

# Paradigmas de Programación

## Interpretación

**2do cuatrimestre de 2024**

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

# Intérpretes

¿Qué es un intérprete?

Un intérprete es un **programa** que ejecuta **programas**.

Involucra dos lenguajes de programación:

**Lenguaje de implementación**

Lenguaje en el que está definido el intérprete.

**Lenguaje fuente**

Lenguaje en el que están escritos los programas que se interpretan.

Pregunta frecuente

¿Podría ser que estos dos lenguajes coincidan?

Sí, puede haber intérpretes capaces de interpretarse a sí mismos.

No es algo que tenga especial interés en la materia.

# Lenguajes que usaremos

En la clase de hoy vamos a definir varios intérpretes.

## Lenguaje de implementación

Vamos a definir intérpretes en Haskell.

## Lenguaje fuente

Vamos a definir intérpretes para distintos lenguajes fuente (p. ej. lenguajes imperativos y funcionales).

## Nota

Vamos a trabajar con lenguajes fuente minimalistas (“de juguete”).  
Nos alcanzan para ilustrar los conceptos importantes.

## Sintaxis concreta vs. sintaxis abstracta

El intérprete recibe como entrada un dato que representa un programa escrito en el lenguaje fuente.

¿Cómo se representa un programa?

### Sintaxis concreta

Se puede representar un programa como una **cadena de texto**.  
Por ejemplo:

```
"while (true) { x = x + 1; }" :: String
```

### Sintaxis abstracta

Se puede representar un programa como un **árbol de sintaxis**.  
Por ejemplo:

```
EWhile  
  ETrue  
    (EAssign "x" (EAdd (EVar "x") (EConstNum 1)))  
  :: Programa
```

# Sintaxis concreta vs. sintaxis abstracta

## En la clase de hoy

Representaremos a los programas como árboles de sintaxis abstracta.

Convertir la sintaxis concreta (texto) en sintaxis abstracta (árbol) es un problema de **análisis sintáctico**.

Queda fuera del alcance de esta materia.

# Lenguaje de expresiones aritméticas

Consideremos un lenguaje de expresiones aritméticas construidas inductivamente de la siguiente manera:

1. Una constante entera es una expresión.
2. La suma de dos expresiones es una expresión.

Las expresiones se pueden representar con un tipo de datos:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...  
          | EAdd Expr Expr         -- e1 + e2
```

Por ejemplo, "1 + (2 + 3)" se representa con:

```
EAdd (EConstNum 1)  
    (EAdd (EConstNum 2) (EConstNum 3))
```

## Ejercicio (1)

Definir un intérprete:

```
eval :: Expr -> Int
```

## Extensión con constantes booleanas

¿Podríamos extender el lenguaje con constantes booleanas?

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EConstBool Bool        -- True, False
          | EAdd Expr Expr          -- e1 + e2
```

### Problema

El intérprete ya no es una función `eval :: Expr -> Int`.

### Valores

Definimos un tipo de datos `Val` para representar los *valores* (posibles resultados de los cálculos):

```
data Val = VN Int
         | VB Bool
```

### Ejercicio (2)

Definir un intérprete:

```
eval :: Expr -> Val
```

# Definiciones locales y entornos

Queremos extender el lenguaje con definiciones locales:

```
let x = 3 in (let y = x + x in 1 + y)
```

Necesitamos mantener registro del valor que tiene cada variable.

## Entornos

Un **entorno** es un diccionario que asocia identificadores a valores.

Vamos a suponer que contamos con tipos:

Id            identificadores (nombres de variables)

(Env a)    entornos que asocian identificadores a valores de tipo a

y la siguiente interfaz:

```
emptyEnv  :: Env a
```

```
lookupEnv :: Env a -> Id -> a
```

```
extendEnv :: Env a -> Id -> a -> Env a
```



# Extensión con variables y definiciones locales

Extendemos el lenguaje de expresiones:

```
data Expr = EConstNum Int           -- 1, 2, 3, ...
          | EConstBool Bool         -- True, False
          | EAdd Expr Expr           -- e1 + e2
          | EVar Id                  -- x
          | ELet Id Expr Expr        -- let x = e1 in e2
```

## Problema

¿Cuál es el resultado de evaluar (EVar "x")?

El intérprete ya no es una función `eval :: Expr -> Val`.

## Ejercicio (3)

Definir un intérprete:

```
eval :: Expr -> Env Val -> Val
```

# Extensión con variables y definiciones locales

## Comentario

En el lenguaje con declaraciones locales, una expresión no denota un valor, sino una *función* que devuelve un valor en función del entorno dado:

$$\text{eval} :: \text{Expr} \rightarrow (\text{Env Val} \rightarrow \text{Val})$$

# Intérprete con memoria para un lenguaje imperativo

Queremos extender el lenguaje con características imperativas:

1. Asignaciones:  $x := e$
2. Composición secuencial:  $e1; e2$

Vamos a suponer que:

1. Existen distintas convenciones para el *valor* de una asignación:
  - ▶ En C, es el valor que se asigna,
  - ▶ En otros lenguajes es de tipo void o Unit.
2. La semántica de la composición  $e1; e2$  corresponde a evaluar primero  $e1$ , descartando su valor, y a continuación evaluar  $e2$ .

Por ejemplo, el siguiente programa debería resultar en el entero 4:

```
let x = 1 in x := x + 1; x + x
```

# Intérprete con memoria para un lenguaje imperativo

## Variables inmutables

En un lenguaje puramente funcional, las variables son *inmutables*.

- ▶ El entorno asocia cada variable directamente a un *valor*.

## Variables mutables

En un lenguaje imperativo, las variables son típicamente *mutables*.

- ▶ El entorno **no** asocia cada variable a un valor.
- ▶ El entorno asocia cada variable a una **dirección de memoria**.
- ▶ Además, hay una **memoria** que asocia direcciones a valores.
- ▶ La evaluación de un programa puede modificar la memoria.

# Intérprete con memoria para un lenguaje imperativo

## Memorias

Una **memoria** es un diccionario que asocia direcciones a valores.

Vamos a suponer que contamos con tipos:

`Addr`            direcciones de memoria

`(Mem a)`        memorias que asocian direcciones a valores de tipo `a`

y la siguiente interfaz:

```
emptyMem        :: Mem a
```

```
freeAddress :: Mem a -> Addr
```

```
load            :: Mem a -> Addr -> a
```

```
store           :: Mem a -> Addr -> a -> Mem a
```

## Extensión con asignación y composición secuencial

Extendemos el lenguaje con características imperativas:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EConstBool Bool        -- True, False
          | EAdd Expr Expr          -- e1 + e2
          | EVar Id                 -- x
          | ELet Id Expr Expr       -- let x = e1 in e2
          | ESeq Expr Expr          -- e1; e2
          | EAssign Id Expr         -- x := e
```

### Problema

El resultado de evaluar una asignación ( $x := e$ ) no puede ser sólo un valor (considerar p. ej. `let x = 1 in x := 2; x`).

**¿Cuál debería ser el tipo del intérprete?**

### Ejercicio (4)

Definir un intérprete:

```
eval :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
```

## Extensión con estructuras de control

Extendamos ahora el intérprete con estructuras de control:

```
data Expr =  
    EConstNum Int          -- 1, 2, 3, ...  
  | EConstBool Bool       -- True, False  
  | EAdd Expr Expr        -- e1 + e2  
  | EVar Id               -- x  
  | ELet Id Expr Expr     -- let x = e1 in e2  
  | ESeq Expr Expr        -- e1; e2  
  | EAssign Id Expr       -- x := e  
  | ELtNum Expr Expr      -- e1 < e2  
  | EIf Expr Expr Expr    -- if e1 then e2 else e3  
  | EWhile Expr Expr      -- while e1 do e2
```

### Ejercicio (5)

Definir un intérprete:

```
eval :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
```

# Intérpretes para lenguajes funcionales

Casi todos los lenguajes funcionales están basados en el **cálculo- $\lambda$** .

El cálculo- $\lambda$  es un lenguaje que tiene sólo tres construcciones:

```
data Expr = EVar Id           --  $x$ 
          | ELam Id Expr      --  $\lambda x. e$ 
          | EApp Expr Expr    --  $e1\ e2$ 
```

Es posible programar usando sólo estas construcciones.

Pero vamos a extender el cálculo- $\lambda$  para que sea más cómodo y se asemeje a un lenguaje realista.



# Intérpretes para lenguajes funcionales

¿Cuál será el resultado de evaluar  $(\lambda x \rightarrow x + x)$ ?

Necesitamos extender el tipo de los valores para incluir funciones.

## Primer intento

El valor de una función es su “código fuente”.

```
data Val = VN Int
         | VB Bool
         | VFunction Id Expr
```

Por ejemplo, el resultado de evaluar  $(\lambda x \rightarrow x + x)$  sería:

```
VFunction "x" (EAdd (EVar "x") (EVar "x"))
```

Veremos en breve que esto no es correcto.

# Intérprete funcional (primer intento)

Considerar el cálculo- $\lambda$  extendido con enteros y booleanos:

```
data Expr =  
    EVar Id           --  $x$   
  | ELam Id Expr      --  $\lambda x \rightarrow e$   
  | EApp Expr Expr    --  $e1\ e2$   
  | EConstNum Int     --  $1, 2, 3, \dots$   
  | EConstBool Bool   --  $True, False$   
  | EAdd Expr Expr     --  $e1 + e2$   
  | ELet Id Expr Expr --  $let\ x = e1\ in\ e2$   
  | EIf Expr Expr Expr --  $if\ e1\ then\ e2\ else\ e3$ 
```

## Ejercicio (6)

Definir un intérprete:

```
eval :: Expr -> Env Val -> Val
```

# Intérprete funcional (primer intento)

## Ejercicio

Evaluar el siguiente programa con el intérprete recién definido:

```
let suma = \ x -> \ y -> x + y in
let f = suma 5 in
let x = 0 in
  f 3
```

## Problema: captura de variables

Queríamos que el resultado sea 8 pero es 3.

El problema es que la variable `f` queda ligada al valor:

```
VFunction "y" (EAdd (EVar "x") (EVar "y"))
```

La variable `x` está **libre**.

# Intérprete funcional (segundo intento: con clausuras)

## Segundo intento

El valor de una función es una **clausura**.

Una clausura es un valor que incluye:

1. El código fuente de la función.
2. Un entorno que le da valor a todas sus variables libres.

```
data Val = VN Int
        | VB Bool
        | VClosure Id Expr (Env Val)
```

# Intérprete funcional (segundo intento: con clausuras)

Recordemos las expresiones del cálculo- $\lambda$  con enteros y booleanos:

```
data Expr =  
    EVar Id           --  $x$   
  | ELam Id Expr      --  $\lambda x \rightarrow e$   
  | EApp Expr Expr    --  $e1\ e2$   
  | EConstNum Int     --  $1, 2, 3, \dots$   
  | EConstBool Bool   --  $True, False$   
  | EAdd Expr Expr     --  $e1 + e2$   
  | ELet Id Expr Expr  --  $let\ x = e1\ in\ e2$   
  | EIf Expr Expr Expr --  $if\ e1\ then\ e2\ else\ e3$ 
```

## Ejercicio (7)

Definir un intérprete usando clausuras:

```
eval :: Expr -> Env Val -> Val
```

# Estrategias de evaluación

Hay distintas técnicas para evaluar una aplicación ( $e_1$   $e_2$ ).  
Estas técnicas se conocen como **estrategias de evaluación**.

El intérprete que hicimos recién usa la siguiente estrategia:

1. **Llamada por valor** (*call-by-value*):

Se evalúa  $e_1$  hasta que sea una clausura.

Se evalúa  $e_2$  hasta que sea un valor.

Se procede con la evaluación del cuerpo de la función.

El parámetro queda ligado al **valor** de  $e_2$ .

Hay otras estrategias de evaluación; por ejemplo:

2. **Llamada por nombre** (*call-by-name*):

Se evalúa  $e_1$  hasta que sea una clausura.

Se procede **directamente** a evaluar el cuerpo de la función.

El parámetro queda ligado a la expresión  $e_2$  **sin evaluar**.

Cada vez que se usa el parámetro, se evalúa la expresión  $e_2$ .

3. **Llamada por necesidad** (*call-by-need*):

(La veremos en un rato).

## Interprete call-by-name

En la estrategia de evaluación **call-by-name**:

- ▶ Al evaluar `(let x = e1 in e2)` se evalúa directamente `e2`. La variable `x` queda ligada a una copia no evaluada de `e1`.
- ▶ Los entornos **no** asocian identificadores a valores. Los entornos asocian identificadores a **thunks**.

### Thunks

Un *thunk* es un dato que incluye:

1. Una expresión no evaluada.
2. Un entorno que le da valor a todas sus variables libres.

Los *thunks* y valores se definen del siguiente modo:

```
data Thunk = TT Expr (Env Thunk)
data Val   = VN Int
           | VB Bool
           | VClosure Id Expr (Env Thunk)
```

# Intérprete call-by-name

Recordemos las expresiones del cálculo- $\lambda$  con enteros y booleanos:

```
data Expr =  
    EVar Id           --  $x$   
  | ELam Id Expr      --  $\lambda x \rightarrow e$   
  | EApp Expr Expr     --  $e1\ e2$   
  | EConstNum Int      --  $1, 2, 3, \dots$   
  | EConstBool Bool    --  $True, False$   
  | EAdd Expr Expr     --  $e1 + e2$   
  | ELet Id Expr Expr  --  $let\ x = e1\ in\ e2$   
  | EIf Expr Expr Expr --  $if\ e1\ then\ e2\ else\ e3$ 
```

## Ejercicio (8)

Definir un intérprete que use la estrategia **call-by-name**:

```
eval :: Expr -> Env Thunk -> Val
```



# Interprete call-by-need

**Llamada por necesidad** (`call-by-need`).

Para evaluar una aplicación (`e1 e2`):

- Se evalúa `e1` hasta que sea una clausura.

- Se procede **directamente** a evaluar el cuerpo de la función.

- El parámetro queda ligado a la expresión `e2` **sin evaluar**.

- La primera vez que el parámetro se necesita, se evalúa `e2`.

- Se guarda el resultado para evitar evaluar `e2` nuevamente.**

Para esto se necesita contar con una **memoria mutable**.

# Interprete call-by-need

## Valores

En call-by-need hay dos tipos de valores:

1. Valores finales (enteros, booleanos, clausuras, etc.).
2. Valores pendientes de ser evaluados (*thunks*).

```
data Val = VN Int
         | VB Bool
         | VClosure Id Expr (Env Addr)
         | VThunk Expr (Env Addr)
```

## Propiedades de la evaluación call-by-need

1. El entorno asocia identificadores a direcciones.
2. La memoria asocia direcciones a valores finales o thunks.
3. El resultado de evaluar una expresión es un valor final.

# Intérprete call-by-need

Recordemos las expresiones del cálculo- $\lambda$  con enteros y booleanos:

```
data Expr =  
    EVar Id           --  $x$   
  | ELam Id Expr      --  $\lambda x \rightarrow e$   
  | EApp Expr Expr    --  $e1\ e2$   
  | EConstNum Int     --  $1, 2, 3, \dots$   
  | EConstBool Bool  --  $True, False$   
  | EAdd Expr Expr    --  $e1 + e2$   
  | ELet Id Expr Expr --  $let\ x = e1\ in\ e2$   
  | EIf Expr Expr Expr --  $if\ e1\ then\ e2\ else\ e3$ 
```

## Ejercicio (9)

Definir un intérprete que use la estrategia **call-by-need**:

```
eval :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
```

# Propagación y manejo de errores

Extendemos el lenguaje de expresiones aritméticas:

```
data Expr = EConstNum Int          -- 1, 2, 3, ...
          | EAdd Expr Expr         -- e1 + e2
          | EVar Id                -- x
          | ELet Id Expr Expr      -- let x = e1 in e2
          | EDiv Expr Expr         -- e1 / e2
          | ETry Expr Expr         -- try e1 else e2
```

- ▶ El operador de división falla si el divisor es 0.
- ▶ La estructura de control (try e1 else e2) evalúa e1 y, en caso de falla, procede a evaluar e2.

**¿Cómo se modifica el tipo del intérprete?**

## Ejercicio (10)

Definir un intérprete:

```
eval :: Expr -> Env Int -> Maybe Int
```

# No determinismo

Extendemos el lenguaje de expresiones aritméticas:

```
data Expr = EConstNum Int           -- 1, 2, 3, ...
          | EAdd Expr Expr          -- e1 + e2
          | EVar Id                  -- x
          | ELet Id Expr Expr        -- let x = e1 in e2
          | EAmb Expr Expr           -- amb e1 e2
```

- El operador (amb e1 e2) elige entre e1 y e2 de manera no determinística.

**¿Cómo se modifica el tipo del intérprete?**

Ejercicio (11)

Definir un intérprete:

```
eval :: Expr -> Env Int -> [Int]
```