



Programación Funcional

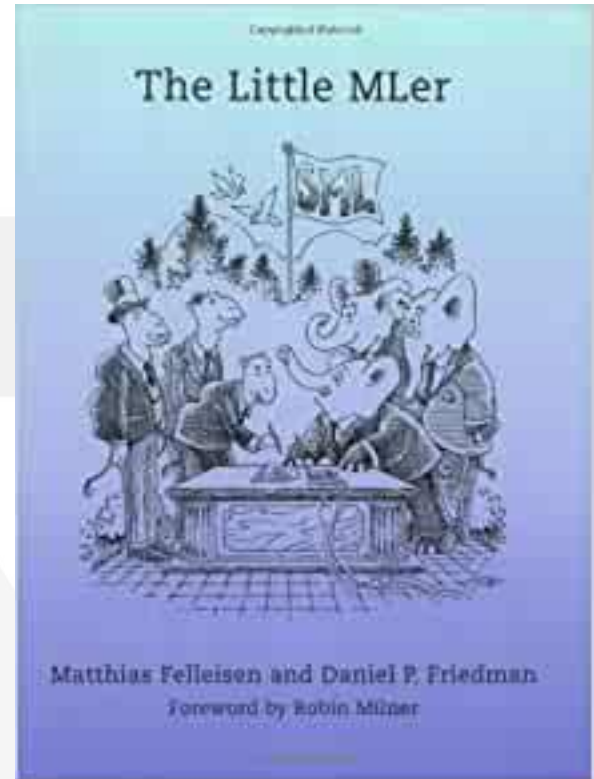
Clases teóricas

por Pablo E. “Fidel” Martínez López

10. Inducción y recursión IV

“There are many ways of trying to understand programs. People often rely too much on one way, which is called 'debugging' and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very program themselves.”

Foreword to *“The Little MLer”*
Robin Milner





Escenas del capítulo anterior

Representar un lenguaje de programación

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



Expresiones aritméticas con variables

Expresiones aritméticas con variables

- ❑ ¿Qué elementos tiene un lenguaje de programación?
 - ❑ Uno de esos elementos es *expresiones con variables*

$x+1$

$x+x$

$a*(x^2) + b*x + c$

- ❑ ¿Cómo enriquecer las **ExpA** para que tengan variables?
 - ❑ Se precisa un nuevo constructor para variables
- ❑ ¿Cómo cambia el significado si lo hacemos?
 - ❑ ¿Cuál es el significado de una variable?
 - ❑ ¿Los demás significados cambian?

Expresiones aritméticas con variables

❑ ¿Cómo agregar *variables* a las expresiones aritméticas?

❑ Constructor para variables

```
data NExp = Var Variable                -- Variables
          | NCte Int                    -- Constantes
          | NBOp NBinOp NExp NExp      -- Operaciones binarias
data NBinOp = Add | Sub | Mul | Div | Mod | Pow
type Variable = String
```

❑ $x+1$ se representa como `NBOp Add (Var "x") (Cte 1)`

❑ $a*(x^2)$ se representa como

```
      NBOp Mul (Var "a")
          (NBOp Pow (Var "x") (NCte 2))
```

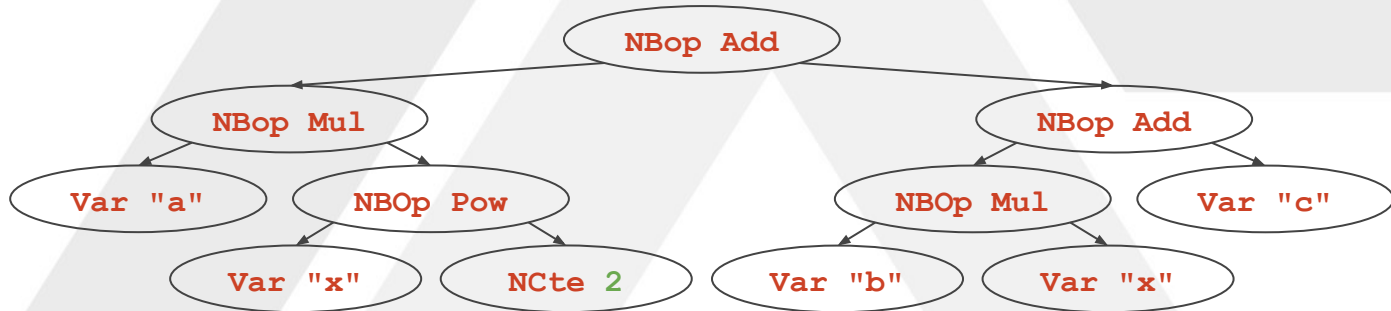
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBop NBinOp NExp NExp
```

```
NBop Add (NBop Mul (Var "a")  
                  (NBop Pow (Var "x") (NCte 2)))  
  (NBop Add (NBop Mul (Var "b") (Var "x"))  
            (Var "c"))
```

$a * (x^2) + b * x + c$



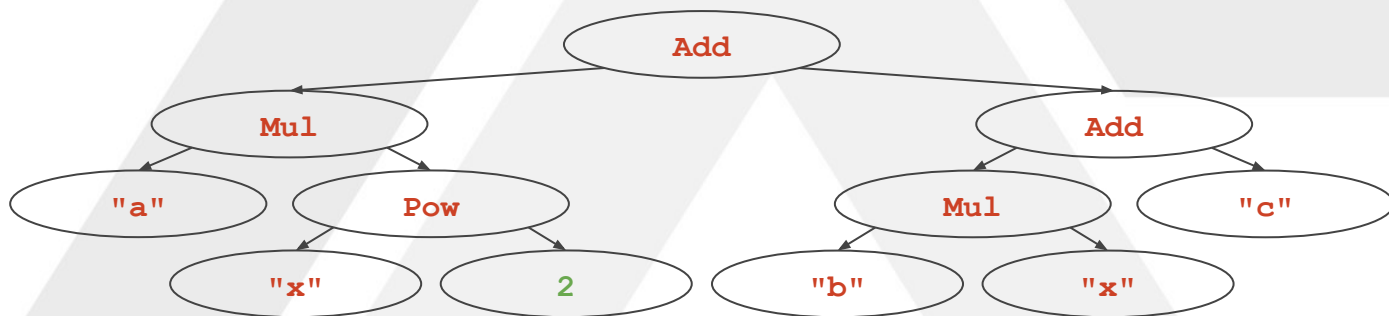
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
                  (NBOp Pow (Var "x") (NCte 2)))  
  (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
            (Var "c"))
```

$a * (x^2) + b * x + c$



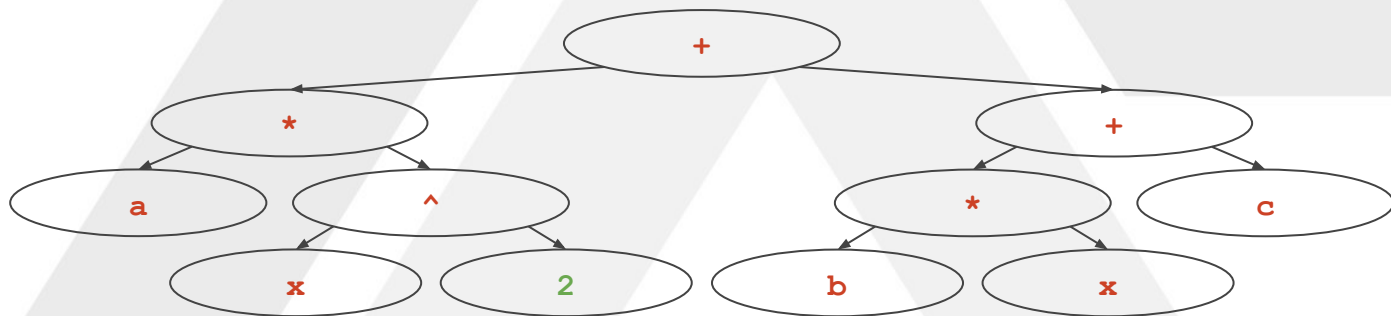
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
    (NBOp Pow (Var "x") (NCte 2)))  
    (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
        (Var "c"))
```

$a * (x^2) + b * x + c$



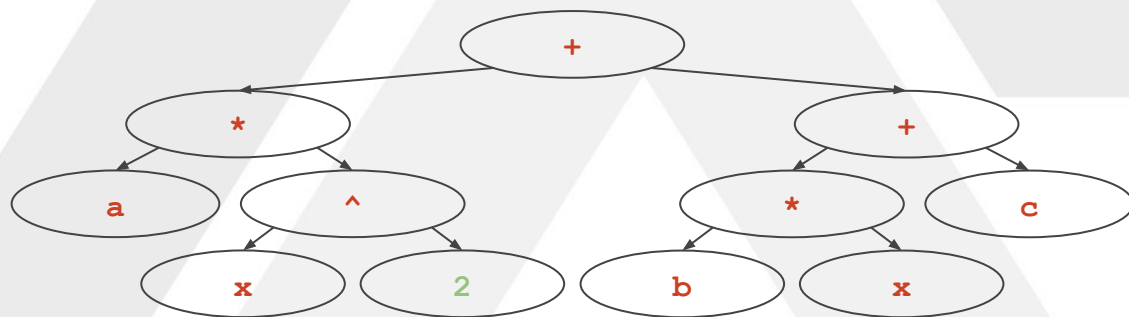
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
                  (NBOp Pow (Var "x") (NCte 2)))  
  (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
            (Var "c"))
```

$a * (x^2) + b * x + c$



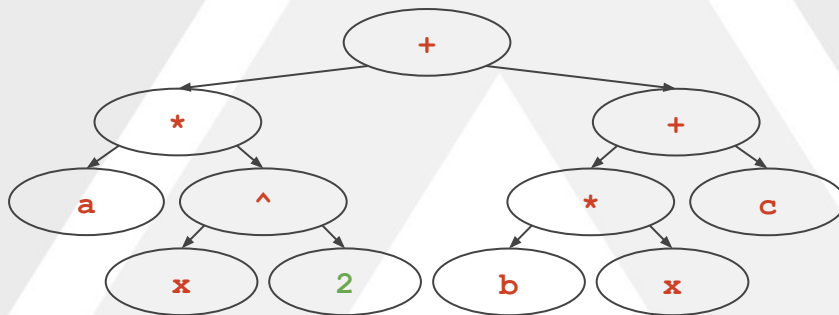
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
  (NBOp Pow (Var "x") (NCte 2)))  
  (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
    (Var "c"))
```

$a * (x^2) + b * x + c$



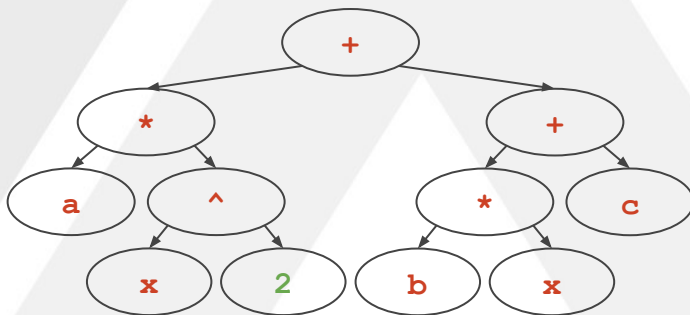
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
                  (NBOp Pow (Var "x") (NCte 2)))  
  (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
            (Var "c"))
```

$a * (x^2) + b * x + c$



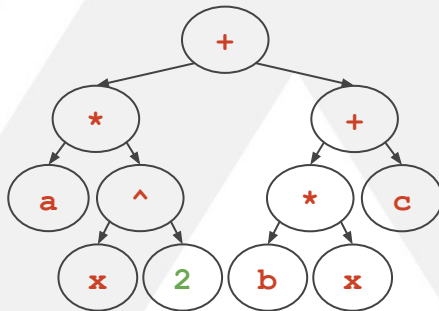
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
                  (NBOp Pow (Var "x") (NCte 2)))  
  (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
            (Var "c"))
```

$a * (x^2) + b * x + c$



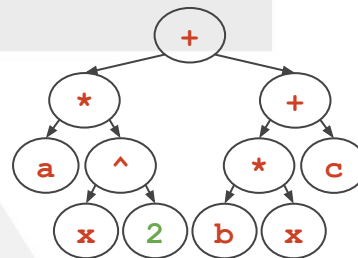
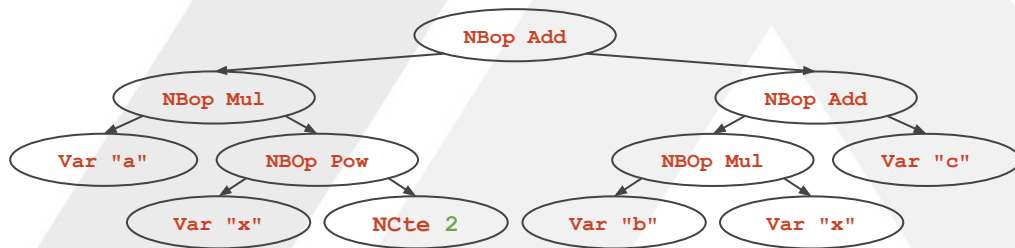
Expresiones aritméticas con variables

■ Expresiones con variables, gráficamente

```
data NExp = Var Variable | NCte Int | NBOp NBinOp NExp NExp
```

```
NBOp Add (NBOp Mul (Var "a")  
                  (NBOp Pow (Var "x") (NCte 2)))  
  (NBOp Add (NBOp Mul (Var "b") (Var "x"))  
            (Var "c"))
```

$a * (x^2) + b * x + c$



Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿Es un número el significado de una expresión?
 - ❑ ¿Cuál es el valor *número*ico de **x+1**?

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿Es un número el significado de una expresión?
 - ❑ ¿Cuál es el valor *númeroico* de **x+1**?
 - ❑ ¿Qué debería preguntarse para poder dar un número?

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBinOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿Es un número el significado de una expresión?
 - ❑ ¿Cuál es el valor *número*ico de **x+1**?
 - ❑ ¿Qué debería preguntarse para poder dar un número?
 - ❑ ¡Claro! ¡El valor de **x**!

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿Es un número el significado de una expresión?
 - ❑ ¿Cuál es el valor *número*ico de **x+1**?
 - ❑ ¿Qué debería preguntarse para poder dar un número?
 - ❑ ¡Claro! ¡El valor de **x**!
 - ¿Cuánto vale **x+1**? `evalNExp :: NExp -> ...`

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿Es un número el significado de una expresión?
 - ❑ ¿Cuál es el valor *númeroico* de **x+1**?
 - ❑ ¿Qué debería preguntarse para poder dar un número?
 - ❑ ¡Claro! ¡El valor de **x**!
 - ¿Cuánto vale **x+1**?
 - ¿Cuánto vale **x**?

```
evalNExp :: NExp -> ...
decimeCuantoVale :: ??
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿Es un número el significado de una expresión?
 - ❑ ¿Cuál es el valor *número*ico de **x+1**?
 - ❑ ¿Qué debería preguntarse para poder dar un número?
 - ❑ ¡Claro! ¡El valor de **x**!
 - ¿Cuánto vale **x+1**?
 - ¿Cuánto vale **x**?

```
evalNExp :: NExp -> ...
```

```
decimeCuantoVale :: Variable -> ...
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ ¿Es un número el significado de una expresión?

❑ ¿Cuál es el valor *número*ico de **x+1**?

❑ ¿Qué debería preguntarse para poder dar un número?

❑ ¡Claro! ¡El valor de **x**!

— ¿Cuánto vale **x+1**?

— ¿Cuánto vale **x**?

— **16**

```
evalNExp :: NExp -> ...
```

```
decimeCuantoVale :: Variable -> Int
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ ¿Es un número el significado de una expresión?

❑ ¿Cuál es el valor *número*ico de **x+1**?

❑ ¿Qué debería preguntarse para poder dar un número?

❑ ¡Claro! ¡El valor de **x**!

— ¿Cuánto vale **x+1**?

— ¿Cuánto vale **x**?

— **16**

— Entonces, **17**

```
evalNExp :: NExp -> ... -> Int
```

```
decimeCuantoVale :: Variable -> Int
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ ¿Es un número el significado de una expresión?

❑ ¿Cuál es el valor *númeroico* de **x+1**?

❑ ¿Qué debería preguntarse para poder dar un número?

❑ ¡Claro! ¡El valor de **x**!

— ¿Cuánto vale **x+1**?

— ¿Cuánto vale **x**?

— **16**

— Entonces, **17**

```
evalNExp :: NExp -> ... -> Int
```

```
decimeCuantoVale :: Variable -> Int
```

❑ Las funciones pueden representar preguntas...

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
             | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿A quién o qué preguntarle el valor de una variable?
 - ❑ ¿Qué usan las personas para recordar?
 - ❑ ¿Qué se usa en programación para guardar información?

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿A quién o qué preguntarle el valor de una variable?
 - ❑ ¿Qué usan las personas para recordar?
 - ❑ ¿Qué se usa en programación para guardar información?
 - ❑ Una **memoria**
 - ❑ ¿Cómo representamos la memoria?

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ ¿A quién o qué preguntarle el valor de una variable?
 - ❑ ¿Qué usan las personas para recordar?
 - ❑ ¿Qué se usa en programación para guardar información?
 - ❑ Una **memoria**
 - ❑ ¿Cómo representamos la memoria?
 - ❑ Cualquier tipo que nos permita recordar y consultar valores asociados a una variable...
 - ❑ ¡Un tipo abstracto!
 - ❑ **data Memoria**

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ Tipo abstracto de datos para recordar valores

```
data Memoria          -- Tipo abstracto de datos
enBlanco :: Memoria
    -- Una memoria vacía, que no recuerda nada
cuantoVale :: Variable -> Memoria -> Maybe Int
    -- El valor recordado para la variable, si existe
recordar :: Variable -> Int -> Memoria -> Memoria
    -- Memoria con el recuerdo del valor para la variable
variables :: Memoria -> [ Variable ]
    -- Las variables que recuerda
```

Expresiones aritméticas con memoria

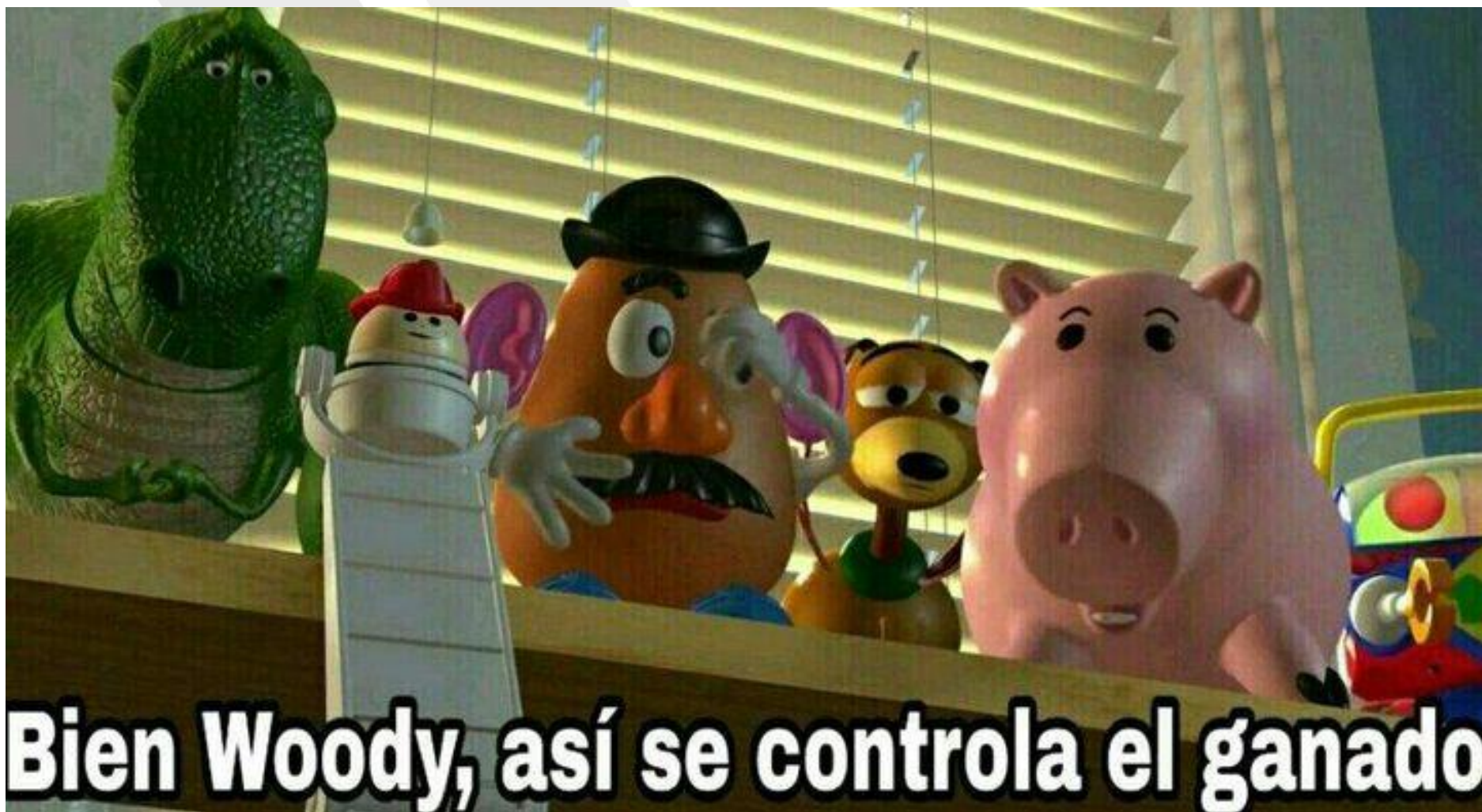
```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Algunas propiedades que debe cumplir la memoria
 - ❑ para todo **x**. **cuantoVale** **x** **enBlanco** = **Nothing**
 - ❑ para todo **m**. para todo **x**. para todo **n**.
cuantoVale **x** (**recordar** **x** **n** **m**) = **Just** **n**
 - ❑ para todo **m**. para todo **x**. para todo **y**. para todo **n**.
si **x** **≠** **y** entonces
cuantoVale **x** (**recordar** **y** **n** **m**) = **cuantoVale** **x** **m**
 - ❑ para todo **m**. para todo **x**. **x** **`elem`** **variables** **m** = **True**
es equivalente a **cuantoVale** **x** **m** **≠** **Nothing**

Disgresión: Tipos Abstractos de Datos

- ❑ ¿Cómo saber qué hace un Tipo Abstracto de Datos?
 - ❑ Se usan *propiedades* sobre las operaciones de la interfaz
 - ❑ Estas propiedades son equivalencias denotacionales entre diferentes combinaciones de las operaciones
 - ❑ De esta forma se puede establecer de forma completa el comportamiento esperado de las operaciones de un TAD **SIN** dar ninguna implementación
 - ❑ **Especificaciones algebraicas** de TADs
 - ❑ (otro de los países de NO serán visitados en este viaje)





Expresiones aritméticas con memoria

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Algunas propiedades que debe cumplir la memoria
 - ❑ para todo **x**. **cuantoVale** **x** **enBlanco** = **Nothing**
 - ❑ para todo **m**. para todo **x**. para todo **n**.
cuantoVale **x** (**recordar** **x** **n** **m**) = **Just** **n**
 - ❑ para todo **m**. para todo **x**. para todo **y**. para todo **n**.
si **x** **≠** **y** entonces
cuantoVale **x** (**recordar** **y** **n** **m**) = **cuantoVale** **x** **m**
 - ❑ para todo **m**. para todo **x**. **x** **`elem`** **variables** **m** = **True**
es equivalente a **cuantoVale** **x** **m** **≠** **Nothing**

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Mediante una función de asignación de significado

evalNExp :: **NExp** -> ??

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Mediante una función de asignación de significado

```
evalNExp :: NExp -> (... -> Int)
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Mediante una función de asignación de significado

```
evalNExp :: NExp -> (Memoria -> Int)
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Mediante una función de asignación de significado

evalNExp :: NExp -> (Memoria -> Int)

- ❑ ¡El significado es una función!
 - ❑ O sea, se responde una pregunta con otra...
(P: ¿Cuál es el valor de esta expresión?
R: ¿Cuánto valen las variables?)

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?

- ❑ Mediante una función de asignación de significado

evalNExp :: NExp -> Memoria -> Int

- ❑ ¡El significado es una función!

- ❑ O sea, se responde una pregunta con otra...

- (P: ¿Cuál es el valor de esta expresión?

- R: ¿Cuánto valen las variables?)

- ❑ Leyéndolo en francés, se necesitan *dos* argumentos para poder dar el número correspondiente

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
 - ❑ Mediante una función de asignación de significado
$$\text{evalNExp} :: \text{NExp} \rightarrow (\text{Memoria} \rightarrow \text{Int})$$
 - ❑ ¡El significado es una función!
 - ❑ O sea, se responde una pregunta con otra...
(P: ¿Cuál es el valor de esta expresión?
R: ¿Cuánto valen las variables?)
 - ❑ Leyéndolo en francés, se necesitan *dos* argumentos para poder dar el número correspondiente

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ Mediante una función de asignación de significado

```
evalNExp :: NExp -> (Memoria -> Int)
evalNExp (Var x) mem =
  case cuantoVale x mem of
    Nothing -> error ("Variable "++x++" indefinida")
    Just v   -> v
evalNExp (NCte n) mem = n
evalNExp (NBOp bop ne1 ne2) mem =
  evalNBOp bop (evalNExp ne1 mem)
                (evalNExp ne2 mem)
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ ¿Cómo darle significado a una **NExp**?
- ❑ Mediante una función de asignación de significado

```
evalNExp :: NExp -> (Memoria -> Int)
evalNExp (Var x) mem =
  case cuantoVale x mem of
    Nothing -> error ("Variable "++x++" indefinida")
    Just v   -> v
evalNExp (NCte n) =
evalNExp (NBOp bop) =
  evalNBOp bop (evalNExp ne1 mem)
  (evalNExp ne2 mem)
```

Algunos lenguajes dan 0, o un número cualquiera no especificado

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ Función auxiliar de significado para operaciones binarias

```
evalNBOp :: NBinOp -> (Int -> Int -> Int)
evalNBOp Add = (+)
evalNBOp Sub = (-)
evalNBOp Mul = (*)
evalNBOp Div = div
evalNBOp Mod = mod
evalNBOp Pow = (^)
```

❑ Observar el uso de currificación y alto orden
(sección de operadores)

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❑ ¿Cómo darle significado a una **NExp**?

❑ Función auxiliar de significado para operaciones binarias

```
evalNBOp :: NBinOp -> (Int -> Int -> Int)
evalNBOp Add = (+)
evalNBOp Sub = (-)
evalNBOp Mul = (*)
evalNBOp Div = div
evalNBOp Mod = mod
evalNBOp Pow = (^)
```

¡El significado de cada símbolo de operación binaria es una función binaria!

❑ Observar el uso de currificación y alto orden (sección de operadores)

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

❏ Recordar que los nombres pueden ser otros...

❏ Modificar **TB** para que sea similar a **NExp**

```
data TB = C String | D TA | E TC TB TB
```

```
data TC = F | G | H | I | J | K
```

```
evalTB :: TB -> (Mem -> Int)
```

```
evalTB (C x) mem = case memg x mem of
```

```
    Nothing -> error ""
```

```
    Just v   -> evalTA v
```

...

```
data Mem -- Tipo abstracto de datos
memf :: Mem
memg :: String -> Mem -> Maybe TA
memh :: String -> TA -> Mem -> Mem
memk :: Mem -> [ String ]
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- Recordar que los nombres pueden ser otros...
- Modificar **TB** para que sea similar a **NExp**
 - Se pueden construir transformaciones y demostrar que ambos tipos son estructuralmente equivalentes

```
nExp2tb :: NExp -> TB
```

```
tb2NExp :: TB -> NExp
```

```
Prop: tb2NExp . nExp2tb = id      -- (:: NExp -> NExp)
```

```
Prop: nExp2tb . tb2NExp = id      -- (:: TB -> TB)
```

Expresiones aritméticas con

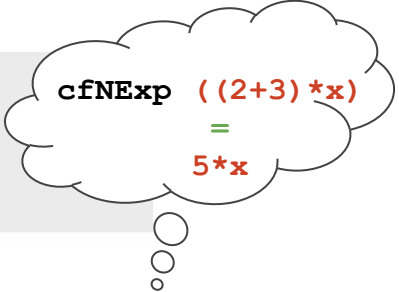
```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

Manipulación simbólica de **NExp**

Constant folding

Lo que no depende de variables se puede resolver

```
cfNExp :: NExp -> NExp
cfNExp ...
```



cfNExp ((2+3)*x)
= 5*x

```
cfNExp (NBOp Mul (NBOp Add (NCte 2) (NCte 3))
          (Var "x")) = NBOp Mul (NCte 5) (Var "x")
```

Expresiones aritméticas con

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

Manipulación simbólica de **NExp**

Constant folding

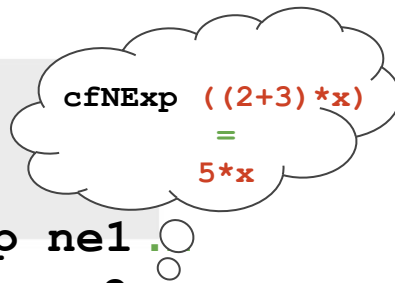
Lo que no depende de variables se puede resolver

cfNExp :: NExp -> NExp

cfNExp (Var x) = ... x ...

cfNExp (NCte n) = ... n ...

cfNExp (NBOp bop ne1 ne2) = ... bop ... **cfNExp** ne1
... **cfNExp** ne2 ...



```
cfNExp (NBOp Mul (NBOp Add (NCte 2) (NCte 3))
        (Var "x")) = NBOp Mul (NCte 5) (Var "x")
```


Expresiones aritméticas

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

- ❑ Manipulación simbólica de **NExp**
 - ❑ **Constant folding**, **cfNExp**
 - ❑ Este procesamiento se dice **estático**, porque NO depende del valor de la memoria
 - ❑ Cuando depende, se denomina **dinámico**
 - ❑ En lenguajes sin memoria se habla de **estático** o **dinámico** en función de la ejecución
 - ❑ P.ej. un *sistema de tipado estático* NO depende de la ejecución ni de valores calculados por ejecución

Expresiones aritméticas cc

```
data NExp = Var Variable | NCte Int
          | NBOp NBinOp NExp NExp
data NBinOp = Add | Sub | Mul
            | Div | Mod | Pow
type Variable = String
```

Manipulación simbólica de **NExp** y su coherencia

Constant folding

Prop: ¿para todo e . $\text{evalNExp} (\text{cfNExp } e) = \text{evalNExp } e$?

Dem: Sea ne cualquiera. Por ppio. de ind. en la estructura de ne

Caso base 1) ¿ $\text{evalNExp} (\text{cfNExp} (\text{Var } x)) = \text{evalNExp} (\text{Var } x)$?

Caso base 2) ¿ $\text{evalNExp} (\text{cfNExp} (\text{NCte } n)) = \text{evalNExp} (\text{NCte } n)$?

Caso ind.) HI1) ¿ $\text{evalNExp} (\text{cfNExp } e_1) = \text{evalNExp } e_1$!

HI2) ¿ $\text{evalNExp} (\text{cfNExp } e_2) = \text{evalNExp } e_2$!

TI) ¿ $\text{evalNExp} (\text{cfNExp} (\text{NBOp } \text{bop } e_1 e_2)) = \text{evalNExp} (\text{NBOp } \text{bop } e_1 e_2)$?



Expresiones booleanas

Expresiones booleanas

- ❑ ¿Qué otros elementos tiene un lenguaje?
 - ❑ Otro de esos elementos es *expresiones booleanas*

$x > 0$

$x == y$

$i > 0 \ \&\& \ i \leq 10$

- ❑ ¿Cómo representar expresiones booleanas?
 - ❑ Se precisa un nuevo tipo algebraico
 - ❑ La estructura será similar a la de expresiones aritméticas
- ❑ ¿Qué elementos debe tener?
 - ❑ No habrá variables booleanas...

Expresiones booleanas

❑ ¿Cómo representar expresiones booleanas?

❑ Un nuevo tipo algebraico

```
data BExp = BCte Bool          | Not BExp
          | And  BExp BExp    | Or   BExp BExp
          | ROp  RelOp NExp NExp -- Operaciones relacionales
```

```
data RelOp = Eq | NEq | Lt | LEq | Gt | GEq
```

❑ `x>0` se representa como `Rop Gt (Var "x") (Cte 0)`

❑ `i>0 && i<=10` se representa como

```
And (ROp Gt  (Var "i") (NCte 0))
    (ROp GEq  (Var "i") (NCte 10))
```

Expresiones booleanas

```
data BExp = BCte Bool | Not BExp
           | And BExp BExp | Or BExp BExp
           | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

□ Recordar que los nombres pueden ser otros...

□ Un tipo **TC**, estructuralmente similar a **BExp**

```
data TD = L Bool | M TD
         | N TD TD | Ñ TD TD | O TE TB TB
```

```
data TE = P | R | S | T | U | V
```

```
bExp2td :: BExp -> TD
```

```
td2BExp :: TD -> BExp
```

```
Prop: td2BExp . bExp2td = id      -- (:: BExp -> BExp)
```

```
Prop: bExp2td . td2BExp = id      -- (:: TD -> TD)
```

Expresiones booleanas

```
data BExp = BCte Bool | Not BExp
           | And BExp BExp | Or BExp BExp
           | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

- ❑ ¿Cómo dar significado a expresiones booleanas?
- ❑ Se necesita una memoria... ¿Por qué?

```
evalBExp :: BExp -> (Memoria -> Bool)
evalBExp ...
```

Expresiones booleanas

```
data BExp = BCte Bool | Not BExp
          | And BExp BExp | Or BExp BExp
          | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

□ ¿Cómo dar significado a expresiones booleanas?

□ Se necesita una memoria... ¿Por qué?

```
evalBExp :: BExp -> (Memoria -> Bool)
evalBExp (BCte b)           mem = ... b ...
evalBExp (Not be)          mem = ... evalBExp be ...
...
evalBExp (ROp rop ne1 ne2) mem =
    ... rop ... evalNExp ne1 ... evalNExp ne2 ...
```


Expresiones booleanas

```
data BExp = BCTe Bool | Not BExp
          | And BExp BExp | Or BExp BExp
          | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

❑ ¿Cómo dar significado a expresiones booleanas?

❑ Se necesita una memoria... ¿Por qué?

```
evalBExp :: BExp -> (Memoria -> Bool)
evalBExp (BCTe b)          mem = b
evalBExp (Not be)          mem = not (evalBExp be mem)
...
evalBExp (ROp rop ne1 ne2) mem =
    evalROp rop (evalNExp ne1 mem) (evalNExp ne2 mem)

evalROp :: RelOp -> (Int -> Int -> Bool)
evalROp Eq = (==)
...
```

Expresiones booleanas

```
data BExp = BCTe Bool | Not BExp
           | And BExp BExp | Or BExp BExp
           | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

- Manipulación simbólica de **BExp**
 - Constant folding extendida a expresiones booleanas*
 - Lo que no depende de variables se puede resolver

```
cfBExp :: NExp -> NExp
cfBExp (BCTe b) = ... b ...
cfBExp (Not be) = ... cdBExp be ...
cfBExp (And be1 be2) = ... cdBExp be1 ... cdBExp be2 ...
cfBExp (Or be1 be2) = ... cdBExp be1 ... cdBExp be2 ...
cfBExp (ROp rop ne1 ne2) = ... rop ... cfNExp ne1
                           ... cfNExp ne2 ...
```

Expresiones booleanas

```
data BExp = BCTe Bool | Not BExp
          | And BExp BExp | Or BExp BExp
          | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

- Manipulación simbólica de **BExp**
 - Constant folding extendida a expresiones booleanas*
 - Lo que no depende de variables se puede resolver

```
cfBExp :: NExp -> NExp
cfBExp (BCTe b) = BCTe b
cfBExp (Not be) = cfNot (cdBExp be)
cfBExp (And be1 be2) = cfAnd (cdBExp be1) (cdBExp be2)
cfBExp (Or be1 be2) = cfOr (cdBExp be1) (cdBExp be2)
cfBExp (ROp rop ne1 ne2) = cfROp rop (cfNExp ne1)
                           (cfNExp ne2)
```

Expresiones booleanas


```
data BExp = BCTe Bool | Not BExp
          | And BExp BExp | Or BExp BExp
          | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

- Manipulación simbólica de **BExp**
 - Constant folding extendida a expresiones booleanas*
 - Lo que no depende de variables se puede resolver

```
cfNot :: BExp -> BExp
cfNot (BCTe b) = BCTe (not b)
cfNot be      = Not be
```

```
cfAnd :: BExp -> BExp -> BExp
cfAnd (BCTe b) be2 = if b then be2 else BCTe False
cfAnd be1          be2 = And be1 be2
```

Es así porque se espera que el **And** use *short-circuit*



Expresiones booleanas

```
data BExp = BCTe Bool | Not BExp
           | And BExp BExp | Or BExp BExp
           | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

- Manipulación simbólica de **BExp**
 - Constant folding extendida a expresiones booleanas*
 - Lo que no depende de variables se puede resolver

```
cfOr :: BExp -> BExp -> BExp
cfOr (BCTe b) be2 = if b then BCTe True else be2
cfOr be1          be2 = Or be1 be2
```

```
cfROp :: NRelOp -> NExp -> NExp -> BExp
cfROp rop (NCTe n) (NCTe m) = BCTe (evalROp rop n m)
cfROp rop ne1          ne2   = ROp ne1 ne2
```

Expresiones booleanas

```
data BExp = BCTe Bool | Not BExp
          | And BExp BExp | Or BExp BExp
          | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

□ Manipulación simbólica de **BExp** y su coherencia

□ *Constant folding extendida a expresiones booleanas*

□ Prop: ¿para todo e . $\text{evalBExp} (\text{cfBExp } e) = \text{evalBExp } e$?

Dem: Sea e' cualquiera. Por ppio. de ind. en la estructura de e'

Caso base 1) ¿ $\text{evalBExp} (\text{cfBExp} (\text{BCTe } b)) = \text{evalBExp} (\text{BCTe } b)$?

Caso ind. 1) HI) ¿ $\text{evalBExp} (\text{cfBExp } be) = \text{evalBExp } be$!

TI) ¿ $\text{evalBExp} (\text{cfBExp} (\text{Not } be)) = \text{evalBExp} (\text{Not } be)$?

...

Caso base 2) ¿ $\text{evalBExp} (\text{cfBExp} (\text{ROp } \text{rop } ne_1 ne_2))$
 $= \text{evalBExp} (\text{ROp } \text{rop } ne_1 ne_2)$?

Expresiones booleanas

```
data BExp = BCTe Bool | Not BExp
          | And BExp BExp | Or BExp BExp
          | ROp RelOp NExp NExp
data RelOp = Eq | Neq | Lt | LEq | Gt | GEq
```

Manipulación simbólica de **BExp** y su coherencia

Constant folding extendida a expresiones booleanas

Prop: ¿para todo e . $\text{evalBExp} (\text{cfBExp } e) = \text{evalBExp } e$?

Dem: Sea e' cualquiera. Por **ppio. de ind. en la estructura** de e'

Caso base 1) ¿ $\text{evalBExp} (\text{cfBExp} (\text{BCTe } b)) = \text{evalBExp} (\text{BCTe } b)$?

Caso ind. 1) **HI** ¿ $\text{evalBExp} (\text{cfBExp } be) = \text{evalBExp } be$!

TI ¿ $\text{evalBExp} (\text{cfBExp} (\text{Not } be)) = \text{evalBExp} (\text{Not } be)$?

...

Caso base 2) ¿ $\text{evalBExp} (\text{cfBExp} (\text{ROp } rop \ ne_1 \ ne_2))$
 $= \text{evalBExp} (\text{ROp } rop \ ne_1 \ ne_2)$?

¡ATENCIÓN a los tipos diferentes!

Lenguaje Imperativo Simple (LIS)

Lenguaje imperativo simple

- ❑ ¿Qué forma tiene un lenguaje imperativo simple?

- ❑ Hay comandos y secuencias de comandos

```
a := N; fn := 1;  
while (a > 0)  
{ fn := a * fn; a := a - 1 }
```

- ❑ ¿Cómo representar la secuencia?

- ❑ ¿Cuáles son los comandos?

- ❑ ¿Cómo representar las partes de los comandos?

Lenguaje imperativo simple

- 📄 ¿Qué forma tiene un lenguaje imperativo simple?

- Hay comandos y secuencias de comandos

data Programa = Prog Bloque

```
type Bloque = [Comando]
```

data Comando = Assign Variable NExp

```
| If BExp Bloque Bloque
```

| While BExp Bloque

-  `fn := 1` se representa como `Program [Assign "fn" (NCte 1)]`

-  $a := N; \text{fn} := 1$

se representa como

```
Program [ Assign "a" (Var "N")
          , Assign "fn" (NCte 1) ]
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Qué forma tiene un lenguaje imperativo simple?

❑ ¿Y con nombres genéricos cómo quedaría?

```
data TF = W TG
type TG = [TH]
data TH = X String TB
        | Y TD TG TG
        | Z TD TG
```

❑ `fn := 1` se representa como `W [X "fn" (D 1)]`

❑ `a := N; fn := 1`
se representa como `W [X "a" (C "N")
 , X "fn" (D 1)]`

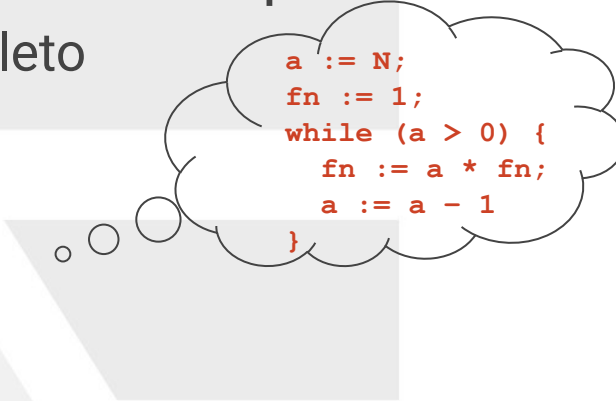
Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❏ ¿Qué forma tiene un lenguaje imperativo simple?

❏ Representación de un programa completo

```
Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , ...
```



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

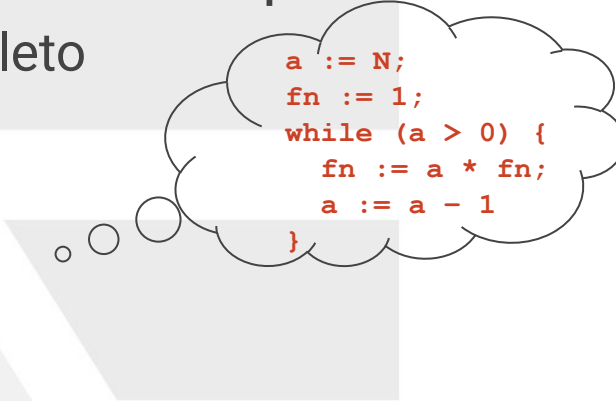
Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❏ ¿Qué forma tiene un lenguaje imperativo simple?

❏ Representación de un programa completo

```
Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , While (...)
        [ ...
        ]
      ]
```



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

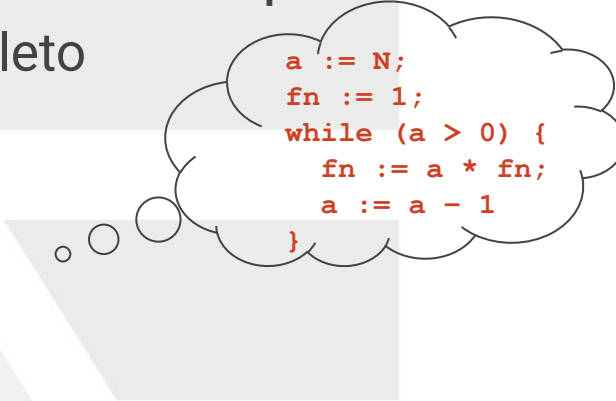
Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❏ ¿Qué forma tiene un lenguaje imperativo simple?

❏ Representación de un programa completo

```
Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , While (...)
        [ Assign "fn" (...)
          , Assign "a" (...)
        ]
      ]
```



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

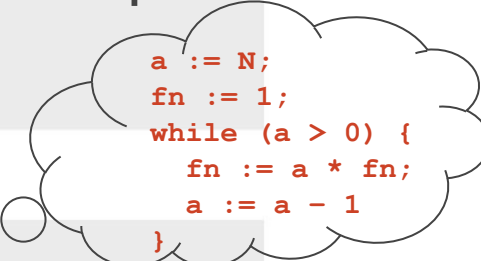
Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❏ ¿Qué forma tiene un lenguaje imperativo simple?

❏ Representación de un programa completo

```
Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , While (ROp Gt (Var "a") (NCte 0))
        [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
        , Assign "a" (NBOp Sub (Var "a") (NCte 1))
        ]
      ]
```



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

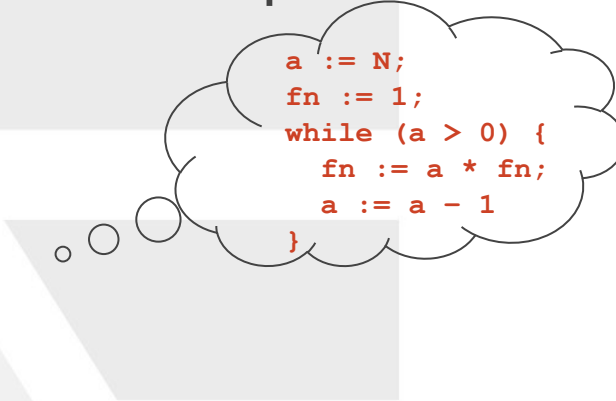
❑ ¿Qué forma tiene un lenguaje imperativo simple?

❑ Con nombres genéricos

```
W [ X "a" (C "N")
  , X "fn" (D (B A))
  , Z (O U (C "a") (D A))
    [ X "fn" (E H (C "a") (C "fn"))
    , X "a" (E G (C "a") (D (B A)))
    ]
  ]
```

❑ ¡Se observa que no hay significado!

❑ Solamente estructura...



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

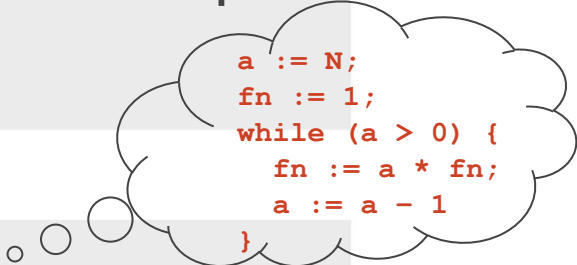

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Qué forma tiene un lenguaje imperativo simple?

❑ Con nombres genéricos á la Matrix

```
シ ラ ス チカチ スイ チカチネ
ヒ ス チカチ スウ スア フネネ
ヒ ツ スエ ム スイ チカチネ スウ フネネ
    ラ ス チカチ スエ ム スイ チカチネ スウ チカチネ
    ヒ ス チカチ スエ ム スイ チカチネ スウ スア フネネ
リ
リ
```



```
a := N;
fn := 1;
while (a > 0) {
    fn := a * fn;
    a := a - 1
}
```

❑ ¡Se observa que no hay significado!

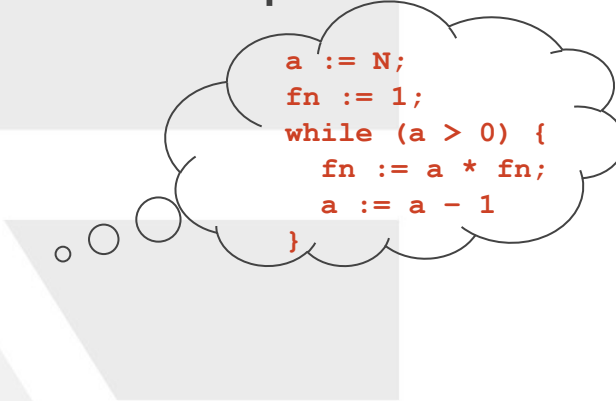
❑ Solamente estructura...

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Qué forma tiene un lenguaje imperativo simple?
- ❑ Con nombres genéricos á la Matrix

```
シ ラ ス ㊦ ㊦ ㊦ ㊦ ㊦
ヒ ス ㊦ ㊦ ㊦ ㊦ ㊦ ㊦
ヒ ソ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦
      ラ ス ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦
      ヒ ス ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦ ㊦
      リ
リ
```



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

- ❑ ¡Se observa que no hay significado!
- ❑ Solamente estructura...

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Qué forma tiene un lenguaje imperativo simple?
- ❑ Con nombres genéricos á la Matrix



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

- ❑ ¡Se observa que no hay significado!
- ❑ Solamente estructura...

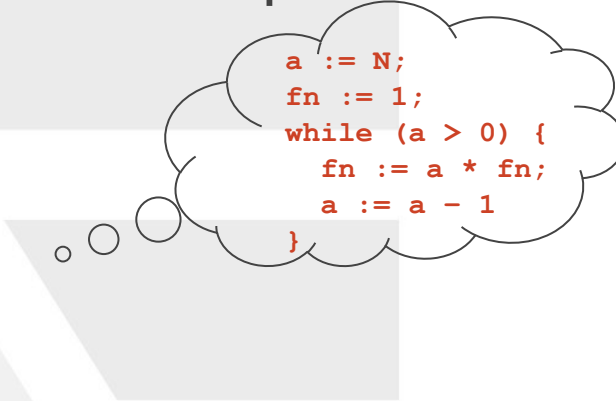
Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Qué forma tiene un lenguaje imperativo simple?

❑ Con nombres genéricos

```
シ ラ ス チカチ スイ チカチネ
ヒ ス チカチ スウ スア フネネ
ヒ ツ スエ ム スイ チカチネ スウ フネネ
    ラ ス チカチ スエ ム スイ チカチネ スウ チカチネ
    ヒ ス チカチ スエ ム スイ チカチネ スウ スア フネネ
リ
リ
```



```
a := N;
fn := 1;
while (a > 0) {
    fn := a * fn;
    a := a - 1
}
```

❑ ¡Se observa que no hay significado!

❑ Solamente estructura...

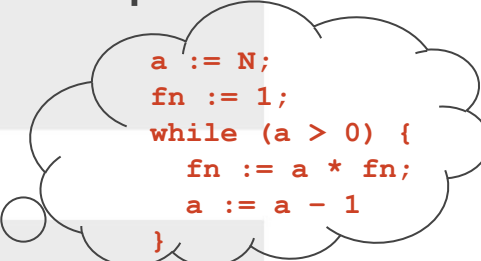
Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Qué forma tiene un lenguaje imperativo simple?

❑ Con nombres NO genéricos

```
Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , While (ROp Gt (Var "a") (NCte 0))
        [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
        , Assign "a" (NBOp Sub (Var "a") (NCte 1))
        ]
      ]
```



```
a := N;
fn := 1;
while (a > 0) {
  fn := a * fn;
  a := a - 1
}
```

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
 - ❑ ¿Cuál es el significado de un programa?

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
 - ❑ ¿Cuál es el significado de un programa?
 - ❑ Un programa es una secuencia de comandos...
 - ❑ ¿Cuál es el significado de un comando?

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
 - ❑ ¿Cuál es el significado de un programa?
 - ❑ Un programa es una secuencia de comandos...
 - ❑ ¿Cuál es el significado de un comando?
 - ❑ Pensar en la asignación: ¿qué intenta significar?

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
 - ❑ ¿Cuál es el significado de un programa?
 - ❑ Un programa es una secuencia de comandos...
 - ❑ ¿Cuál es el significado de un comando?
 - ❑ Pensar en la asignación: ¿qué intenta significar?
 - ❑ ¡Modificación de la memoria!
 - ❑ ¿Y una secuencia de modificaciones?

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
 - ❑ ¿Cuál es el significado de un programa?
 - ❑ Un programa es una secuencia de comandos...
 - ❑ ¿Cuál es el significado de un comando?
 - ❑ Pensar en la asignación: ¿qué intenta significar?
 - ❑ ¡Modificación de la memoria!
 - ❑ ¿Y una secuencia de modificaciones?
 - ❑ Es solamente una modificación más compleja

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma (Prog cs) = ... cs ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma (Prog cs) = evalBloque cs
evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

□ ¿Cómo dar significado a este lenguaje imperativo?

□ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma (Prog cs) = evalBloque cs

evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque []          = ...
evalBloque (c:cs)     = ... c ... evalBloque cs ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma (Prog cs) = evalBloque cs

evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque []          = \mem -> ...
evalBloque (c:cs) =
    \mem -> ... c ...
    ... evalBloque cs ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
```

```
evalPrograma (Prog cs) = evalBloque cs
```

```
evalBloque :: Bloque -> (Memoria -> Memoria)
```

```
evalBloque [] = \mem -> ...
```

```
evalBloque (c:cs) =
```

```
    \mem -> ... evalComando c ...
```

```
    ... evalBloque cs ...
```

```
evalComando :: Comando -> (Memoria -> Memoria)
```

```
evalComando ...
```


Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
```

```
evalPrograma (Prog cs) = evalBloque cs
```

```
evalBloque :: Bloque -> (Memoria -> Memoria)
```

```
evalBloque [] = \mem -> ...
```

```
evalBloque (c:cs) =
```

```
    \mem -> ... evalComando c mem
```

```
    ... evalBloque cs ...
```

```
evalComando :: Comando -> (Memoria -> Memoria)
```

```
evalComando ...
```

Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma (Prog cs) = evalBloque cs

evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque []          = \mem -> ...
evalBloque (c:cs) =
    \mem -> let mem' = evalComando c mem
              in evalBloque cs mem'

evalComando :: Comando -> (Memoria -> Memoria)
evalComando ...
```

Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalPrograma :: Program -> (Memoria -> Memoria)
evalPrograma (Prog cs) = evalBloque cs

evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque []          = \mem -> mem
evalBloque (c:cs) =
    \mem -> let mem' = evalComando c mem
              in evalBloque cs mem'

evalComando :: Comando -> (Memoria -> Memoria)
evalComando ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalBloque :: Bloque -> (Memoria -> Memoria)
evalBloque []      = \mem -> mem
evalBloque (c:cs) =
    \mem -> let mem' = evalComando c mem
            in evalBloque cs mem'
```

- ❑ El significado es una función
- ❑ ¡La secuencia de comandos ALTERA la memoria!

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando ...
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    ... x ... ne ...
evalComando (If be cs1 cs2) =
    ... be ...
    ... evalBloque cs1 ...
    ... evalBloque cs2 ...
evalComando (While be cs) = ??
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    \mem -> ... x ... ne ...
evalComando (If be cs1 cs2) =
    \mem -> ... evalBExp be ...
              ... evalBloque cs1 ...
              ... evalBloque cs2 ...
evalComando (While be cs) = ??
```

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    \mem -> ... x ... ne ...
evalComando (If be cs1 cs2) =
    \mem -> if (evalBExp be mem)
               then evalBloque cs1 mem
               else evalBloque cs2 mem
evalComando (While be cs) = ??
```


Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    \mem -> ... x ... evalNExp ne ...
evalComando (If be cs1 cs2) =
    \mem -> if (evalBExp be mem)
               then evalBloque cs1 mem
               else evalBloque cs2 mem
evalComando (While be cs) = ??
```

Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    \mem -> recordar x (evalNExp ne mem) mem
evalComando (If be cs1 cs2) =
    \mem -> if (evalBExp be mem)
               then evalBloque cs1 mem
               else evalBloque cs2 mem
evalComando (While be cs) = ??
```

Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
  \mem -> recordar x (evalNExp ne mem) mem
evalComando (If be cs1 cs2) =
  \mem -> if (evalBExp be mem)
             then evalBloque cs1 mem
             else evalBloque cs2 mem
evalComando (While be cs) = ??
```

¿Cuál es el problema?
¡Debe poder no terminar!

Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
  \mem -> recordar x (evalNExp ne mem) mem
evalComando (If be cs1 cs2) =
  \mem -> if (evalBExp be mem)
             then evalBloque cs1 mem
             else evalBloque cs2 mem
evalComando (While be cs) = ??
```

¡No se puede usar recursión estructural ni primitiva!

Lenguaje imperativo simple

- ¿Cómo dar significado a este lenguaje imperativo?
- ¿Cómo debe comportarse un **While**?

While be cs

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?

- ❑ ¿Cómo debe comportarse un **While**?

While be cs

- ❑ Evaluar **be**

- ❑ Si el resultado es falso, terminar

- ❑ Si no, evaluar **cs** y luego volver a repetir todo el ciclo

- ❑ O sea, volver a ejecutar **While be cs**

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?

- ❑ ¿Cómo debe comportarse un **While**?

While **be** **cs**

- ❑ Evaluar **be**

- ❑ Si el resultado es falso, terminar

- ❑ Si no, evaluar **cs** y luego volver a repetir todo el ciclo

- ❑ O sea, volver a ejecutar **While** **be** **cs**

- ❑ Es equivalente a un **If**

- ❑ donde una de las ramas vuelve a contener al **While**

Lenguaje imperativo simple

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    \mem -> recordar x (evalNExp ne mem) mem
evalComando (If be cs1 cs2) =
    \mem -> if (evalBExp be mem)
               then evalBloque cs1 mem
               else evalBloque cs2 mem
evalComando (While be cs) = ??
```

El significado es
como el de un **If**

Lenguaje imperativo simp

```
data Programa = Prog Bloque
type Bloque   = [Comando]
data Comando  = Assign Variable NExp
              | If BExp Bloque Bloque
              | While BExp Bloque
```

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Significado de un programa: modificación de la memoria

```
evalComando :: Comando -> (Memoria -> Memoria)
evalComando (Assign x ne) =
    \mem -> recordar x (evalNExp ne mem) mem
evalComando (If be cs1 cs2) =
    \mem -> if (evalBExp be mem)
               then evalBloque cs1 mem
               else evalBloque cs2 mem
evalComando (While be cs) =
    evalComando (If be (cs ++ [While be cs]) [])
```

El significado es
como el de un **If**

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
 - ❑ Observaciones
 - ❑ No puede hacerse en su totalidad con recursión estructural
 - ❑ La pregunta interesante no es por qué un **While** no termina
 - ❑ Sino que es ¿por qué termina un **While**?
 - ❑ ¡La evaluación de **be** depende de una memoria!
 - ❑ Y esa memoria puede ser modificada por **cs**
 - ❑ Si **cs** no modifica el estado, el **While** no termina

```
evalComando (While (BCte True) []) = \mem -> ⊥
```

Lenguaje imperativo simple

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Ejemplos

evalPrograma

```
(Prog [ Assign "a" (Var "N")
        , Assign "fn" (NCte 1)
        , While (ROp Gt (Var "a") (NCte 0))
          [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
            , Assign "a" (NBOp Sub (Var "a") (NCte 1))
          ] ])
(recordar "N" 5 enBlanco) = ??
```

Lenguaje imperativo simple

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Ejemplos

evalPrograma

```
(Prog [ Assign "a" (Var "N")
        , Assign "fn" (NCte 1)
        , While (ROp Gt (Var "a") (NCte 0))
              [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
                , Assign "a" (NBOp Sub (Var "a") (NCte 1))
              ])
(recordar "N" 5 enBlanco) = 120
```

Lenguaje imperativo simple

❑ ¿Cómo dar significado a este lenguaje imperativo?

❑ Ejemplos

evalPrograma

```
(Prog [ Assign "a" (Var "N")
        , Assign "fn" (NCte 1)
        , While (ROp Gt (Var "a") (NCte 0))
              [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
                , Assign "a" (NBOp Sub (Var "a") (NCte 1))
              ])
= \mem -> ??
```

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Ejemplos: demostrar esto es complejo

evalPrograma

```
(Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , While (ROp Gt (Var "a") (NCte 0))
        [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
        , Assign "a" (NBOp Sub (Var "a") (NCte 1)) ] ])
= \mem -> case (cuantoVale "N" mem) of
  Nothing -> error ("Variable N indefinida")
  Just n   -> if n >= 0 then recordar "fn" (fact n)
                (recordar "a" 0 mem)
                else ⊥
```

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Ejemplos: demostrar esto es complejo

evalPrograma

```
(Prog [ Assign "a" (Var "N")
      , Assign "fn" (NCte 1)
      , While (ROp Gt (Var "a") (NCte 1))
        [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
        , Assign "a" (NBOp Sub (Var "a") (NCte 1)) ] ] )
= \mem -> case (cuantoVale "N" mem) of
  Nothing -> error ("Variable N indefinida")
  Just n   -> if n >= 0 then recordar "fn" (fact n)
                                (recordar "a" 0 mem)
                                else ⊥
```

Esto es lo que se pretende
que el programa signifique...

Lenguaje imperativo simple

- ❑ ¿Cómo dar significado a este lenguaje imperativo?
- ❑ Ejemplos: demostrar esto es complejo

evalPrograma

```
(Prog [ Assign "a" (Var "N")
        , Assign "fn" (NCte 1)
        , While (ROp Gt (Var "a") (NCte 0))
          [ Assign "fn" (NBOp Mul (Var "a") (Var "fn"))
            , Assign "a" (NBOp Sub (Var "a") (NCte 1)) ] ])
= \mem -> case (cuantoVale "N" mem) of
    Nothing -> error ("Variable N indefinida")
    Just n   -> if n >= 0 then recordar "fn" (fact n)
                                     (recordar "a" 0 mem)
                                     else 1
```

...pero en imperativo no es posible prescindir de esto

Lenguaje imperativo simple

- ❑ Manipulación simbólica de **Programa**
 - ❑ ***Constant folding extendida a programas***
 - ❑ Lo que no depende de variables se puede resolver
- ```
optimize :: Programa -> Programa
optimize ...
```
- ❑ Se pueden utilizar las operaciones de constant folding definidas para **NExp** y **BExp**
  - ❑ No es necesario que se resuelvan comandos, pero podría ser interesante...

# Lenguaje imperativo simple

- ❑ Manipulación simbólica de **Programa** y su coherencia
  - ❑ *Constant folding extendida a programas*
  - ❑ Prop:  $\text{evalPrograma} \cdot \text{optimize} = \text{evalPrograma}$ ?
  - Dem: ??
- ❑ Dice que optimizar un programa no cambia su significado
- ❑ Van a precisarse varios lemas...
  - ❑ ¿Ya se habrán demostrado algunos?



# **Manipulación simbólica de lenguajes: Compilación**

# Compilación

❏ ¿Qué es ***compilar*** un programa?

# Compilación

- ❏ ¿Qué es **compilar** un programa?
  - ❏ Convertirlo en un programa de otro lenguaje de forma tal que ambos tengan el mismo significado

# Compilación

- ❏ ¿Qué es **compilar** un programa?
  - ❏ Convertirlo en un programa de otro lenguaje de forma tal que ambos tengan el mismo significado
  - ❏ ¡Es manipulación simbólica!
  - ❏ Es necesario contar primero con otro lenguaje...

# Compilación

- ❑ ¿Qué es **compilar** un programa?
  - ❑ Convertirlo en un programa de otro lenguaje de forma tal que ambos tengan el mismo significado
  - ❑ ¡Es manipulación simbólica!
  - ❑ Es necesario contar primero con otro lenguaje...
  - ❑ Para el lenguaje imperativo simple visto la compilación es complejo hacer esta manipulación
  - ❑ Se ejemplificará solamente para **ExpA**

# Compilación

- ❏ Lenguaje assembler muy elemental
  - ❏ Simplemente una lista de “mnemónicos”

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int | ADD | MUL
```



# Compilación

- ❑ Lenguaje assembler muy elemental

- ❑ Simplemente una lista de “mnemónicos”

```
type AssmProg = [AssmInstr]
```

```
data AssmInstr = PUSH Int | ADD | MUL
```

- ❑ Se pueden representar expresiones aritméticas usando una representación llamada “*polaca inversa*” o “posfija”

- ❑  $6 * (5 + 2)$  se representa como

```
[PUSH 2, PUSH 5, ADD, PUSH 6, MUL]
```

- ❑ Es la inversa de la notación  $((*) 6 ((+) 5 2))$ , que es llamada “notación prefija” o “notación polaca”

# Compilación

## ❑ Significado para este lenguaje assembler

### ❑ Se utilizará un tipo abstracto de datos para pilas

```
data Stack a -- Tipo abstracto
emptyS :: Stack a -- Una stack vacío
push :: Int -> Stack -> Stack -- Apila un valor
topN :: Int -> Stack -> [Int] -- Los 1ros n valores
popN :: Int -> Stack -> Stack -- Desapila los 1eros n
 -- Ambas si hay menos de n, usan las que hay
finalTop :: Stack -> Int -- Desapila el último
 -- o falla si no hay exactamente un valor
```

# Compilación

- Significado para este lenguaje assembler
  - Algunas propiedades que deben cumplir las pilas
    - para todo  $n$ .  $\text{topN } n \text{ emptyS} = []$
    - para todo  $s$ .  $\text{topN } 0 \ s = []$
    - para todo  $v$ . para todo  $n > 0$ . para todo  $s$ .  
 $\text{topN } n (\text{push } v \ s) = v : \text{topN } (n-1) \ s$
    - para todo  $v$ .  $\text{finalTop } (\text{push } v \ \text{emptyS}) = v$
    - para todo  $s$ . si para todo  $v$ .  $s \neq \text{push } v \ \text{emptyS}$   
entonces  $\text{finalTop } s = \perp$
    - para todo  $s$ .  $\text{popN } 0 \ s = s$
    - para todo  $v$ . para todo  $n > 0$ . para todo  $s$ .  
 $\text{popN } n (\text{push } v \ s) = \text{popN } (n-1) \ s$

# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

- Significado para este lenguaje assembler
  - Se necesita una pila para almacenar datos intermedios

```
execAssm :: AssmProg -> Int
execAssm is = exec is emptyS

where exec :: AssmProg -> (Stack Int -> Int)
 exec [] s = finalTop s
 exec (i:is) s = let s' = execInstr i s
 in exec is s'
```

- Hay varias formas en las que podría fallar
  - Se requieren reglas de buena formación de programas...

# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

- Significado para este lenguaje assembler
  - El significado de una instrucción es una transformación entre pilas

```
execInstr :: AssmInstr -> (Stack Int -> Stack Int)
execInstr ...
```

# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

- Significado para este lenguaje assembler
  - El significado de una instrucción es una transformación entre pilas (observar el uso de una “ALU”)

```
execInstr :: AssmInstr -> (Stack Int -> Stack Int)
execInstr (PUSH n) s = push n s
execInstr ADD s = useALU (+) s
execInstr MUL s = useALU (*) s

useALU :: (Int->Int->Int) -> (Stack Int -> Stack Int)
useALU op s = let [n,m] = topN 2 s
 in push (op n m) (popN 2 s)
```

# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

## Significado para este lenguaje assembler

### Ejemplo

```
execAssm [PUSH 2,PUSH 5,ADD,PUSH 6,MUL]
→ exec [PUSH 2,PUSH 5,ADD,PUSH 6,MUL] emptyS
→ exec [PUSH 5,ADD,PUSH 6,MUL] (execInstr (PUSH 2) emptyS)
→ exec [PUSH 5,ADD,PUSH 6,MUL] (push 2 emptyS)
→ exec [ADD,PUSH 6,MUL] (push 5 (push 2 emptyS))
→ exec [PUSH 6,MUL] ((useALU (+) (push 5 (push 2 emptyS))))
→ exec [PUSH 6,MUL] (push (5+2) emptyS)
→ exec [MUL] (push 6 (push (5+2) emptyS))
→ exec [] ((useALU (*) (push 6 (push (5+2) emptyS))))
→ finalTop (push (6*(5+2)) emptyS)
→ 6*(5+2)
```

# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

- Ahora sí, la compilación de **ExpA**
  - Manipulación simbólica (por recursión estructural)

```
compileExpA :: ExpA -> AssmProg
compileExpA ...
```



# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

□ Ahora sí, la compilación de **ExpA**

□ Manipulación simbólica (por recursión estructural)

```
compileExpA :: ExpA -> AssmProg
compileExpA (Cte n) = ...
compileExpA (Suma e1 e2) =
 ... compileExpA e2 ... compileExpA e1 ...
compileExpA (Mult e1 e2) =
 ... compileExpA e2 ... compileExpA e1 ...
```

# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

□ Ahora sí, la compilación de **ExpA**

□ Manipulación simbólica (por recursión estructural)

```
compileExpA :: ExpA -> AssmProg
compileExpA (Cte n) = [PUSH n]
compileExpA (Suma e1 e2) =
 compileExpA e2 ++ compileExpA e1 ++ [ADD]
compileExpA (Mult e1 e2) =
 compileExpA e2 ++ compileExpA e1 ++ [MUL]
```

□ Observar que

- en ningún momento se usa el significado
- la “inversión” de apilamiento de los hijos (e2 al último)

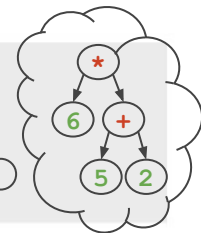
# Compilación

```
type AssmProg = [AssmInstr]
data AssmInstr = PUSH Int
 | ADD | MUL
```

## Compilación de expresiones aritméticas

### Ejemplo

```
→ compileExpA (Mult (Cte 6) (Suma (Cte 5) (Cte 2)))
→ compileExpA (Suma (Cte 5) (Cte 2)) ++ compileExpA (Cte 6) ++ [MUL]
→ (compileExpA (Cte 2) ++ compileExpA (Cte 5) ++ [ADD]) ++ [PUSH 6] ++ [MUL]
→ ([PUSH 2] ++ [PUSH 5] ++ [ADD]) ++ [PUSH 6, MUL]
→ [PUSH 2, PUSH 5, ADD] ++ [PUSH 6, MUL]
→ [PUSH 2, PUSH 5, ADD, PUSH 6, MUL] . . .
```



- Observar que el resultado es el inverso de un recorrido preorden del árbol (por eso, “polaca inversa”)

# Compilación

## ❏ *Corrección* de la compilación

❏ ¿Es la coherencia entre manipulación y significado!

❏ Prop:  $\text{execAssm} \circ \text{compileExpA} = \text{evalExpA}$ ?

Dem: Por ppio de ext. y (.) y luego ppio. de ind. estructural

Caso base)  $\text{execAssm} (\text{compileExpA} (\text{Cte } n)) = \text{evalExpA} (\text{Cte } n)$ ?

Caso ind.1)  $\text{HI1)} \text{execAssm} (\text{compileExpA } e_1) = \text{evalExpA } e_1!$

$\text{HI2)} \text{execAssm} (\text{compileExpA } e_2) = \text{evalExpA } e_2!$

$\text{TI)} \text{execAssm} (\text{compileExpA} (\text{Suma } e_1 e_2))$   
 $= \text{evalExpA} (\text{Suma } e_1 e_2)$ ?

Caso ind.2)  $\text{HI1)} \text{execAssm} (\text{compileExpA } e_1) = \text{evalExpA } e_1!$

$\text{HI2)} \text{execAssm} (\text{compileExpA } e_2) = \text{evalExpA } e_2!$

$\text{TI)} \text{execAssm} (\text{compileExpA} (\text{Mult } e_1 e_2))$   
 $= \text{evalExpA} (\text{Mult } e_1 e_2)$ ?

# Compilación

## ❏ *Corrección* de la compilación

❏ ¡Es la coherencia entre manipulación y significado!

❏ **Prop:**  $\text{execAssm} \circ \text{compileExpA} = \text{evalExpA}?$

**Dem:** Por ppio de ext. y (.) y luego ppio. de ind. estructural

❏ Para demostrar algunos casos es necesario un Lema:

¿para todo  $e$ . para todo  $is$ . para todo  $s$ .

$\text{execAssm} (\text{compileExpA } e ++ is) s$

$= \text{exec } is (\text{push} (\text{execAssm} (\text{compileExpA } e)) s) ?$

❏ La demostración de este Lema es más compleja que lo visto

❏ Demo completa, en este [link](#)



# Recursión (explícita)

- Fin de temporada
  - Lo visto es todo lo que vamos a decir sobre recursión estructural (explícita)



# Resumen



# Resumen

- ❑ Utilización de las técnicas vistas
  - ❑ Asignación de significado por recursión estructural
  - ❑ Manipulación simbólica por recursión estructural
  - ❑ Demostración de coherencia por inducción estructural
- ❑ Límites a la recursión estructural
- ❑ Aplicación a un lenguaje imperativo simple
  - ❑ Significado como función de transformación de memorias
  - ❑ Compilación como manipulación simbólica que preserva significado



la!

# Post títulos...

- ❑ Fin de temporada
  - ❑ Lo visto es todo lo que vamos a decir sobre recursión estructural (explícita)
  - ❑ Pero la buena noticia es...
    - ❑ ...¡que ya se firmaron los contratos para la nueva temporada!
    - ❑ No se pierdan la temporada 3 (empieza en 15 días)