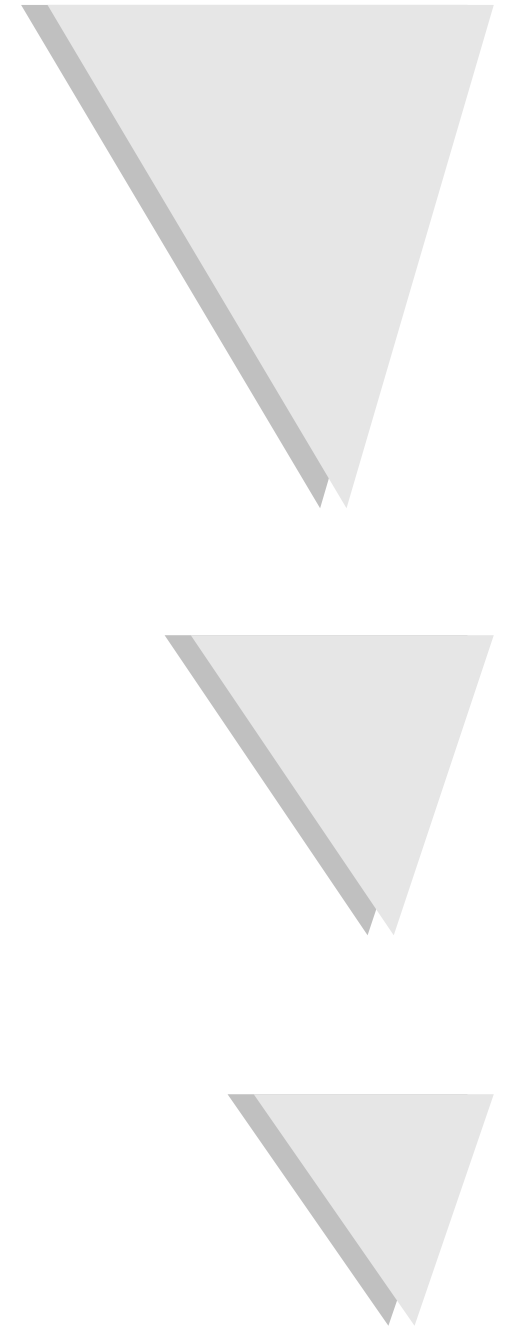
`inOrder (Hoja n) = [ n ]`

`inOrder (Nodo n t1 t2) =  
 inOrder t1 ++ [ n ] ++ inOrder t2`

`preOrder (Hoja n) = [ n ]`

`preOrder (Nodo n t1 t2) =  
 n : (preOrder t1 ++ preOrder t2)`

## ➤ ¿Cómo sería posOrder?



# Expresiones Aritméticas

- ◆ Definimos expresiones aritméticas
    - ◆ constantes numéricas
    - ◆ sumas y productos de otras expresiones
- data ExpA = Cte Int | Suma ExpA ExpA  
                  | Mult ExpA ExpA
- ◆ Ejemplos:
    - ◆ 2 se representa (Cte 2)
    - ◆  $(4*4)$  se representa (Mult (Cte 4) (Cte 4))
    - ◆  $((2*3)+4)$  se representa  
Suma (Mult (Cte 2) (Cte 3)) (Cte 4)



# Expresiones Aritméticas

- ◆ ¿Cómo dar el significado de una ExpA?

$\text{evalEA} :: \text{ExpA} \rightarrow \text{Int}$

$\text{evalEA} (\text{Cte } n) = n$

$\text{evalEA} (\text{Suma } e1 \ e2) = \text{evalEA } e1 + \text{evalEA } e2$

$\text{evalEA} (\text{Mult } e1 \ e2) = \text{evalEA } e1 * \text{evalEA } e2$

- ◆ Reduzca:

$\text{evalEA} (\text{Suma} (\text{Mult} (\text{Cte } 2) (\text{Cte } 3)) (\text{Cte } 4))$

$\text{evalEA} (\text{Mult} (\text{Cte } 2) (\text{Suma} (\text{Cte } 3) (\text{Cte } 4)))$

# Definición de LIS

- ◆ Definimos un Lenguaje Imperativo Simple
  - ◆ asignación de expresiones numéricas
  - ◆ sentencias if y while sobre expresiones booleanas
  - ◆ secuencia de sentencias

- ◆ Ejemplo:

```
a := n; fac_n := 1;
while (a > 0) do
    fac_n := a * fac_n;
    a := a - 1;
od;
```

# Definición de LIS

- Definimos tipos algebraicos para representar un programa LIS

type Variable = String

type Program = [Statement]

data Statement = Skip

| Assign Variable NExp

| If BExp [Statement] [Statement]

| While BExp [Statement]

# Definición de LIS (cont.)

data NExp = Vble String | NCte Int | Add NExp NExp  
          | Sub NExp NExp | Mul NExp NExp  
          | Div NExp NExp | Mod NExp NExp

data BExp = BCte Bool | Not BExp  
          | And BExp BExp | Or BExp BExp  
          | RelOp ROp NExp NExp

data ROp = Equal | NotEqual  
          | Greater | Lower  
          | GreaterEqual | LowerEqual

# Evaluador de LIS

- ◆ Tipo abstracto para representar la memoria

data State -- Tipo abstracto de estados

initialState :: State

-- Un estado sin variables

getVar :: Variable -> State -> Maybe Int

-- Retorna el valor asociado a la variable dada

putVar :: Int -> Variable -> State -> State

-- Asigna un valor a una variable

variables :: State -> [ Variable ]

-- Retorna las variables definidas en el estado

# Evaluador de LIS (cont.)

## ◆ Semántica de expresiones aritméticas

$\text{evalN} :: \text{NExp} \rightarrow (\text{State} \rightarrow \text{Int})$

$\text{evalN} (\text{Vble } x) \text{ st} =$

$\text{case } (\text{getVar } x \text{ st}) \text{ of}$

$\text{Nothing} \rightarrow \text{error } ("variable \text{ \"++x++\" indefinida"})$

$\text{Just } v \rightarrow v$

$\text{evalN} (\text{NCte } n) \text{ st} = n$

$\text{evalN} (\text{Add } e1 \ e2) \text{ st} = \text{evalN } e1 \text{ st} + \text{evalN } e2 \text{ st}$

...

## ◆ Observar que las expresiones tienen variables, y eso complica su semántica

# Evaluador de LIS (cont.)

- ◆ Semántica de expresiones booleanas

$\text{evalB} :: \text{BExp} \rightarrow (\text{State} \rightarrow \text{Bool})$

$\text{evalB} (\text{BCte } b) \text{ st} = b$

$\text{evalB} (\text{RelOp } \text{rop } e1 \ e2) \text{ st} =$

$\text{evalROp } \text{rop} (\text{evalN } e1 \text{ st}) (\text{evalN } e2 \text{ st})$

$\text{evalB} (\text{And } e1 \ e2) \text{ st} = \text{evalB } e1 \text{ st} \ \&\& \ \text{evalB } e2 \text{ st}$

...

$\text{evalROp} :: \text{ROp} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Bool}$

$\text{evalROp } \text{Equal} = (==)$

...

- ◆ ¿Por qué hace falta el estado para dar significado a una expresión booleana?

# Evaluador de LIS (cont.)

## ◆ Semántica de sentencias LIS

$\text{evalS} :: \text{Statement} \rightarrow (\text{State} \rightarrow \text{State})$

$\text{evalS Skip} = \lambda \text{st} \rightarrow \text{st}$

$\text{evalS (Assign } x \text{ ne)} = \lambda \text{st} \rightarrow \text{putVar (evalN ne st) } x \text{ st}$

$\text{evalS (If be } p1 \text{ } p2) =$   
 $\lambda \text{st} \rightarrow \text{if (evalB be st) then (evalP } p1 \text{ st)}$   
 $\qquad \qquad \qquad \text{else (evalP } p2 \text{ st)}$

$\text{evalS (While be } p) =$   
 $\lambda \text{st} \rightarrow \text{evalS (If be (p ++ [While be p]) [Skip]) st}$



# Evaluador de LIS (cont.)

## ◆ Semántica de programas LIS

$\text{evalP} :: \text{Program} \rightarrow (\text{State} \rightarrow \text{State})$

$\text{evalP} [] = \lambda \text{st} \rightarrow \text{st}$

$\text{evalP} (p:ps) =$

$\lambda \text{st} \rightarrow \text{let } \text{st}' = \text{evalS } p \text{ st}$   
 $\text{in } \text{evalP } ps \text{ st}'$

## ◆ ¡Observar cómo la secuencia de sentencias ALTERA el estado antes de proseguir!

# Resumen

- ◆ Tipos algebraicos recursivos
  - ◆ Aleph, BE, Listas,
- ◆ Árboles
  - ◆ Arbol a, ExpA
- ◆ Funciones sobre tipos recursivos
- ◆ Programas LIS
- ◆ Semántica (integral) de programas LIS

