Paradigmas de Programación

Esquemas de recursión
Tipos de datos inductivos

2do cuatrimestre de 2025

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Breve repaso

Esquemas de recursión sobre listas

Tipos de datos algebraicos

Esquemas de recursión sobre otras estructuras

Las funciones `map` y `filter`

La clase pasada vimos las siguientes funciones:

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x : xs) = f x : map f xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs
```

¿Qué tipo tiene la expresión `map filter`?
¿Cómo la podríamos usar?

Funciones anónimas

Notación "lambda"

Una expresión de la forma:

```
\ x -> e
```

representa una función que recibe un parámetro `x` y devuelve `e`.

`(\ x1 x2 ... xn -> e)` $\equiv$ `(\ x1 -> (\ x2 -> ... (\ xn -> e)))`

Ejemplo

```
>> map (\ x -> (x, x)) [1, 2, 3]
⇝ [(1, 1), (2, 2), (3, 3)]
```

Funciones de orden superior

¿Qué relación hay entre las siguientes funciones?

```
suma :: Int -> Int -> Int
suma x y = x + y

suma' :: (Int, Int) -> Int
suma' (x, y) = x + y
```

Están relacionadas del siguiente modo:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (x, y) = f x y
```

Dentro de algunas clases, veremos que se puede demostrar:

```
suma = curry suma'      suma' = uncurry suma
```

Breve repaso

Esquemas de recursión sobre listas

Tipos de datos algebraicos

Esquemas de recursión sobre otras estructuras

Esquemas sobre listas

Pensemos algunas funciones sobre listas

- `sumaL` : la suma de todos los valores de una lista de enteros
- `concatena` : la concatenación de todos los elementos de una lista de listas
- `reverso` : el reverso de una lista

Recursión estructural

Sea `g :: [a] -> b` definida por dos ecuaciones:

$$
\begin{array}{lcl}
\texttt{g []} & = & \langle \textit{caso base} \rangle \\
\texttt{g (x : xs)} & = & \langle \textit{caso recursivo} \rangle
\end{array}
$$

g está dada por **recursión estructural** si:

1. El caso base devuelve un valor `z` "fijo" (no depende de g).
2. El caso recursivo **no** puede usar las variables g ni `xs`, salvo en la expresión `(g xs)`.

$$
\begin{array}{lcl}
\texttt{g []} & = & \texttt{z} \\
\texttt{g (x : xs)} & = & \ldots \texttt{x} \ldots \texttt{(g xs)} \ldots
\end{array}
$$

Recursión estructural

Ejemplos de recursión estructural

```haskell
suma :: [Int] -> Int
suma []       = 0
suma (x : xs) = x + suma xs

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

-- Insertion sort
isort :: Ord a => [a] -> [a]
isort []       = []
isort (x : xs) = insertarOrdenado x (isort xs)
```

Recursión estructural

Ejemplo:

```
-- Selection sort
ssort :: Ord a => [a] -> [a]
ssort []       = []
ssort (x : xs) = minimo (x : xs)
               : ssort (sacarMinimo (x : xs))
```

¿Es recursión estructural?
No.

Plegado de listas a derecha

La función `foldr` abstrae el esquema de recursión estructural:

foldr

```
foldr f z []       = z
foldr f z (x : xs) = f x (foldr f z xs)
```

¿Cuál es su tipo?

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Toda recursión estructural es una instancia de `foldr`.

Plegado de listas a derecha

Ejemplo — suma con foldr

```
suma :: [Int] -> Int
suma = foldr (+) 0
```

$$
\begin{array}{lll}
\texttt{suma [1, 2]} & \leadsto & \texttt{foldr (+) 0 [1, 2]} \\
 & \leadsto & \texttt{(+) 1 (foldr (+) 0 [2])} \\
 & \leadsto & \texttt{(+) 1 ((+) 2 (foldr (+) 0 []))} \\
 & \leadsto & \texttt{(+) 1 ((+) 2 0)} \\
 & \leadsto^{*} & \texttt{3}
\end{array}
$$

Análogamente:

```
producto :: [Int] -> Int
producto = foldr (*) 1

and, or :: [Bool] -> Bool
and = foldr (&&) True
or  = foldr (||) False
```

Plegado de listas a derecha

Ejemplo — reverso con foldr

```
reverso :: [a] -> [a]
reverso []       = []
reverso (x : xs) = reverso xs ++ [x]
```

Es recursiva estructural. ¿Cómo la escribiríamos usando `foldr`?

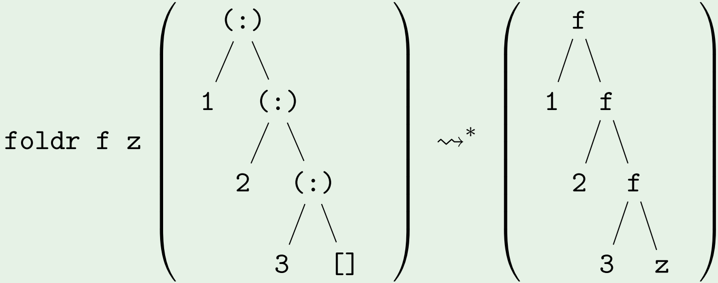
```
reverso = foldr (\ x rec -> rec ++ [x]) []
```

Otras formas equivalentes:

```
reverso = foldr (\ x rec -> flip (++) [x] rec) []
reverso = foldr (\ x -> flip (++) [x]) []
reverso = foldr (\ x -> flip (++) ((: []) x)) []
reverso = foldr (\ x -> (flip (++) . (: [])) x) []
reverso = foldr (flip (++) . (: [])) []
```

Plegado de listas a derecha

Ilustración gráfica del plegado a derecha



En particular, se puede demostrar que:

```
        foldr (:) []  =  id
   foldr ((:) . f) []  =  map f
foldr (const (+ 1)) 0  =  length
```

Recursión primitiva

Sea `g :: [a] -> b` definida por dos ecuaciones:

$$\begin{array}{lcl} \texttt{g []} & = & \langle \textit{caso base} \rangle \\ \texttt{g (x : xs)} & = & \langle \textit{caso recursivo} \rangle \end{array}$$

Decimos que la definición de g está dada por **recursión primitiva** si:

1. El caso base devuelve un valor `z` "fijo" (no depende de g).
2. El caso recursivo **no** puede usar la variable g, salvo en la expresión `(g xs)`.
   (Sí puede usar la variable `xs`).

$$\begin{array}{lcl} \texttt{g []} & = & \texttt{z} \\ \texttt{g (x : xs)} & = & \ldots \texttt{x} \ldots \texttt{xs} \ldots \texttt{(g xs)} \ldots \end{array}$$

Similar a la recursión estructural, pero permite referirse a `xs`.

Recursión primitiva

Observación

- Todas las definiciones dadas por recursión estructural también están dadas por recursión primitiva.
- Hay definiciones dadas por recursión primitiva que no están dadas por recursión estructural.

Ejemplo

Dado un texto, borrar todos los espacios iniciales.

```
trim :: String -> String
>> trim "   Hola PLP" ⇝ "Hola PLP"

trim []       = []
trim (x : xs) = if x == ' ' then trim xs else x : xs
```

Intenten escribirla con `foldr`.
¿Están haciendo recursión estructural?

Recursión primitiva

Escribamos una función `recr` para abstraer el esquema de recursión primitiva:

```
recr f z []       = z
recr f z (x : xs) = f x xs (recr f z xs)
```

¿Cuál es su tipo?

```
recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
```

Toda recursión primitiva es una instancia de `recr`.

Escribamos `trim` ahora usando `recr`:

```
trim = recr (\ x xs rec -> if x == ' '
                            then rec
                            else x : xs)
            []
```

Recursión iterativa

Sea `g :: b -> [a] -> b` definida por dos ecuaciones:

```
g ac []        = ⟨caso base⟩
g ac (x : xs)  = ⟨caso recursivo⟩
```

Recursión iterativa

Decimos que la definición de g está dada por *recursión iterativa* si:

1. El caso base devuelve el acumulador `ac`.
2. El caso recursivo invoca inmediatamente a `(g ac' xs)`, donde `ac'` es el acumulador actualizado en función de su valor anterior y el valor de `x`.

Recursión iterativa

Ejemplos de recursión iterativa

```
-- Reverso con acumulador.
reverso' :: [a] -> [a] -> [a]
reverso' ac []       = ac
reverso' ac (x : xs) = reverso' (x : ac) xs

-- Pasaje de binario a decimal con acumulador.
-- Precondición: recibe una lista de 0s y 1s.
bin2dec' :: Int -> [Int] -> Int
bin2dec' ac []       = ac
bin2dec' ac (b : bs) = bin2dec' (b + 2 * ac) bs

-- Insertion sort con acumulador.
isort' :: Ord a => [a] -> [a] -> [a]
isort' ac []       = ac
isort' ac (x : xs) = isort' (insertarOrdenado x ac)
    xs
```

Plegado de listas a izquierda

Escribamos una función `foldl` para abstraer el esquema de recursión iterativa:

```
foldl f ac []       = ac
foldl f ac (x : xs) = foldl f (f ac x) xs
```

¿Cuál es su tipo?

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

Toda recursión iterativa es una instancia de `foldl`.

Plegado de listas a izquierda

En general `foldr` y `foldl` tienen comportamientos diferentes:

```
foldr (★) z [a, b, c]  =  a ★ (b ★ (c ★ z))
foldl (★) z [a, b, c]  =  ((z ★ a) ★ b) ★ c
```

Si (★) es un operador asociativo y conmutativo, `foldr` y `foldl` definen la misma función. Por ejemplo:

```
suma     = foldr (+) 0      = foldl (+) 0
producto = foldr (*) 1      = foldl (*) 1
and      = foldr (&&) True  = foldl (&&) True
or       = foldr (||) False = foldl (||) False
```

Plegado de listas a izquierda

Ejemplo — pasaje de binario a decimal

```
bin2dec :: [Int] -> Int
bin2dec = foldl (\ ac b -> b + 2 * ac) 0
```

```
   bin2dec [1, 0, 0]
⇝  foldl (\ ac b -> b + 2 * ac)  0                              [1, 0, 0]
⇝  foldl (\ ac b -> b + 2 * ac)  (1 + 0)                        [0, 0]
⇝  foldl (\ ac b -> b + 2 * ac)  (0 + 2 * (1 + 0))              [0]
⇝  foldl (\ ac b -> b + 2 * ac)  (0 + 2 * (0 + 2 * (1 + 0)))   []
⇝  0 + 2 * (0 + 2 * (1 + 0))
⇝* 4
```

Plegado de listas a izquierda

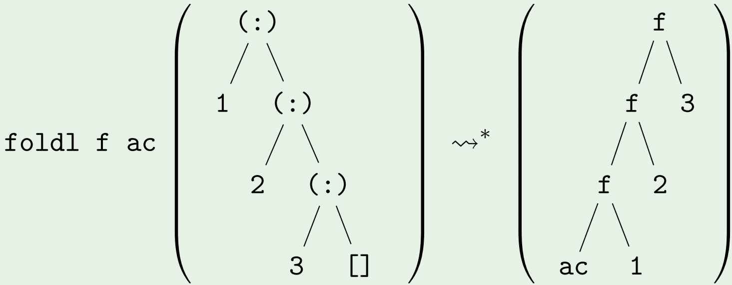
La función `foldl` es un operador de iteración.

Pseudocódigo imperativo:

función `foldl f ac xs` {
  mientras `xs` no es vacía {
    `ac` := `f ac (head xs)`
    `xs` := `tail xs`
  }
  devolver `ac`
}

Plegado de listas a izquierda

Ilustración gráfica del plegado a izquierda



En particular, se puede demostrar que:

```
foldl (flip (:)) [] = reverse
```

Resumen: esquemas de recursión sobre listas

Vimos los siguientes esquemas de recursión sobre listas:

1. Recursión estructural. .... `foldr`
2. Recursión primitiva. .... `recr`
3. Recursión iterativa. .... `foldl`

Para pensar

Recursión en simultáneo sobre más de una estructura

Definir la siguiente función usando `foldr`. (No tan fácil).

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

Relación entre recursión estructural y primitiva

1. Definir `foldr` en términos de `recr`. (Fácil).
2. Definir `recr` en términos de `foldr`. (No tan fácil).
   Idea: devolver una tupla con una copia de la lista original.

Relación entre recursión estructural e iterativa

1. Definir `foldl` en términos de `foldr`.
2. Definir `foldr` en términos de `foldl`.

Breve repaso

Esquemas de recursión sobre listas

Tipos de datos algebraicos

Esquemas de recursión sobre otras estructuras

Tipos de datos algebraicos

Conocemos algunos tipos de datos "primitivos":

`Char` `Int` `Float` `(a -> b)` `(a, b)` `[a]`

`String` (sinónimo de `[Char]`)

Se pueden definir nuevos tipos de datos con la cláusula `data`:

`data Tipo =` ⟨*declaración de los constructores*⟩

Tipos de datos algebraicos

Ejemplo — tipos enumerados

Muchos constructores sin parámetros:

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
```

Declara que existen constructores:

```
Dom :: Dia     Lun :: Dia   ...   Sab :: Dia
```

Declara además esos son los **únicos** constructores del tipo `Dia`.

```
esFinDeSemana :: Dia -> Bool
esFinDeSemana Sab = True
esFinDeSemana Dom = True
esFinDeSemana _   = False
```

```
>> esFinDeSemana Vie
⇝ False
```

Tipos de datos algebraicos

Ejemplo — tipos producto (tuplas/estructuras/registros/...)

Un solo constructor con muchos parámetros:

```
data Persona = LaPersona String String Int
```

Declara que el tipo `Persona` tiene un constructor (y **sólo ese**):

```
LaPersona :: String -> String -> Int -> Persona
```

```
nombre, apellido :: Persona -> String
nombre           (LaPersona n _ _) = n
apellido         (LaPersona _ a _) = a
fechaNacimiento  :: Persona -> Int
fechaNacimiento (LaPersona _ _ f) = f
```

```
rebecaGuber = LaPersona "Rebeca" "Guber" 1926
>> apellido rebecaGuber
⇝ "Guber"
```

Tipos de datos algebraicos

Ejemplo

Un tipo puede tener muchos constructores con muchos parámetros:

```
data Forma = Rectangulo Float Float
           | Circulo Float
```

Declara que el tipo `Forma` tiene dos constructores (y **sólo esos**):

```
Rectangulo  :: Float -> Float -> Forma
Circulo     :: Float -> Forma
```

```
area :: Forma -> Float
area (Rectangulo ancho alto) = ancho * alto
area (Circulo radio)         = radio * radio * pi
```

Tipos de datos algebraicos

Ejemplo

Algunos constructores pueden ser **recursivos**:

```
data Nat = Zero
         | Succ Nat
```

Declara que el tipo `Nat` tiene dos constructores (y **sólo esos**):

```
Zero  :: Nat
Succ  :: Nat -> Nat
```

¿Qué forma tienen los valores de tipo `Nat`?

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
```

Podemos definir alguna función usando los Nats:

```
isZero :: Nat -> Bool
isZero Zero = True
isZero _ = False
```

Otra función con Nat usando recursión:

```
doble :: Nat -> Nat
doble Zero     = Zero
doble (Succ n) = Succ (Succ (doble n))
```

La siguiente ecuación, ¿define un valor de tipo `Nat` o es un error?

```
infinito :: Nat
infinito = Succ infinito
```

¿Qué pasa cuando hacemos `isZero(infinito)`?

- Que se cuelgue o no depende de cómo se interpreten las definiciones recursivas.
- En Haskell se permite trabajar con estructuras infinitas.
  - En vez de exigir que termine la evaluación (visión *inductiva*), entiende que se pueden seguir obteniendo resultados sin terminar (visión *coinductiva*).
- En la materia hablaremos de estructuras finitas e infinitas.

Tipos de datos algebraicos

Tipos de datos algebraico — caso general

En general un tipo de datos algebraico tiene la siguiente forma:

$$
\begin{array}{lcl}
\texttt{data T} & = & \texttt{CBase}_1\ \langle \textit{parámetros} \rangle \\
 & & \ldots \\
 & | & \texttt{CBase}_n\ \langle \textit{parámetros} \rangle \\
 & | & \texttt{CRecursivo}_1\ \langle \textit{parámetros} \rangle \\
 & & \ldots \\
 & | & \texttt{CRecursivo}_m\ \langle \textit{parámetros} \rangle
\end{array}
$$

- Los constructores base no reciben parámetros de tipo `T`.
- Los constructores recursivos reciben al menos un parámetro de tipo `T`.
- Los valores de tipo `T` son los que se pueden construir aplicando constructores base y recursivos un número **finito** de veces y **sólo** esos.

(Entendemos la definición de `T` de forma **inductiva**).

Ejemplo: cuentas corrientes

```haskell
type Cuenta = String
data Banco = Iniciar
           | Depositar  Cuenta Int Banco
           | Extraer    Cuenta Int Banco
           | Transferir Cuenta Cuenta Int Banco

bancoPLP = Transferir "A" "B" 3 (Depositar "A" 10 Iniciar)

saldo :: Cuenta -> Banco -> Int

saldo cuenta Iniciar = 0
saldo cuenta (Depositar cuenta’ monto banco)
  | cuenta == cuenta’ = saldo cuenta banco + monto
  | otherwise         = saldo cuenta banco
saldo cuenta (Extraer cuenta’ monto banco)
  | cuenta == cuenta’ = saldo cuenta banco - monto
  | otherwise         = saldo cuenta banco
saldo cuenta (Transferir origen destino monto banco)
  | cuenta == origen  = saldo cuenta banco - monto
  | cuenta == destino = saldo cuenta banco + monto
  | otherwise         = saldo cuenta banco
```

Ejemplo: listas

Las listas son un caso particular de tipo algebraico:

```
data Lista a = Vacia | Cons a (Lista a)
```

O, con la notación ya conocida:

```
data [a] = [] | a : [a]
```

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

```
productoCartesiano xs ys =
  concat (map (\ x -> map (\ y -> (x, y)) ys) xs)
```

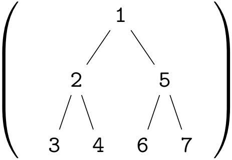
Ejemplo: árboles binarios

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Definamos las siguientes funciones:

```
preorder  :: AB a -> [a]
postorder :: AB a -> [a]
inorder   :: AB a -> [a]
```

t =



preorder t $\rightsquigarrow^*$ [1, 2, 3, 4, 5, 6, 7]
postorder t $\rightsquigarrow^*$ [3, 4, 2, 6, 7, 5, 1]
inorder t $\rightsquigarrow^*$ [3, 2, 4, 1, 6, 5, 7]

Ejemplo: árboles binarios

```
insertar :: Ord a => a -> AB a -> AB a
```

Pre: el árbol de entrada es un ABB (sin repetidos).
Post: el árbol resultante es un ABB que contiene a los elementos del ABB de entrada y al elemento dado.

```
insertar x Nil = Bin Nil x Nil
insertar x (Bin izq y der)
  | x < y     = Bin (insertar x izq) y der
  | x > y     = Bin izq y (insertar x der)
  | otherwise = Bin izq y der
```

Breve repaso

Esquemas de recursión sobre listas

Tipos de datos algebraicos

Esquemas de recursión sobre otras estructuras

Recursión estructural

En el caso de las listas, dada una función `g :: [a] -> b`:

$$\begin{array}{lcl} \texttt{g []} & = & \langle \textit{caso base} \rangle \\ \texttt{g (x : xs)} & = & \langle \textit{caso recursivo} \rangle \end{array}$$

decíamos que `g` estaba dada por recursión estructural si:

- El caso base devuelve un valor `z` "fijo" (no depende de `g`).
- El caso recursivo **no** puede usar las variables `g` ni `xs`, salvo en la expresión `(g xs)`.

Recursión estructural

La recursión estructural se generaliza a tipos algebraicos en general.
Supongamos que `T` es un tipo algebraico.
Dada una función `g :: T -> Y` definida por ecuaciones:

$$
\begin{array}{lcl}
\texttt{g (CBase}_1\ \langle \textit{parámetros} \rangle\texttt{)} & = & \langle \textit{caso base}_1 \rangle \\
\ldots & & \\
\texttt{g (CBase}_n\ \langle \textit{parámetros} \rangle\texttt{)} & = & \langle \textit{caso base}_n \rangle \\
\texttt{g (CRecursivo}_1\ \langle \textit{parámetros} \rangle\texttt{)} & = & \langle \textit{caso recursivo}_1 \rangle \\
\ldots & & \\
\texttt{g (CRecursivo}_m\ \langle \textit{parámetros} \rangle\texttt{)} & = & \langle \textit{caso recursivo}_m \rangle
\end{array}
$$

Decimos que `g` está dada por **recursión estructural** si:

1. Cada caso base se escribe combinando los parámetros.
2. Cada caso recursivo se escribe combinando:
   - Los parámetros del constructor que no son de tipo `T`.
   - El llamado recursivo sobre cada parámetro de tipo `T`.

   Pero:
   - Sin usar los parámetros del constructor que son de tipo `T`.
   - Sin hacer a otros llamados recursivos.

Recursión estructural

```
data AB a = Nil
          | Bin (AB a) a (AB a)
```

Ejemplo

Definamos una función `foldAB` que abstraiga el esquema de recursión estructural sobre árboles binarios.

```
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
```

```
foldAB cNil cBin Nil           = cNil
foldAB cNil cBin (Bin i r d) =
  cBin (foldAB cNil cBin i) r (foldAB cNil cBin d)
```

Recursión estructural

Para llevarse para pensar

1. ¿A qué es igual `(foldAB Nil Bin)`?
2. Definir `mapAB :: (a -> b) -> AB a -> AB b` usando `foldAB`.
3. Definir `maximo :: Ord a => AB a -> Maybe a` usando `foldAB`.
4. Definir `altura :: AB a -> Int` usando `foldAB`.
5. ¿Se puede definir la función `esABB :: Ord a => AB a -> Bool` usando `foldAB`?
6. ¿Se puede definir la función `caminoMasLargo :: AB a -> [a]` usando `foldAB`?

¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ¿ ? ? ? ? ? ? ? ? ?

Lectura recomendada

Artículo de Hutton.
Graham Hutton. *A tutorial on the universality and expressiveness of fold.*
J. Functional Programming 9 (4): 355–372, julio de 1999.

Comentarios: recursión estructural

Casos degenerados de recursión estructural

Es recursión estructural (no usa la cabeza):

```
length :: [a] -> Int
length []       = 0
length (_ : xs) = 1 + length xs
```

Es recursión estructural (no usa el llamado recursivo sobre la cola):

```
head :: [a] -> a
head []       = error "No tiene cabeza."
head (x : _) = x
```