Programación Funcional

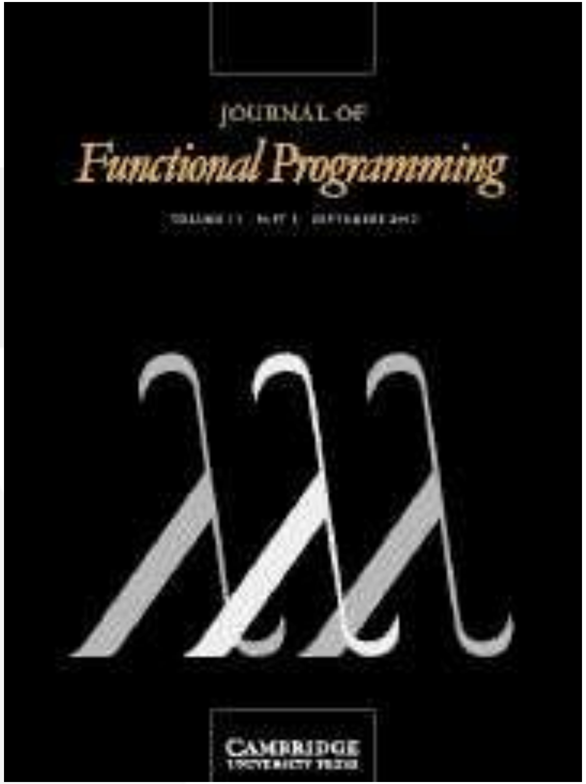
Clases teóricas

por Pablo E. “Fidel” Martínez López

9. Inducción y recursión III

"Education matters. The lack of education on functional programming languages and techniques is visible on a daily basis. Our students, co-workers, friends and colleagues just don't know enough about these ideas and therefore often fail to implement the best possible solutions for their programming problems."

Welcome to the Educational Pearls Column
Matthias Felleisen
Journal of Functional Programming, 13 (5)
Septiembre 2003



Escenas del capítulo anterior

Principio de inducción estructural

Principio de inducción estructural

¿para todo **x :: S**. **P(x)**?

es equivalente a

Caso base 1) ¿$P(z_1)$?

...

Caso base n) ¿$P(z_n)$?

Caso inductivo 1) HI_{11}) ¡$P(e_{11})$! … HIi1) ¡$P(e_{i1})$!

TI_1) ¿$P(e_1)$?

...

Caso inductivo k) HI_{1k}) ¡$P(e_{1k})$! … HIik) ¡$P(e_{ik})$!

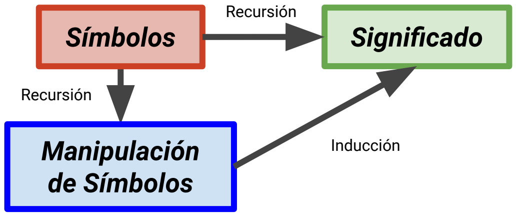
TI_k) ¿$P(e_k)$?

Recursión estructural sobre listas

- Definición *inductiva* de listas `[a]`
  - **Regla base:** `[] :: [a]`
  - **Regla inductiva:** si `x :: a`, `xs :: [a]`, entonces `x:xs :: [a]`
- Funciones por recursión estructural sobre listas

```
f :: [A] -> B
f []     = ...
f (x:xs) = ... x ... f xs ...
```

Representaciones de datos



- ¿Qué cosas interesantes se pueden representar?
  - Definición de *conjuntos de símbolos estructurados* por inducción
    - Ej. **`N`**
  - *Asignación de significado* mendiante funciones recursivas
    - Ej. **`evalN`**
  - *Tratamiento simbólico* mediante funciones recursivas
    - Ej. **`addN`**
  - *Coherencia* de manipulación y significado mediante inducción
    - Ej. para todo **n**. para todo **m**.

```
evalN (addN n m) = evalN n + evalN m
```

Representaciones de datos: Expresiones aritméticas

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?
  - Sumas y productos de constantes numéricas
    - `2`
    - `(2*3)+4`
    - `(0+0)+(2+(3+0))`
    - `2*3`
    - `2*(3+4)`
    - `(2+3)*(4+0)`
  - Es necesario un constructor por cada forma posible
    - No se trata *solamente* de hacer la cuenta
    - Se desea poder contar operaciones, y otras manipulaciones

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?
  - Tres formas, tres constructores (dos inductivos)

```
data ExpA = Cte Int             -- Para representar constantes
          | Suma ExpA ExpA      -- Para representar sumas
          | Prod ExpA ExpA      -- Para representar productos
```

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?
  - Tres formas, tres constructores (dos inductivos)

```
data ExpA = Cte   Int          -- Para representar constantes
          | Suma ExpA ExpA -- Para representar sumas
          | Prod ExpA ExpA -- Para representar productos
```

    - `2` se representa como `Cte 2`
    - `2*(3+4)` como `Prod (Cte 2)`
      `(Suma (Cte 3) (Cte 4))`
    - `(2*3)+4` como `Suma (Prod (Cte 2) (Cte 3))`
      `(Cte 4)`

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

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

  - ¿Por qué el argumento del caso base está en verde?

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

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

  - ¿Por qué el argumento del caso base está en verde?
    - Es más simple representar los números como significados
      - Porque el foco está en las expresiones y no en los números
    - Se combinan representaciones (símbolos y significados)

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

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

  - ¿Por qué el argumento del caso base está en verde?
    - Es más simple representar los números como significados
      - Porque el foco está en las expresiones y no en los números
    - Se combinan representaciones (símbolos y significados)
  - ¿Cómo sería una representación *totalmente* simbólica?

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

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

  - ¿Por qué el argumento del caso base está en verde?
    - Es más simple representar los números como significados
      - Porque el foco está en las expresiones y no en los números
    - Se combinan representaciones (símbolos y significados)
  - ¿Cómo sería una representación *totalmente* simbólica?
    - Habría que usar una representación simbólica de números
    - Se puede elegir (unaria, binaria, decimal, etc.)

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

```
data ExpS = CteS N | SumS ExpS ExpS | ProdS ExpS ExpS
```

  - ¿En qué difieren **ExpS** y **ExpA**?

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

```
data ExpS = CteS N | SumS ExpS ExpS | ProdS ExpS ExpS
```

  - ¿En qué difieren **`ExpS`** y **`ExpA`**?
    - Solamente en los símbolos usados
    - Tienen la misma *estructura*

Expresiones aritméticas

- ¿Cómo representar *expresiones aritméticas*?

```
data ExpS = CteS N | SumS ExpS ExpS | ProdS ExpS ExpS
```

  - ¿En qué difieren **ExpS** y **ExpA**?
    - Solamente en los símbolos usados (que indican formas de mirar)
    - Tienen la misma *estructura*

```
Cte 2                    vs.  CteS (S(S Z))

Suma (Cte 2) (Cte 3)     vs.  SumS (CteS (S(S Z)))
                                   (CteS (S(S(S Z))))
```

Expresiones aritméticas

- ¿Los nombres son importantes?

```
data TA = A | B TA        -- Estructuralmente similar a N
data TB = C TA | D TB TB | E TB TB
```

Expresiones aritméticas

- ¿Los nombres son importantes?

```
data TA = A | B TA        -- Estructuralmente similar a N
data TB = C TA | D TB TB | E TB TB
```

  - ¡Tiene la misma *estructura* que **`ExpA`** o **`ExpS`**!

Expresiones aritméticas

- ¿Los nombres son importantes?

```
data TA = A | B TA         -- Estructuralmente similar a N
data TB = C TA | D TB TB | E TB TB
```

  - ¡Tiene la misma *estructura* que **ExpA** o **ExpS**!
    - Pero no es fácil leerlo y saber qué se quiere representar

Expresiones aritméticas

- ¿Los nombres son importantes?

```
data TA = A | B TA        -- Estructuralmente similar a N
data TB = C TA | D TB TB | E TB TB
```

- ¡Tiene la misma *estructura* que **ExpA** o **ExpS**!
  - Pero no es fácil leerlo y saber qué se quiere representar
    - `Cte 2` vs. `C (B(B A))`
    - `Suma (Cte 2) (Cte 3)` vs. `D (C (B(B A))) (C (B(B(B A))))`
  - Los nombres son importantes para la legibilidad

Expresiones aritméticas

- ¿Cómo es la recursión estructural sobre **`ExpA`**?

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

Expresiones aritméticas

- ¿Cómo es la recursión estructural sobre **`ExpA`**?

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

  - El esquema queda

```
f :: ExpA -> B
f (Cte n)        = ... n ...
f (Suma e1 e2) = ... f e1 ... f e2 ...
f (Prod e1 e2) = ... f e1 ... f e2 ...
```

Expresiones aritméticas

- ¿Cómo es la recursión estructural sobre **`ExpA`**?

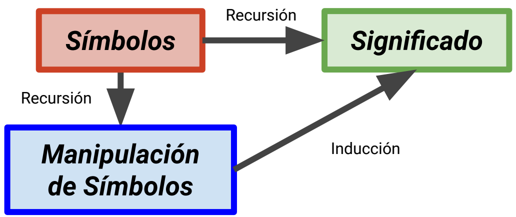
```
data ExpA = Cte Int | Suma ExpA ExpA | Prod ExpA ExpA
```

  - El esquema queda

```
f :: ExpA -> B
f (Cte n)      = ... n ...
f (Suma e1 e2) = ... f e1 ... f e2 ...
f (Prod e1 e2) = ... f e1 ... f e2 ...
```

    - ¿Por qué son 3 casos?
    - ¿Por qué hay 2 aplicaciones recursivas?

Expresiones aritméticas

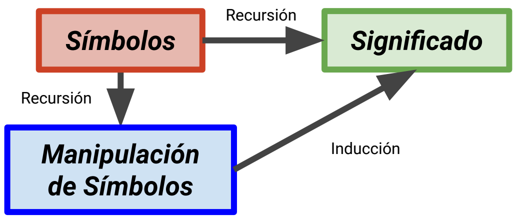


- Significado para las expresiones aritméticas
  - Función de asignación de significado
  - Por recursión estructural en la expresión aritmética

```
evalExpA :: ExpA -> Int
evalExpA (Cte n)        = ... n ...
evalExpA (Suma e1 e2) = ... evalExpA e1 ... evalExpA e2 ...
evalExpA (Prod e1 e2) = ... evalExpA e1 ... evalExpA e2 ...
```

  - Se decide usar recursión
  - Se plantea el esquema

Expresiones aritméticas

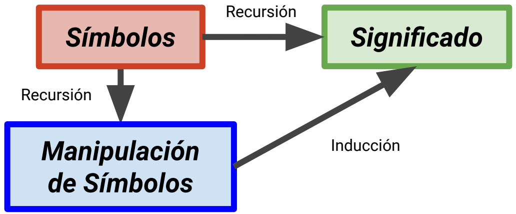


- Significado para las expresiones aritméticas
  - Función de asignación de significado
  - Por recursión estructural en la expresión aritmética

```
evalExpA :: ExpA -> Int
evalExpA (Cte n)       = ... n ...
evalExpA (Suma e1 e2) = evalExpA e1 + evalExpA e2
evalExpA (Prod e1 e2) = evalExpA e1 * evalExpA e2
```

- Se definen los casos inductivos

Expresiones aritméticas

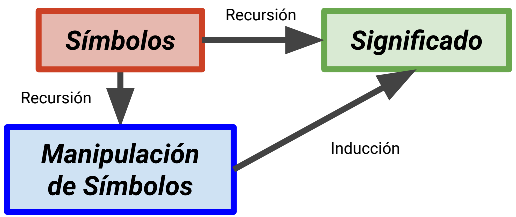


- Significado para las expresiones aritméticas
  - Función de asignación de significado
  - Por recursión estructural en la expresión aritmética

```
evalExpA :: ExpA -> Int
evalExpA (Cte n)       = n
evalExpA (Suma e1 e2) = evalExpA e1 + evalExpA e2
evalExpA (Prod e1 e2) = evalExpA e1 * evalExpA e2
```

- Se completa con los casos base

Expresiones simbólicas

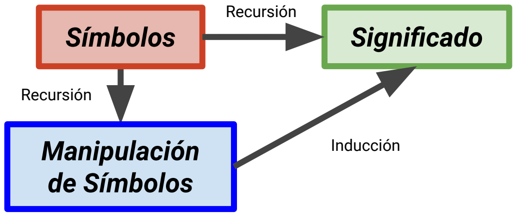


- Significado para las expresiones aritméticas
  - Función recursiva de asignación de significado
  - ¿Cómo sería con las expresiones simbólicas?

```
evalES    :: ExpS -> Int
evalES (CteS   n)      = evalN n
evalES (SumS   s1 s2) = evalES s1 + evalES s2
evalES (ProdS s1 s2) = evalES s1 * evalES s2
```

  - ¡El argumento simbólico del caso base debe transformarse en un significado, usando la función correspondiente!
    - Cada representación tiene sus ventajas y desventajas

Expresiones aritméticas



- Significado para las expresiones aritméticas

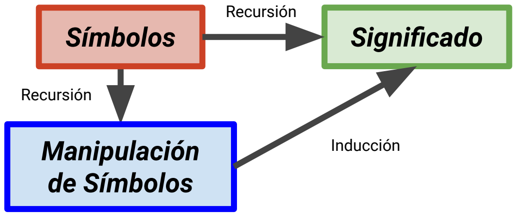
```
evalExpA :: ExpA -> Int
evalExpA (Cte n)       = n
evalExpA (Suma e1 e2) = evalExpA e1 + evalExpA e2
evalExpA (Prod e1 e2) = evalExpA e1 * evalExpA e2
```

```
  evalExpA (Suma (Cte 2) (Prod (Cte 3) (Cte 4)))
→                                 (evalExpA.2, e1=Cte 2, e2=Prod (Cte 3) (Cte 4))
  evalExpA (Cte 2) + evalExpA (Prod (Cte 3) (Cte 4))
→2                                (evalExpA.1, n=2; evalExpA.3, e1=Cte 3, e1=Cte 4)
  2 + (evalExpA (Cte 3) * evalExpA (Cte 4))
→2                                (evalExpA.1, n=3; evalExpA.1, n=4)
  2 + (3 * 4)
```

- La función da el significado esperado

```
evalExpA (Suma (Cte 2) (Prod (Cte 3) (Cte 4))) = 2 + (3 * 4)
```

Expresiones simbólicas



- Significado para las expresiones simbólicas

```
evalES :: ExpS -> Int
evalES (CteS n)       = evalN n
evalES (SumS  s1 s2) = evalES s1 + evalES s2
evalES (ProdS s1 s2) = evalES s1 * evalES s2
```

`evalES (SumS (CteS(S(S Z))) (ProdS (CteS(S(S(S Z)))) (CteS(S(S(S(S Z)))))))`

$\longrightarrow$ (evalES.2, s1=`CteS(S(S Z))`, s2=`ProdS (CteS(S(S(S Z)))) (CteS(S(S(S(S Z)))))`)

`evalES (CteS(S(S Z))) + evalES (ProdS (CteS(S(S(S Z)))) (CteS(S(S(S(S Z))))))`

$\longrightarrow^{2}$ (evalES.1, n=`S(S Z)`; evalES.3, s1=`CteS(S(S(S Z)))`, s2=`CteS(S(S(S(S Z))))`)

`evalN (S(S Z)) + (evalES (CteS(S(S(S Z)))) * evalES (Cte(S(S(S(S Z))))))`

$\longrightarrow^{*}$ (Lema de evalN; evalES.1, n=`S(S(S Z))`; evalES.1, n=`S(S(S(S Z)))`)

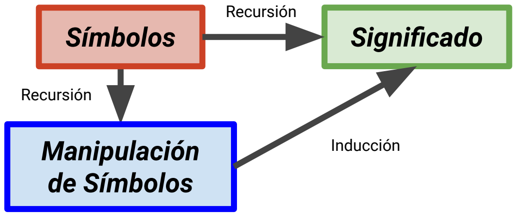
`2 + (evalN (S(S(S Z))) * evalN (S(S(S(S Z)))))`

$\longrightarrow^{*}$ (Lema de evalN dos veces)

`2 + (3 * 4)`

- Totalmente simbólico tarda más en el cálculo, pero significa lo mismo (pero se pueden analizar todos los símbolos)

Expresiones aritméticas

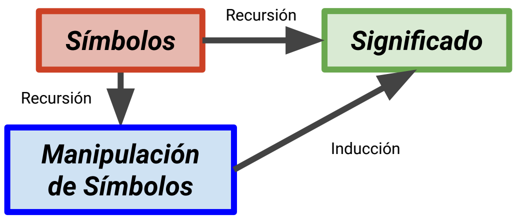


- Significado para las expresiones aritméticas
  - Función recursiva de asignación de significado
  - Volviendo a **`ExpA`**

```
evalExpA :: ExpA -> Int
evalExpA (Cte n)        = n
evalExpA (Suma e1 e2) = evalEA exp1 + evalExpA e2
evalExpA (Prod e1 e2) = evalEA exp1 * evalExpA e2
```

- ¡Observar que `Suma` representa al + y `Prod`, al `*`!
  - Puede parecer obvio, pero no lo es
  - Muestra la adecuación de los nombres elegidos

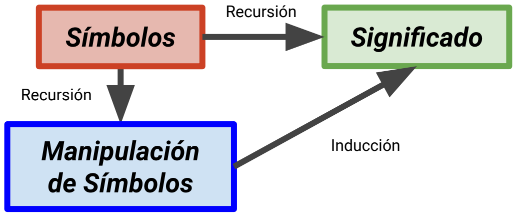
Expresiones aritméticas



- Significado para las expresiones aritméticas
  - Función recursiva de asignación de significado
  - Comparar con la versión con nombres “automáticos”

```
evalTB :: TB -> Int
evalTB    (C n)          = evalTA n
evalTB    (D e1 e2)      = evalTB e1    + evalTB e2
evalTB    (E e1 e2)      = evalTB e1    * evalTB e2
```

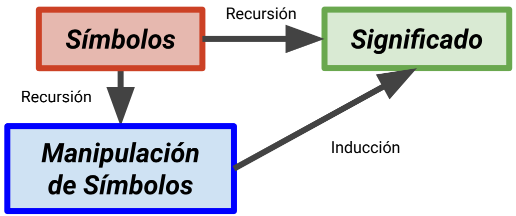
- No es tan obvio que D representa al + y E, al *, ¿no?
  - Siempre es mejor elegir nombres adecuados



Expresiones aritméticas

- Manipulación simbólica de expresiones aritméticas
  - ¿Cómo transformar de **`ExpA`** a **`TB`**?
  - Por recursión estructural en la expresión aritmética

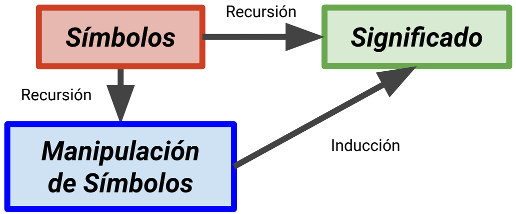
Expresiones aritméticas



- Manipulación simbólica de expresiones aritméticas
  - ¿Cómo transformar de **`ExpA`** a **`TB`**?
  - Por recursión estructural en la expresión aritmética

```
expA2tb :: ExpA -> TB
expA2tb (Cte n)       = ...
expA2tb (Suma e1 e2) = ... expA2tb e1 ... expA2tb e2 ...
expA2tb (Prod e1 e2) = ... expA2tb e1 ... expA2tb e2 ...
```

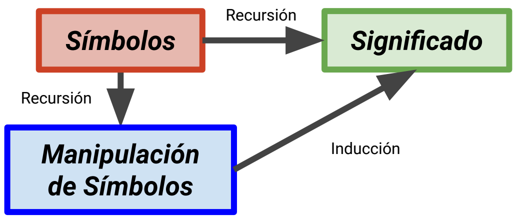
Expresiones aritméticas



- Manipulación simbólica de expresiones aritméticas
  - ¿Cómo transformar de **ExpA** a **TB**?
  - Por recursión estructural en la expresión aritmética

```
expA2tb :: ExpA -> TB
expA2tb (Cte n)        = C (int2ta n)
expA2tb (Suma e1 e2) = D (expA2tb e1) (expA2tb e2)
expA2tb (Prod e1 e2) = E (expA2tb e1) (expA2tb e2)
```

Expresiones aritméticas

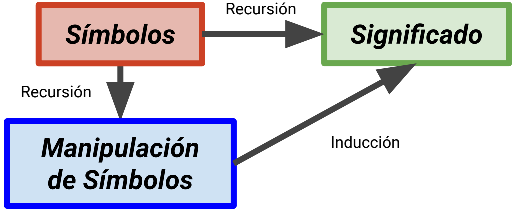


- Manipulación simbólica de expresiones aritméticas
  - ¿Cómo transformar de **`ExpA`** a **`TB`**?
  - Por recursión estructural en la expresión aritmética

```
expA2tb :: ExpA -> TB
expA2tb (Cte n)       = C (int2ta n)
expA2tb (Suma e1 e2) = D (expA2tb e1) (expA2tb e2)
expA2tb (Prod e1 e2) = E (expA2tb e1) (expA2tb e2)
```

- Se pueden hacer operaciones sobre los símbolos
- Una vez más: NO hace falta conocer el significado

Expresiones aritméticas



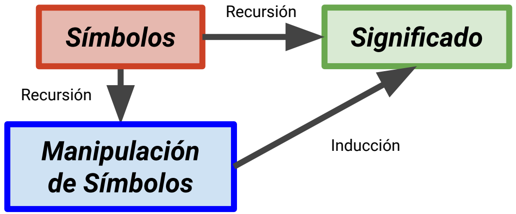
- Manipulación simbólica de expresiones aritméticas

(0+0)+(2+(3+0))

```
ej1 = Suma (Suma (Cte 0) (Cte 0))
           (Suma (Cte 2) (Suma (Cte 3) (Cte 0)))

  expA2tb ej1
= ...
```

Expresiones aritméticas



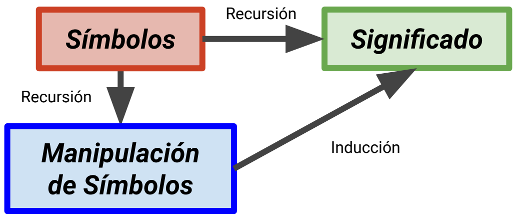
❑ Manipulación simbólica de expresiones aritméticas

```
ej1 = Suma (Suma (Cte 0) (Cte 0))
           (Suma (Cte 2) (Suma (Cte 3) (Cte 0)))
```

(0+0)+(2+(3+0))

```
  expA2tb ej1
= D (expA2tb (Suma (Cte 0) (Cte 0)))
    (expA2tb (Suma (Cte 2) (Suma (Cte 3) (Cte 0))))
```

Expresiones aritméticas



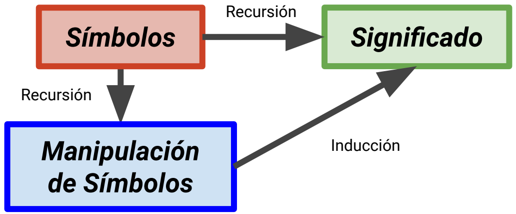
- Manipulación simbólica de expresiones aritméticas

```
ej1 = Suma (Suma (Cte 0) (Cte 0))
           (Suma (Cte 2) (Suma (Cte 3) (Cte 0)))
```

(0+0)+(2+(3+0))

```
  expA2tb ej1
= D (expA2tb (Suma (Cte 0) (Cte 0)))
    (expA2tb (Suma (Cte 2) (Suma (Cte 3) (Cte 0))))
= D (D (expA2tb (Cte 0)) (expA2tb (Cte 0)))
    (D (expA2tb (Cte 2)) (expA2tb (Suma (Cte 3) (Cte 0))))
```

Expresiones aritméticas



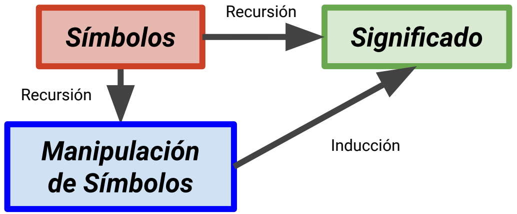
- Manipulación simbólica de expresiones aritméticas

```
ej1 = Suma (Suma (Cte 0) (Cte 0))
           (Suma (Cte 2) (Suma (Cte 3) (Cte 0)))
```

(0+0)+(2+(3+0))

```
  expA2tb ej1
= D (expA2tb (Suma (Cte 0) (Cte 0)))
    (expA2tb (Suma (Cte 2) (Suma (Cte 3) (Cte 0))))
= D (D (expA2tb (Cte 0)) (expA2tb (Cte 0)))
    (D (expA2tb (Cte 2)) (expA2tb (Suma (Cte 3) (Cte 0))))
= D (D (C (int2ta 0)) (C (int2ta 0)))
    (D (C (int2ta 2)) (D (expA2tb (Cte 3)) (expA2tb (Cte 0))))
```

Expresiones aritméticas



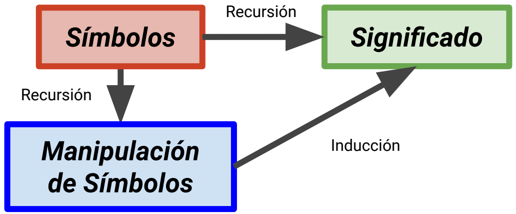
- Manipulación simbólica de expresiones aritméticas

```
ej1 = Suma (Suma (Cte 0) (Cte 0))
           (Suma (Cte 2) (Suma (Cte 3) (Cte 0)))
```

(0+0)+(2+(3+0))

```
  expA2tb ej1
= D (expA2tb (Suma (Cte 0) (Cte 0)))
    (expA2tb (Suma (Cte 2) (Suma (Cte 3) (Cte 0))))
= D (D (expA2tb (Cte 0)) (expA2tb (Cte 0)))
    (D (expA2tb (Cte 2)) (expA2tb (Suma (Cte 3) (Cte 0))))
= D (D (C (int2ta 0)) (C (int2ta 0)))
    (D (C (int2ta 2)) (D (expA2tb (Cte 3)) (expA2tb (Cte 0))))
= D (D (C A) (C A))
    (D (C (B(B A))) (D (C (B(B(B A)))) (C A)))
```

Expresiones aritméticas



- Manipulación simbólica de expresiones aritméticas

(0+0)+(2+(3+0))

```
ej1 = Suma (Suma (Cte 0) (Cte 0))
           (Suma (Cte 2) (Suma (Cte 3) (Cte 0)))

  expA2tb ej1
= D (expA2tb (Suma (Cte 0) (Cte 0)))
    (expA2tb (Suma (Cte 2) (Suma (Cte 3) (Cte 0))))
= D (D (expA2tb (Cte 0)) (expA2tb (Cte 0)))
    (D (expA2tb (Cte 2)) (expA2tb (Suma (Cte 3) (Cte 0))))
= D (D (C (int2ta 0)) (C (int2ta 0)))
    (D (C (int2ta 2)) (D (expA2tb (Cte 3)) (expA2tb (Cte 0))))
= D (D (C A) (C A))
    (D (C (B(B A))) (D (C (B(B(B A)))) (C A)))

  expA2d ej1 = D (D (C A) (C A))
                 (D (C (B(B A))) (D (C (B(B(B A)))) (C A)))
```

Expresiones aritméticas: inducción

- Principio de inducción estructural para **`ExpA`**

  ¿para todo **`e::ExpA`**. P(e)?

  es equivalente a

  Caso base) ¿P(`Cte` **n**)?

  Caso ind.1) **HI$_{11}$)** ¡P($\mathbf{e_1}$)!

  HI$_{21}$) ¡P($\mathbf{e_2}$)!

  TI$_1$) ¿P(`Suma` $\mathbf{e_1}$ $\mathbf{e_2}$)?

  Caso ind.2) **HI$_{12}$)** ¡P($\mathbf{e_1}$)!

  HI$_{22}$) ¡P($\mathbf{e_2}$)!

  TI$_2$) ¿P(`Mult` $\mathbf{e_1}$ $\mathbf{e_2}$)?

¡Solamente vale para la parte finita y totalmente definida (inductiva) de **`ExpA`**!

Expresiones aritméticas: inducción

- **Prop:** ¿para todo `e::ExpA`. `evalTB (expA2tb e) = evalExpA e`?

  Dem: sea **e'** una ExpA. Por ppio. de ind. en la estructura de **e'**:

  Caso base) ¿`evalTB (expA2tb (Cte n)) = evalExpA (Cte n)`?

  Caso ind.1) HI_{11}) ¡`evalTB (expA2tb` e_1`) = evalExpA` e_1!

  HI_{21}) ¡`evalTB (expA2tb` e_2`) = evalExpA` e_2!

  TI_1) ¿`evalTB (expA2tb (Suma` e_1 e_2`)) = evalExpA (Suma` e_1 e_2`)`?

  Caso ind.2) HI_{12}) ¡`evalTB (expA2tb` e_1`) = evalExpA` e_1!

  HI_{22}) ¡`evalTB (expA2tb` e_2`) = evalExpA` e_2!

  TI_2) ¿`evalTB (expA2tb (Mult` e_1 e_2`)) = evalExpA (Mult` e_1 e_2`)`?

Árboles

Árboles: motivación

- Se trabajaron algunos tipos similares

```
data EquipoS = BecarioS Nombre
             | InvestigadorS Nombre EquipoS EquipoS
data ExpA = Cte Int | Suma ExpA ExpA | Mult ExpA ExpA
```

  - ¿Se podrán generalizar estos tipos para quedarse solamente con la estructura, sin significado?
    - Nuevamente, para generalizar deben usarse *parámetros*…
    - ¿Qué tienen en común?
      - Un constructor base
      - Uno (o más) constructor(es) con *varias* partes recursivas

Árboles binarios: definición

- Definición inductiva de árboles binarios

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

  - Un tipo ***árbol*** es un tipo algebraico con al menos un constructor con dos o más partes inductivas
  - ¡Se pueden usar *todas* las técnicas vistas hasta ahora!
    - Propiedades de los tipos algebraicos, inducción y recursión, visión denotacional vs. operacional, etc.
  - ¡No existe un único tipo de árboles!

Árboles binarios

- Definición inductiva de árboles binarios

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

  - Elementos de `Arbol Int`

```
Hoja 2
```

```
Hoja 0
```

```
Nodo 3 (Hoja 2)
       (Hoja 0)
```

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```

Árboles binarios

- Definición inductiva de árboles binarios

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

  - Elementos de `Arbol Int`
    - Cada árbol contiene *toda* la información que lo define
    - *La visión funcional de estructuras de datos es **diferente** a la imperativa o de objetos*
    - No existe la idea de "posición de memoria" ni la de "identidad"

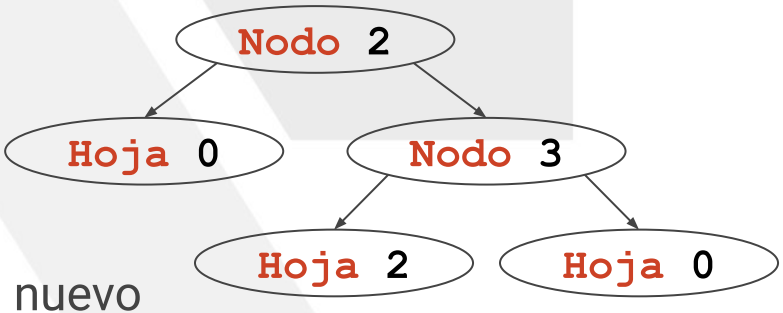
Árboles binarios

- Graficación de árboles binarios

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

  - Es posible usar *diagramas jerárquicos*

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```



    - Cada parte inductiva como un nuevo diagrama apuntado con flechas desde el diagrama del elemento

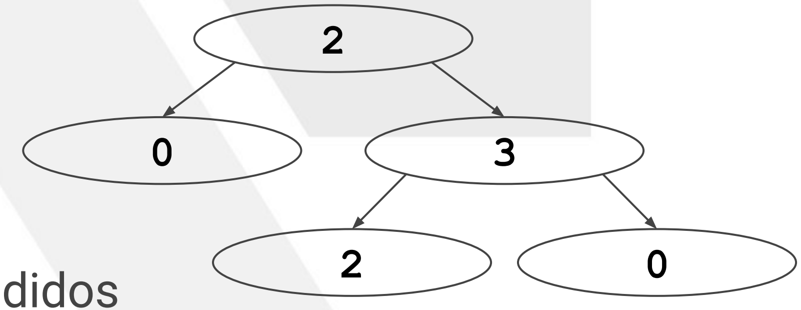
Árboles binarios

- Graficación de árboles binarios

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

  - Es posible usar diagramas jerárquicos

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```



    - Podemos dar por sobreentendidos los constructores para las personas

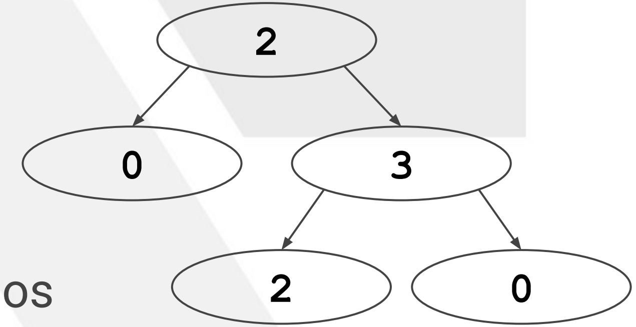
Árboles binarios

- Graficación de árboles binarios

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

- Es posible usar diagramas jerárquicos

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```



- Podemos dar por sobreentendidos los constructores para las personas

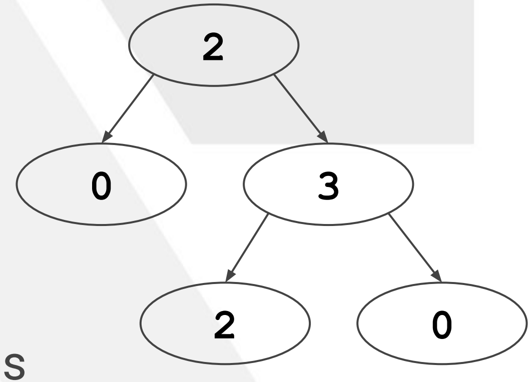
Árboles binarios

- Graficación de árboles binarios

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

- Es posible usar diagramas jerárquicos

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```



- Podemos dar por sobreentendidos los constructores para las personas

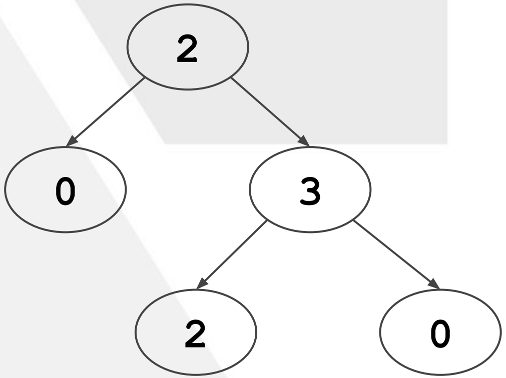
Árboles binarios

- Graficación de árboles binarios

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

  - Es posible usar diagramas jerárquicos

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```



    - Podemos dar por sobreentendidos los constructores para las personas

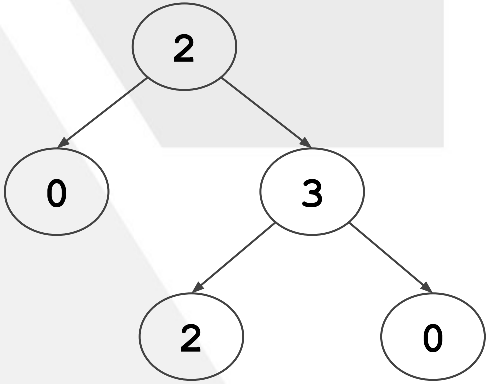
Árboles binarios

- Graficación de árboles binarios

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

  - Es posible usar diagramas jerárquicos

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```



    - Podemos dar por sobreentendidos los constructores para las personas

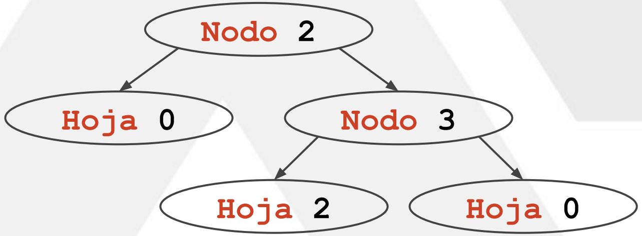
Árboles binarios

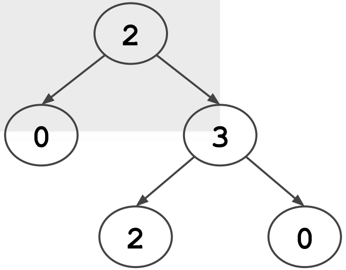
- Graficación de árboles binarios

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

  - Es posible usar diagramas jerárquicos

```
Nodo 2 (Hoja 0)
       (Nodo 3 (Hoja 2)
               (Hoja 0))
```





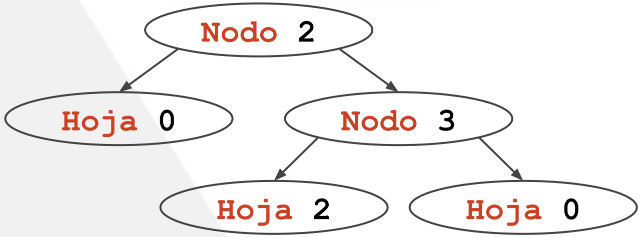
    - En distintas formas

Árboles binarios

- Naturaleza inductiva de los árboles (como datos)

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

  - El árbol es *todo* el dato
  - Hay preguntas *sin sentido en este modelo* de cómputo
    - ¿Cuál es el *padre* de `Hoja 0`?

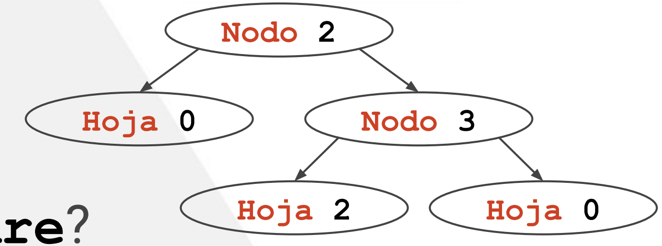


Árboles binarios

- Naturaleza inductiva de los árboles (como datos)

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

  - El árbol es *todo* el dato
  - Hay preguntas *sin sentido en este modelo* de cómputo
    - ¿Cuál es el *padre* de `Hoja 0`?
    - ¿Es posible definir *"padre de"*?
    - ¿Qué implica pedir "*el padre*"?
    - ¿Qué tipo tendría que tener `padre`?



Árboles binarios: recursión estructural

- ¿Cómo es la recursión estructural sobre **`Arbol`**?

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

Árboles binarios: recursión estructural

- ¿Cómo es la recursión estructural sobre **`Arbol`**?

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

  - El esquema queda

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

Árboles binarios: recursión estructural

- ¿Cómo es la recursión estructural sobre **`Arbol`**?

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

  - El esquema queda

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

    - Posee todas las propiedades y técnicas vistas
    - La metodología de construcción es la ya vista

Árboles binarios: recursión estructural

- Funciones por recursión en la estructura del árbol
  - Cantidad de hojas

```
hojas :: Arbol a -> Int
hojas (Hoja x)        = … x …
hojas (Nodo x t1 t2) = … x … hojas t1 … hojas t2 …
```

  - Altura del árbol (cantidad máxima de nodos hasta una hoja)

```
altura :: Arbol a -> Int
altura (Hoja x)        = … x …
altura (Nodo x t1 t2) = … x … altura t1 … altura t2 …
```

Árboles binarios: recursión estructural

- Funciones por recursión en la estructura del árbol
  - Cantidad de hojas

```
hojas :: Arbol a -> Int
hojas (Hoja x)        = … x …
hojas (Nodo x t1 t2) = hojas t1 + hojas t2
```

  - Altura del árbol (cantidad máxima de nodos hasta una hoja)

```
altura :: Arbol a -> Int
altura (Hoja x)        = … x …
altura (Nodo x t1 t2) = 1 + (altura t1 `max` altura t2)
```

Árboles binarios: recursión estructural

- Funciones por recursión en la estructura del árbol
  - Cantidad de hojas

```
hojas :: Arbol a -> Int
hojas (Hoja x)        = 1
hojas (Nodo x t1 t2) = hojas t1 + hojas t2
```

  - Altura del árbol (cantidad máxima de nodos hasta una hoja)

```
altura :: Arbol a -> Int
altura (Hoja x)        = 0
altura (Nodo x t1 t2) = 1 + (altura t1 `max` altura t2)
```

Árboles binarios: inducción estructural

- Principio de inducción estructural para **`Arbol`**

  ¿para todo **`t::Arbol`** **A**. P(**t**)?

  es equivalente a

  Caso base) ¿P(**`Hoja`** **x**)?

  Caso ind.) **HI_1**) ¡P($\mathbf{t_1}$)!

  HI_2) ¡P($\mathbf{t_2}$)!

  TI) ¿P(**`Nodo`** **x** $\mathbf{t_1}$ $\mathbf{t_2}$)?

- Recordar que solamente sirve para la parte finita y totalmente definida (inductiva) de **`Arbol`**

Árboles binarios: inducción

```
hojas  (Hoja x)      = 1
hojas  (Nodo t1 t2)  =
   hojas t1 + hojas t2

altura  (Hoja x)      = 0
altura  (Nodo t1 t2)  =
   1 + (altura t1 `max` altura t2)
```

- **Prop:** ¿para todo **t**`::Arbol` A. `hojas t` ≤ `2^(altura t)`?

  Dem: sea **t'** un árbol. Por ppio. de ind. sobre la estructura de **t'**

  Caso base) ¿`hojas (Hoja x)` ≤ `2^(altura (Hoja x))`?

  Caso ind.) **HI$_1$)** ¡`hojas` t_1 ≤ `2^(altura` t_1`)`!

  HI$_2$) ¡`hojas` t_2 ≤ `2^(altura` t_2`)`!

  TI) ¿`hojas (Nodo` x t_1 t_2`)` ≤ `2^(altura (Nodo` x t_1 t_2`))`?

- Recordar que
  - primero se decide usar inducción
  - después se plantean los casos
  - finalmente se demuestran los casos planteados

Árboles binarios: inducción

```
hojas  (Hoja x)       = 1
hojas  (Nodo t1 t2)  =
    hojas t1 + hojas t2

altura  (Hoja x)       = 0
altura  (Nodo t1 t2)  =
    1 + (altura t1 `max` altura t2)
```

- **Prop:** ¿para todo `t::Arbol A`. `hojas t` ≤ `2^(altura t)`?
  Dem: sea **t'** un árbol. Por ppio. de ind. sobre la estructura de **t'**

  Caso base) ¿`hojas (Hoja x)` ≤ `2^(altura (Hoja x))`?

`hojas (Hoja x)`	`2^(altura (Hoja x))`
= (`hojas`.1)	= (`altura`.1)
`1`	`2^0`
	= (aritm.)
	`1`

Vale el caso

Árboles binarios: inducción

```
hojas  (Hoja x)       = 1
hojas  (Nodo t1 t2)  =
   hojas t1 + hojas t2

altura  (Hoja x)       = 0
altura  (Nodo t1 t2)  =
   1 + (altura t1 `max` altura t2)
```

- **Prop:** ¿para todo t `::Arbol A`. `hojas` $t \leq$ `2^(altura` t`)`?

  Dem: sea **t'** un árbol. Por ppio. de ind. sobre la estructura de **t'**

Caso ind.) HI$_1$) ¡`hojas` $t_1 \leq$ `2^(altura` t_1`)`!

HI$_2$) ¡`hojas` $t_2 \leq$ `2^(altura` t_2`)`!

TI) ¿`hojas (Nodo` $x\ t_1\ t_2$`))` $\leq$ `2^(altura (Nodo` $x\ t_1\ t_2$`))`?

Atención a las desigualdades

`hojas (Nodo` $x\ t_1\ t_2$`)`

= (`hojas`.2)

`hojas` t_1 `+ hojas` t_2

$\leq$ (HI$_1$, HI$_2$, aritm.)

`2^(altura` t_1`)+2^(altura` t_2`)`

$\leq$ (aritm.)

`2^(1 + altura` t_1 `` `max` `` `altura` t_2`)`

`2^(altura (Nodo` $x\ t_1\ t_2$`))`

= (`altura`.2)

`2^(1 + altura` t_1 `` `max` `` `altura` t_2`)`

Vale el caso

Vale la propiedad

Árboles binarios: naturaleza de los datos

- Algunas cuestiones sobre cómo hablar de árboles
  - “Cambiar” los **2** en las hojas, por **42**

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 ...
```

Árboles binarios: naturaleza de los datos

- Algunas cuestiones sobre cómo hablar de árboles
  - “Cambiar” los **2** en las hojas, por **42**

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n)          = … n …

cambiar2 (Nodo n t1 t2) = … n … cambiar2 t1 …
                                cambiar2 t2 …
```

Árboles binarios: naturaleza de los datos

- Algunas cuestiones sobre cómo hablar de árboles
  - “Cambiar” los **2** en las hojas, por **42**

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n)         = … n …

cambiar2 (Nodo n t1 t2) = Nodo n (cambiar2 t1)
                                 (cambiar2 t2)
```

Árboles binarios: naturaleza de los datos

- Algunas cuestiones sobre cómo hablar de árboles
  - “Cambiar” los **2** en las hojas, por **42**

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n)         = Hoja ( … )

cambiar2 (Nodo n t1 t2) = Nodo n (cambiar2 t1)
                                 (cambiar2 t2)
```

Árboles binarios: naturaleza de los datos

- Algunas cuestiones sobre cómo hablar de árboles
  - "Cambiar" los **2** en las hojas, por **42**

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n)         = Hoja (if n==2 then 42
                                             else n)
cambiar2 (Nodo n t1 t2) = Nodo n (cambiar2 t1)
                                    (cambiar2 t2)
```

Árboles binarios: naturaleza de los datos

- Algunas cuestiones sobre cómo hablar de árboles
  - "Cambiar" los **2** en las hojas, por **42**

```
cambiar2 :: Arbol Int -> Arbol Int
cambiar2 (Hoja n)         = Hoja (if n==2 then 42
                                           else n)
cambiar2 (Nodo n t1 t2) = Node n (cambiar2 t1)
                                  (cambiar2 t2)
```

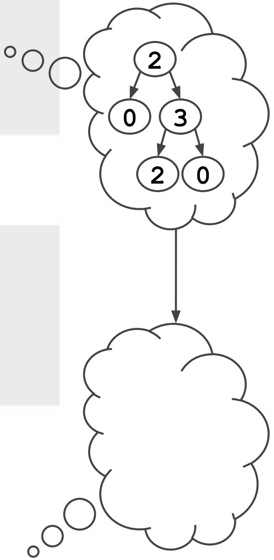
  - El término "cambiar" es engañoso
    - El árbol resultante es *OTRO*
    - No podemos "cambiar" un valor, sin cambiar *DE* valor
      - Ej. "cambiar" el número 2 sumándole 40…

Árboles binarios: naturaleza de lo

```
cambiar2 (Hoja n)          =
     Hoja (if n==2 then 42
                   else n)
cambiar2 (Nodo n t1 t2) =
     Node n (cambiar2 t1)
            (cambiar2 t2)
```

- Algunas cuestiones sobre cómo hablar de árboles

```
aej1 = Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0))
  cambiar2 aej1
= ...
```



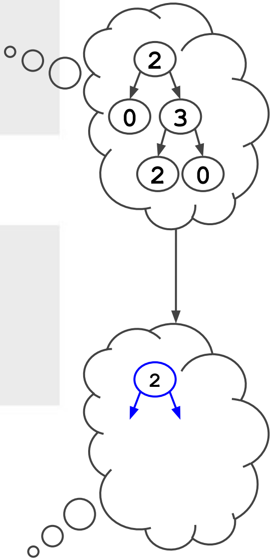
Árboles binarios: naturaleza de lo

```
cambiar2 (Hoja n)          =
     Hoja (if n==2 then 42
                   else n)
cambiar2 (Nodo n t1 t2) =
     Node n (cambiar2 t1)
            (cambiar2 t2)
```

- Algunas cuestiones sobre cómo hablar de árboles

```
aej1 = Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0))

  cambiar2 aej1
= Nodo 2 (cambiar2 (Hoja 0))
         (cambiar2 (Nodo 3 (Hoja 2) (Hoja 0)))
```



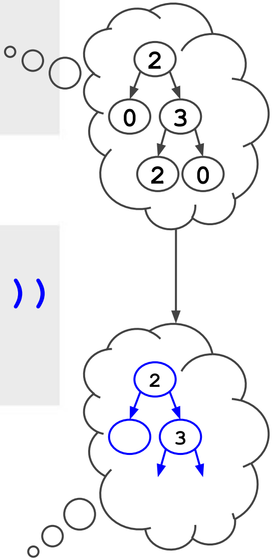
Árboles binarios: naturaleza de lo

```
cambiar2 (Hoja n)          =
     Hoja (if n==2 then 42
                    else n)
cambiar2 (Nodo n t1 t2) =
     Node n (cambiar2 t1)
            (cambiar2 t2)
```

- Algunas cuestiones sobre cómo hablar de árboles

```
aej1 = Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0))

  cambiar2 aej1
= Nodo 2 (cambiar2 (Hoja 0))
         (cambiar2 (Nodo 3 (Hoja 2) (Hoja 0)))
= Nodo 2 (Hoja (if 0==2 then 42 else 0))
         (Nodo 3 (cambiar2 (Hoja 2)) (cambiar2 (Hoja 0)))
```



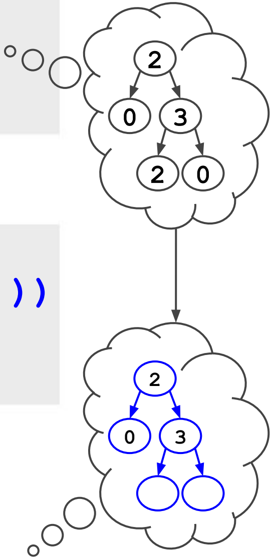
Árboles binarios: naturaleza de lo

```
cambiar2 (Hoja n)          =
     Hoja (if n==2 then 42
                  else n)
cambiar2 (Nodo n t1 t2) =
     Node n (cambiar2 t1)
            (cambiar2 t2)
```

- Algunas cuestiones sobre cómo hablar de árboles

```
aej1 = Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0))

  cambiar2 aej1
= Nodo 2 (cambiar2 (Hoja 0))
         (cambiar2 (Nodo 3 (Hoja 2) (Hoja 0)))
= Nodo 2 (Hoja (if 0==2 then 42 else 0))
         (Nodo 3 (cambiar2 (Hoja 2)) (cambiar2 (Hoja 0)))
= Nodo 2 (Hoja 0)
         (Nodo 3 (Hoja (if 2==2 then 42 else 2))
                 (Hoja (if 0==2 then 42 else 0)))
```



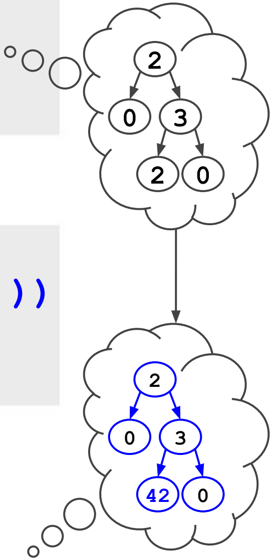
Árboles binarios: naturaleza de lo

```
cambiar2 (Hoja n)         =
     Hoja (if n==2 then 42
                  else n)
cambiar2 (Nodo n t1 t2) =
     Node n (cambiar2 t1)
            (cambiar2 t2)
```

- Algunas cuestiones sobre cómo hablar de árboles

```
aej1 = Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0))

  cambiar2 aej1
= Nodo 2 (cambiar2 (Hoja 0))
         (cambiar2 (Nodo 3 (Hoja 2) (Hoja 0)))
= Nodo 2 (Hoja (if 0==2 then 42 else 0))
         (Nodo 3 (cambiar2 (Hoja 2)) (cambiar2 (Hoja 0)))
= Nodo 2 (Hoja 0)
         (Nodo 3 (Hoja (if 2==2 then 42 else 2))
                 (Hoja (if 0==2 then 42 else 0)))
= Nodo 2 (Hoja 0)(Nodo 3 (Hoja 42) (Hoja 0))
```



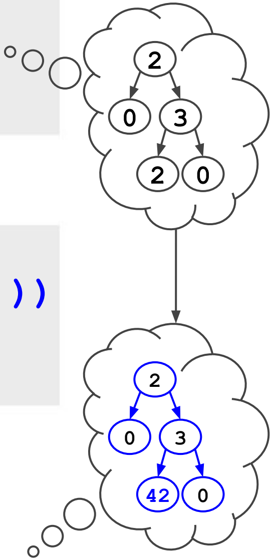
Árboles binarios: naturaleza de lo

```
cambiar2 (Hoja n)          =
     Hoja (if n==2 then 42
                    else n)
cambiar2 (Nodo n t1 t2) =
     Node n (cambiar2 t1)
            (cambiar2 t2)
```

- Algunas cuestiones sobre cómo hablar de árboles

```
aej1 = Nodo 2 (Hoja 0) (Nodo 3 (Hoja 2) (Hoja 0))

  cambiar2 aej1
= Nodo 2 (cambiar2 (Hoja 0))
         (cambiar2 (Nodo 3 (Hoja 2) (Hoja 0)))
= Nodo 2 (Hoja (if 0==2 then 42 else 0))
         (Nodo 3 (cambiar2 (Hoja 2)) (cambiar2 (Hoja 0)))
= Nodo 2 (Hoja 0)
         (Nodo 3 (Hoja (if 2==2 then 42 else 2))
                 (Hoja (if 0==2 then 42 else 0)))
= Nodo 2 (Hoja 0)(Nodo 3 (Hoja 42) (Hoja 0))
```




```
cambiar2 aej1 = Nodo 2 (Hoja 0)(Nodo 3 (Hoja 42) (Hoja 0))
```

Árboles binarios: recursión estructural

- Más funciones por recursión estructural en el árbol
  - Duplicar cada uno de los elementos

```
duplA :: Arbol Int -> Arbol Int
duplA (Hoja n)         = … n …
duplA (Nodo n t1 t2) = … n … duplA t1 … duplA t2 …
```

  - Describir la suma de todos los elementos

```
sumA :: Arbol Int -> Int
sumA (Hoja n)          = …
sumA (Nodo n t1 t2)    =  … n … sumA t1 … sumA t2 …
```

Árboles binarios: recursión estructural

- Más funciones por recursión estructural en el árbol
  - Duplicar cada uno de los elementos

```
duplA :: Arbol Int -> Arbol Int
duplA (Hoja n)        = Hoja (n*2)
duplA (Nodo n t1 t2) = Nodo (n*2) (duplA t1) (duplA t2)
```

  - Describir la suma de todos los elementos

```
sumA :: Arbol Int -> Int
sumA (Hoja n)          = n
sumA (Nodo n t1 t2)   = n + sumA t1 + sumA t2
```

Árboles binarios: recursión estructural

- Más funciones por recursión estructural en el árbol
  - Listado en orden de los elementos

```
inorder :: Arbol a -> [a]
inorder (Hoja x)        = … x …
inorder (Nodo x t1 t2)  = … inorder t1 … x … inorder t2 …
```

  - Listado preorden de los elementos

```
preorder :: Arbol a -> [a]
preorder (Hoja x)       = … x …
preorder (Nodo x t1 t2) = … x … preorder t1 … preorder t2 …
```

Árboles binarios: recursión estructural

- Más funciones por recursión estructural en el árbol
  - Listado en orden de los elementos

```
inorder :: Arbol a -> [a]
inorder (Hoja x)          = [x]
inorder (Nodo x t1 t2)    = inorder t1 ++ [x] ++ inorder t2
```

  - Listado preorden de los elementos

```
preorder :: Arbol a -> [a]
preorder (Hoja x)         = [x]
preorder (Nodo x t1 t2) = [x] ++ preorder t1 ++ preorder t2
```

Árboles binarios: recursión estructural

- Más funciones por recursión estructural en el árbol
  - Listado en orden de los elementos

```
inorder :: Arbol a -> [a]
inorder (Hoja x)          = [x]
inorder (Nodo x t1 t2)   = inorder t1 ++ [x] ++ inorder t2
```

  - Listado preorden de los elementos

```
preorder :: Arbol a -> [a]
preorder (Hoja x)         = [x]
preorder (Nodo x t1 t2) = [x] ++ preorder t1 ++ preorder t2
```

  - ¿Cómo sería un listado posorden?

Árboles binarios: características

- ¿Cuántos elementos de tipo `a` tiene un `Arbol a`?
  - Cantidad de elementos

```
sizeA :: Arbol a -> Int
sizeA (Hoja x)       = 1
sizeA (Nodo x t1 t2) = 1 + sizeA t1 + sizeA t2
```

  - ¿El valor obtenido tiene alguna particularidad interesante?

Árboles binarios: características

- ¿Cuántos elementos de tipo **a** tiene un **Arbol a**?
  - Cantidad de elementos

```
sizeA :: Arbol a -> Int
sizeA (Hoja x)        = 1
sizeA (Nodo x t1 t2) = 1 + sizeA t1 + sizeA t2
```

  - ¿El valor obtenido tiene alguna particularidad interesante?
    - ¡Nunca es un número par!
    - Ningún **Arbol** vincula una cantidad par de datos
      - ¡No hay árboles de 2 elementos!

Árboles binarios: inducción estructural

- **Prop:** ¿para todo **t**`::Arbol A`. `sizeA` **t** es impar?
  Dem: sea **t'** un árbol. Por ppio. de ind. sobre la estructura de **t'**
  Caso base) ¿`sizeA (Hoja x)` es impar?
  Caso ind.) **HI_1)** ¡`sizeA` t_1 es impar!
  HI_2) ¡`sizeA` t_2 es impar!
  TI) ¿`sizeA (Nodo` x t_1 t_2`)` es impar?
  - **Dem:** (informal)
    - El caso base es trivialmente `1` (por `sizeA`.1), que es impar
    - En el caso inductivo, basta recordar que la suma de dos impares (dados por las **HIs**) es par, y al sumarle `1`, da como resultado un número impar (luego de usar `sizeA`.2)

Más sobre árboles

Más sobre árboles

- No existe un único tipo de árboles

```
data Arbol     a = Hoja a | Nodo   a (Arbol a)     (Arbol a)
data Tree      a = EmptyT | NodeT a (Tree a)      (Tree a)
data TipTree   a = Tip   a | Branch   (TipTree a)   (TipTree a)
data ABTree a b = Leaf b | Node   a (ABTree a b) (ABTree a b)
data ExpTree a b c = Atom c
                   | BinOp a (ExpTree a b c) (ExpTree a b c)
                   | UnOp b (ExpTree a b c)
data QuadTree a = LeafQ a | NodeQ (QuadTree a) (QuadTree a)
                                  (QuadTree a) (QuadTree a)
```

- Hay muchas más variantes posibles

Más sobre árboles

- No existe un único tipo de árboles

```
Nodo 2 (Hoja 3) (Hoja 4) :: Arbol Int
NodeT 42 EmptyT (NodeT 17 EmptyT EmptyT) :: Tree Int
Branch (Branch (Tip 10) (Tip 20)) (Tip 30) :: TipTree  Int
Node "a" (Leaf 12) (Leaf 13) :: ABTree String Int
Node 14 (Leaf "b") (Leaf "c") :: ABTree Int String
Node True (Leaf succ) (Leaf id) :: ABTRee Bool (Int->Int)
BinOp "a" (UnOp True (Atom 99))
          (Atom 66) :: ExpTree String Bool Int
NodeQ (LeafQ Rojo) (LeafQ Azul)
      (LeafQ Negro) (LeafQ Rojo) :: QuadTree Color
```

  - Hay similitudes…

Árboles binarios: recursión estructural

- Límites de la recursión estructural sobre árboles binarios
  - Insertar un elemento en un BST

```
insertA :: a -> Tree a -> Tree a
              -- PRECOND: el argumento es un BST
insertA x EmptyT             = NodeT x EmptyT EmptyT
insertA x (NodeT y t1 t2) =
    if x==y      then NodeT x t1            t2
     else if x<y then NodeT y (insertA x t1) t2
                 else NodeT y t1    (insertA x t2)
```

  - No toda función recursiva sobre árboles es estructural

Más sobre árboles

- Se pueden encontrar relaciones entre tipos de árboles

```
data Arbol  a   = Hoja a | Nodo a (Arbol a)    (Arbol a)
data ABTree a b = Leaf b | Node a (ABTree a b) (ABTree a b)

type ArbolD a = ABTree a a
a2ad :: Arbol a -> ArbolD a
ad2a :: ArbolD a -> Arbol a
```

Prop: a) `a2ad . ad2a = id      -- (::??)`
b) `ad2a . a2ad = id      -- (::??)`

Más sobre árboles

- Se pueden encontrar relaciones entre tipos de árboles

```
data Arbol  a   = Hoja a | Nodo a (Arbol a)     (Arbol a)
data ABTree a b = Leaf b | Node a (ABTree a b)  (ABTree a b)

type ArbolD a = ABTree a a          -- Nodo 2 (Hoja 3) (Hoja 4)
a2ad :: Arbol a -> ArbolD a         -- Node 2 (Leaf 3) (Leaf 4)
ad2a :: ArbolD a -> Arbol a         -- ¿ Node "a" (Leaf 3) (Leaf 4) ?
```

Prop: a) `a2ad . ad2a = id`   `-- :: ArbolD a -> ArbolD a`

b) `ad2a . a2ad = id`   `-- :: Arbol a -> Arbol a`

- El tipo **`Arbol`** es un caso particular de **`ABTree`**

Más sobre árboles

- Se pueden encontrar relaciones entre tipos de árboles

```
data ABTree  a   b = Leaf b | Node  a (ABTree a b)     (ABTree a b)
data ExpTree a b c = Atom c | BinOp a (ExpTree a b c) (ExpTree a b c)
                            | UnOp  b (ExpTree a b c)

abt2ext :: ABTree a b -> ExpTree a () b    -- Node  "a" (Leaf 3) (Leaf 4)
ext2abt :: ExpTree a () b -> ABTree a b    -- BinOp "a" (Atom 3) (Atom 4)
    -- PRECOND: el argumento no usa el constructor UnOp
```

Prop: a) `ext2abt . abt2ext = id` -- `(:: ABTree a b -> ABTree a b)`

b) para todo **et** `:: ExpTree a`. si **et** no usa el constructor `UnOp`

entonces `abt2ext (ext2abt et) = et`

- `ABTree` es estructuralmente similar a un *subconjunto* de `ExpTree`
  - el tipo `()` podría ser reemplazado por cualquier otro (no se usa)

Más sobre árboles

- Se pueden encontrar relaciones entre tipos de árboles

```
data Arbol   a = Hoja a | Nodo  a (Arbol a)   (Arbol a)
data Tree    a = EmptyT | NodeT a (Tree a)    (Tree a)
data TipTree a = Tip  a | Branch  (TipTree a) (TipTree a)

separateT :: Arbol a -> (Tree a, TipTree a)     -- Branch  (Tip 3)  (Tip 4)
fuseT :: (Tree a, TipTree a) -> ABTree a b      -- Node  2 (Hoja 3) (Hoja 4)
  -- PRECOND: sin las partes no inductivas,     -- NodeT 2 EmptyT   EmptyT
              la estructura de los datos es la misma
```

Prop: a) `fuseT . separateT = id   -- (:: Arbol a -> Arbol a)`

b) para todo `(it,ft)::(Tree a, TipTree a)`.

si **it** y **ft** tienen la misma estructura (ignorando las partes no inductivas)

entonces `separateT (fuseT (it,ft)) = (it,ft)`

- Todo **`Arbol`** se puede separar en 2 partes “estructuralmente” iguales

Más sobre árboles

- Árboles de "conocimiento" (o árboles de *decisión*)

```
data ABTree a b = Leaf b | Node  a (ABTree a b) (ABTree a b)
type KnowledgeTree s a = ABTree (s -> Bool) a
```

```
decide :: KnowledgeTree data answer -> (data -> answer)
decide (Leaf a)       d = a
decide (Node f d1 d2) d = if f d then decide d1 d
                                 else decide d2 d
```

  - El árbol permite almacenar algún tipo de "conocimiento"
    - Basado en preguntas por sí o no
  - Permite tomar decisiones en base al mismo
    - Podría modificarse la estructura para "aprender"

Más sobre árboles

- Los elementos de un árbol pueden ser funciones

```
data Sintoma = Fiebre | Decaimiento | PerdidaDelGusto | …
data Diagnostico = Gripe | ActivarProtocolo
                 | ConsultarAlMedico | EntregarCertHabilitante

autoDoctor :: KnowledgeTree [Sintoma] Diagnostico
autoDoctor = Node f1 (Node f2 (Leaf ActivarProtocolo)
                              (Leaf Gripe))
                     (Node f3 (Leaf EntregarCertHabilitante)
                              (Leaf ConsultarAlMedico))
 where f1 sintomas = Fiebre `elem` sintomas
       f2 sintomas = Decaimiento     `elem` sintomas
                  || PerdidaDelGusto `elem` sintomas
       f3 sintomas = null sintomas
```

Más sobre árboles

- Representación de árboles mediante funciones

```
data FTree a = FT ([Dir] -> Maybe a)
data Dir = Izq | Der

t2ft :: Tree a -> FTree a
t2ft EmptyT          = FT (\ds -> Nothing)
t2ft (NodeT x t1 t2) = FT (\ds -> genNodeFT x (t2ft t1)
                                               (t2ft t2) ds)
  where genNodeFT x _        _        []        = Just x
        genNodeFT _ (FT f1) _        (Izq:ds') = f1 ds'
        genNodeFT _ _        (FT f2) (Der:ds') = f2 ds'
```

- ¡El árbol se representa con una función!

Más sobre árboles

- Representación de árboles mediante funciones

```
data FTree a = FT ([Dir] -> Maybe a)
data Dir = Izq | Der

t2ft (NodeT 20 (NodeT 10 EmptyT EmptyT)
               (NodeT 30 EmptyT EmptyT))
  = FT (\ds -> case ds of
                 []    -> Just 20
                 [Izq] -> Just 10
                 [Der] -> Just 30
                 _     -> Nothing)
```

- ¡La función *ES* la estructura!

Más sobre árboles

- Representación de árboles mediante funciones

```
data FTree a = FT ([Dir] -> Maybe a)
data Dir = Izq | Der

ft2t :: FTree a -> Tree a                 -- PRECOND: si la función da
ft2t (FT f) = generateFrom f []           --  Nothing en una lista, entonces
  where generateFrom f ds =               --  da Nothing en todas las que
            case f ds of                  --  la tengan de prefijo
              Nothing -> EmptyT
              Just x  -> NodeT x (generateFrom f (ds++[Izq]))
                                 (generateFrom f (ds++[Der]))
```

- No es recursión estructural… (¡Puede generar árboles infinitos!)
- `Tree a` es "estructuralmente" equivalente a un subconjunto de `([Dir] -> Maybe a)`

Más sobre árboles

- Representación de árboles mediante funciones

```
data FTree a = FT ([Dir] -> Maybe a)
data Dir = Izq | Der

ft2t (FT (\_ -> Just 42))
  = ??
```

Más sobre árboles

- Representación de árboles mediante funciones

```
data FTree a = FT ([Dir] -> Maybe a)
data Dir = Izq | Der

ft2t (FT (\_ -> Just 42))
  = theAnswerToLifeUniverseAndEverythingTree

theAnswerToLifeUniverseAndEverythingTree =
    NodeT 42 theAnswerToLifeUniverseAndEverythingTree
             theAnswerToLifeUniverseAndEverythingTree
```

Más sobre árboles

- Representación de árboles mediante funciones

```
data FTree a = FT ([Dir] -> Maybe a)
data Dir = Izq | Der

ft2t (FT (\_ -> Just 42))
  = theAnswerToLifeUniverseAndEverythingTree

theAnswerToLifeUniverseAndEverythingTree =
    NodeT 42 theAnswerToLifeUniverseAndEverythingTree
             theAnswerToLifeUniverseAndEverythingTree
```

- El árbol se ramifica infinitamente
  - ¡No es un elemento inductivo!

Representar un lenguaje de programación

Representar un lenguaje de programación

- ¿Cómo representar un lenguaje de programación imperativo simple?

Representar un lenguaje de programación

- ¿Cómo representar un lenguaje de programación imperativo simple?
  - Estructura sintáctica
  - Significado
  - Manipulación simbólica

Re

Representar un lenguaje de programación

- ¿Cómo representar un lenguaje de programación imperativo simple?
  - Estructura sintáctica
  - Significado
  - Manipulación simbólica

"No se pierda la próxima clase"

Resumen

Resumen

- Representación de expresiones aritméticas
  - Significado, manipulación simbólica y coherencia
  - Diferentes representaciones, ventajas y desventajas
- Árboles
  - Árboles binarios
    - Definición, recursión y principio de inducción estructural
  - Otros tipos de árboles
  - Árboles, estructuras y alto orden