Funciones de Orden Superior y Currying

- Expresiones Lambda: Definición generalizada de funciones en Scheme.
- Funciones de orden superior.
- Definiciones locales.
- Evaluación parcial (Currying).

Expresiones Lambda (λ)

- Cálculo-λ - Alonzo Church (1932-33)
- Lenguajes Funcionales: versión "suavizada"
- Necesidad de funciones "anónimas"
  - Definir una función directamente como argumento
  - Definir una función como resultado de una función
  - Definir una función local a una función

Ventaja **Scheme**: S-Expresiones
(Comparten la misma representación)

Ejemplos de F.O.S.

- Definir:
  - 1 $f(a,b)=\sum_{n=a}^{b} n$
  - 2 $g(a,b)=\sum_{n=a}^{b} n^3$
  - 3 $h(a,b)=\sum_{n=a}^{b} \frac{1}{(4n-3a)(4n-3a+s)}$

```
f(a,b) ::= si a>b
             entonces 0
             sino a+f(a+1,b)
           fsi

g(a,b) ::= si a>b
             entonces 0
             sino a*a*a+g(a+1,b)
           fsi

h(a,b) ::= si a>b
             entonces 0
             sino 1/(a*(a+2)) +
h(a+4,b)
      fsi
```

```
sum(term, a, next, b) ::=
        si a>b
          entonces 0
          sino term(a)+sum(term, next(a), next, b)
        fsi
```

```
term(X) ::= X
next(X) ::= X+1
f(a,b) ::= sum(term, a, next, b)
```

Ejemplos de F.O.S.

- Definición en Scheme de $f(a,b) = \sum_{n=a}^{b} n$

```
;; sum::(Int->Int) x Int x (Int->Int) x Int -> Int
(define (sum term a next b)
  (if (> a b) 0
      (+ (term a) (sum term (next a) next b))))


;; definimos term y next con expresiones lambda
;;
;; f::Int x Int -> Int
(define (f a b)
  (sum (lambda (X) X) a (lambda (X) (+ X 1)) b))
```

Expresiones Lambda

- Forma general de expresiones Lambda

(lambda (par1 …) <expresion>) ⇒ función λ

- Definición con nombre

(define <nom-funcion> <expr-lambda>)
⇒
función <nom-funcion>

Ejemplo: Función como resultado de evaluar una función

```
> (define (make-suma num) (lambda (X) (+ X num)))
make-suma

> (make-suma 4)
(lambda(x) (+ x 4))

> ((make-suma 4) 7)
11
```

```
> (define suma4 (make-suma 4))
suma4

> (suma4 7)
11
```

Funciones de orden superior

- Definición: Una función se dice de orden superior si :
  - alguno de sus argumentos es una función
  - o si devuelve una función
  - o si devuelve una estructura conteniendo una función.

- Ventaja:
  - Programas más concisos sin perder generalidad
  - Programas genéricos, funciones parametrizadas por medio de otras funciones
  - Alto nivel de reutilización
  - Generación automática de código (funciones)

F.O.S. definidas sobre listas

- Filter – filtra elementos no deseados de una lista

(filter <funcion-test> <list>)

```
>(filter positive? '(-3 4 6 -2 9 -11))
(4 6 9)
```

- Take-while – devuelve el prefijo de una lista

(take-while f lista)

```
>(take-while positive? '(1 2 -3 9))
(1 2)
```

- Drop-until – devuelve el sufijo de una lista

(drop-until f lista)

```
(drop-until number? '(jose varon 19 23)) => (19 23)
```

```
(take-while (lambda(x) (not(eq? x '<)))
(cdr (drop-until (lambda(x)(eq? x '>)) '( < a > Enlace < / a > ))))
```

F.O.S. Predefinidas: Apply

- Apply – aplica una función a una lista de argumentos
  - Ejemplo: calcular el máximo de los elementos de una lista `ls`

```
> (max ls)
```

max(n1 ...) no trabaja sobre listas

Solución A: Construir un programa recursivo que calcule *max* de una lista *ls*.

Solución B: **(apply** `<funcion> arg1 ... <lista-de-elementos>`**)** ⇒

(`<funcion> arg1 ... <elementos-de-la-lista-de-elementos>`**)**

```
> (apply max '(2 4))    ; equivalente a (max 2 4)
4
> (apply + '(4 11 23)) ; equivalente a (+ 4 11 23)
38
> (apply max 0 `(3 4 1)); equivalente a (max 0 3 4 1)
4
```

Expresiones Lambda (otras sintaxis)

- Número arbitrario de argumentos

(lambda <Lista-Args> <expresion>) ⇒ función λ

```
> (define num-args (lambda ListaArgs (length ListaArgs))
num-args

> (num-args 'a 'b 'c)
3
> (num-args '(1 b c) 5)
2
```

Expresiones Lambda (otras sintaxis)

- Cota inferior de argumentos: n –requeridos + ?-opcionales

(lambda (<Arg-1>...<Arg-n> . <LArgs>) ⇒ función λ

  - LArgs: la lista de argumentos restantes

```
> (define mis-args (lambda (a b c . Largs)
    (append (list a b c) Largs))
mis-args

> (mis-args 1 2 3 4 5 6 7)
(1 2 3 4 5 6 7)

> (mis-args 'a 'b 'c)
(a b c)

> (mis-args 'a 'b)
ERROR – FALTAN ARGUMENTOS
```

F.O.S. Predefinidas: Map

- Map – aplica una función de aridad n, a los i-simos elementos de n-listas

(map <funcion> <list1> <list2> ...)

```
> (map car '((a b) (c d) (e f)))
(a c e)
> (map + '(1 2) '(4 5))
(5 7)
```

```
> (map (lambda(x) (+ x 2)) '(1 2 3 4))
(3 4 5 6)
```

```
(map (lambda(x) (member 'a x)) '((a b c) (b c d) (c d a)))
(#t #f #t)
```

```
> (apply and ) ⇒ #f
```

F.O.S. Predefinidas: Ejemplos

- Distancia Euclídea de N dimensiones
  - `Dist(A,B)` $= \sqrt[2]{\sum_{i=1}^{N}(A_i - B_i)^2}$

```
;; suma los elementos de una lista
;; sum::Int -> Int
(define (sum L)
  (apply + L))
```

```
;; diferencia al cuadrado de dos números
;; Number x Number -> Number
(define (dif2 x y)
  (expt (- x y) 2))
```

Con funciones auxiliares

```
;; distancia euclídea de dos puntos N-dimensionales
;; dist::(list-of Number) x (list-of Number) -> Number
(define (dist A B)
  (sqrt (sum (dif-cuadradas A B))))
```

Directamente utilizando FOS Y λ-expresiones

```
;; distancia euclídea de dos puntos N-dimensionales
;; dist2::(list-of Number) x (list-of Number) -> Number
(define (dist2 A B)
  (sqrt ; raíz cuadrada
   (apply + ; de la suma de ...
          (map (lambda(x y)(expt (- x y) 2))  ; las diferencias al cuadrado
               A B)                           ; de los i-simos elementos
          )
   )
  )
```

F.O.S. Predefinidas: Ejemplos

- Composición

```
((compose sqrt *) 12 75) ⇒ 30

;; (A -> B) x (C1 x ...x Cn -> A) -> B
(define (compose f g)
        (lambda args
          (f (apply g args)))))
```

- Matriz traspuesta

```
(tras '((1 2) (3 4) (5 6))) ⇒ ((1 3 5) (2 4 6))

;; Matriz -> Matriz
(define (tras M)
  (apply map list M)
```

F.O.S. definidas sobre listas

- fold(l/r) – pliegue de los elementos a un solo valor

```
(foldl <funcion(elem,Res)> ini <list>)
(foldr <funcion(elem,Res)> ini <list>)
```

- Forma generalizada de foldl/r:

```
(foldl <f(e1,…en,Res)> ini <list1>…<listn>)
```

Funciones de plegado de listas: foldl y foldr


```
foldl :: (A x B -> B) x B x (listof A) -> B
```

propósito:

```
(foldl f base (list a1 a2 a3 .. an)) =
                                    (f an ...(f a3 (f a2 (f a1 base)))...)
```

```
(foldl + 0 '(4 3 2 1)) => 10        (sumalista LNums)

(foldl cons '( ) '(a b c d)) => (d c b a)  (reverse Lista)

(foldl append '( ) '((1 2)(3 4)(5 6)) => (5 6 3 4 1 2)
                                    (flat (listof listas))
```

¿ORDEN?

Funciones de plegado de listas: foldl y foldr


```
foldr :: (A x B -> C) x B x (listof A) -> C
```

propósito:

```
(foldr f base (list a1 .. an2 an1 an)) =
                (f a1 ...(f an2 (f an1 (f an base)))...)
```

```
(foldr + 0 '(4 3 2 1))  => 10        (sumalista LNums)

(foldr cons '( ) '(a b c d))  => (a b c d)  (f-id Lista)

(foldr append '( ) '((1 2)(3 4)(5 6))   => (1 2 3 4 5 6)
                                  (flat (listof listas))
```

¿ORDEN?

Funciones de plegado de listas: foldl y foldr

Conjuntos: **`intersec:: Set x Set -> Set`**

- Utilizando **`(filter f lista)`**

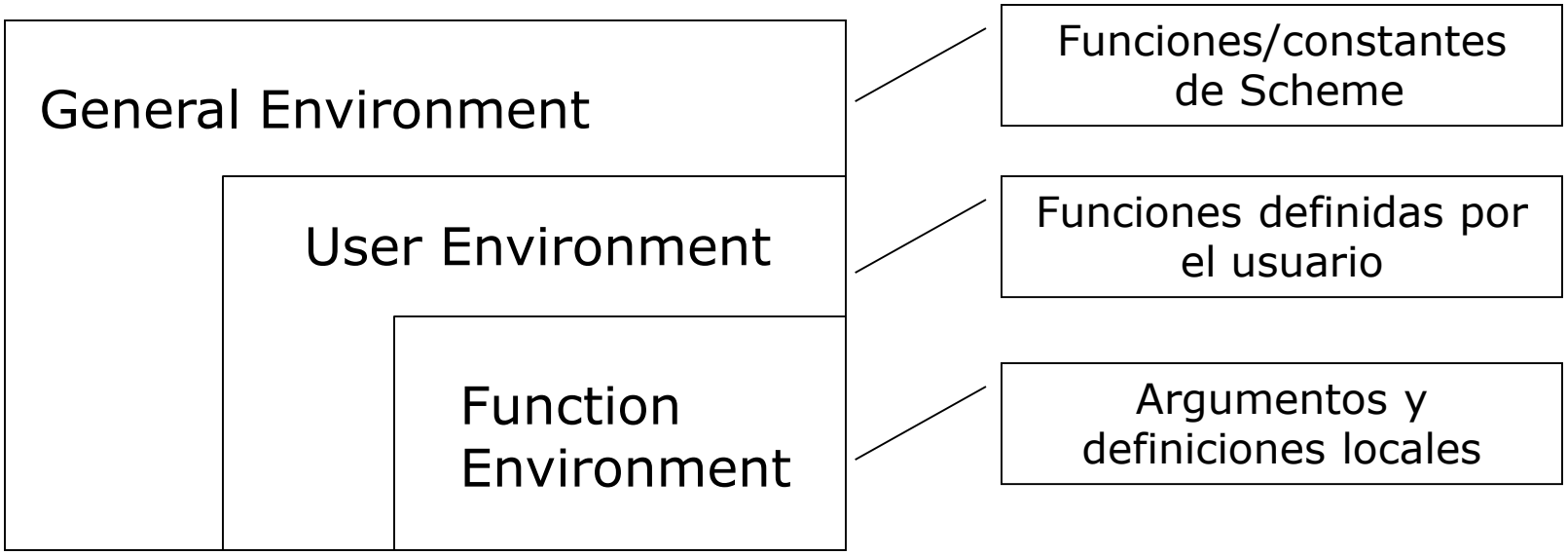
```
(define (intersec A B)
        (filter (lambda(x)(member x A)) B))
```

- Utilizando **`(foldl f base lista)`**

```
(define (intersec A B)
        (foldl (lambda(e r)
                       (if (member e A) (cons e r) r)) '() B)
```

Definiciones Locales – Let(*) y Letrec

- Ámbito de definición de los objetos del lenguaje



Let

- Nos permite asociar una definición a un conjunto de símbolos, limitando su ámbito a dicha expresión.
- Sintaxis:

```
(let ( (id1 val1)  (id2  val2)  … (idn  valn) ) body)
```

  - Suma de dos definiciones internas (a y b) en una expresión

```
(let ((a 2) (b 3))
  (+ a b) )  ⇒ 5
```

```
(let ((a 2) (b (+ a 3)))
  (+ a b) ) ⇒ ERROR ¿a?
```

Ámbito de a y b

  - Distancia Euclídea

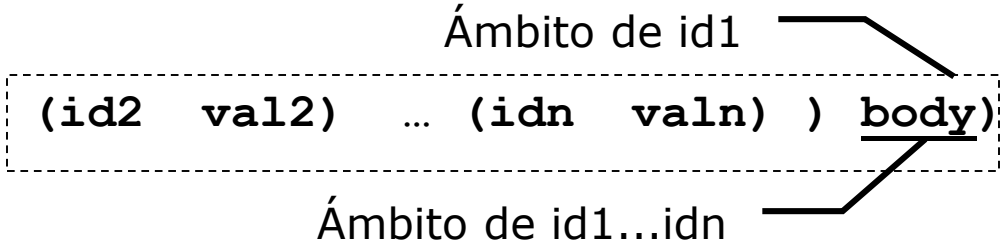
```
(define (distancia X Y)
  (let ( (dif2 (lambda (X1 X2)
                 (expt (- X1 X2) 2)))
    (sqrt (+ (dif2 (car X) (car Y) )
             (dif2 (cadr X) (cadr Y)))))))
```

```
(distancia '(2 3) '(3 6))
```

$\Rightarrow \sqrt{(2-3)^2+(3-6)^2} = 3{,}16$

Let*

- Similar a Let pero las definiciones se realizan de manera secuencial.
- Sintaxis:



Ejemplos:

- Let y Let* anidados

```
(let ((x 2) (y 3))
  (let* ([x 7] [z (+ x y)])
    (* z x)))  ⇒ 70
```

- Dos Let anidados

```
(let ((x 2) (y 3))
  (let ([x 7] [z (+ x y)])
   (* z x))) ⇒ 35
```

Letrec

- Similar a Let* **pudiendo utilizarlos recursivamente.**
- Sintaxis:

Ámbito de id1-idn

```
(letrec ( (id1 val1)  (id2  val2)  … (idn  valn) ) body )
```

  - Factorial

```
(define (fact n)
 (letrec ((aux (lambda (n) (if (zero? n) 1
                               (* n (aux (- n 1)))))))
          (if (integer? n) (aux n) (error "n no es entero"))))

(fact 5)      ⇒     120
```

  - Definiciones mutuamente recursivas (par? ⟷ impar?)

```
(letrec ((par? (lambda (n) (if (zero? n) #t (impar? (- n 1)))))
         (impar?  (lambda (n) (if (zero? n) #f (par? (- n 1))))))
   (par? 88))
                        ⇒     #t
```

Currying - Currificación

- Definición:

  Se define "currying" como la capacidad de definir una función de n argumentos:

  f:A1 x A2 x … x An -> B

  como n funciones de 1 argumento:

  f':A1 -> A2 -> A3 ... ->An -> B

  f': F.O.S. Devuelve exp.lambda ← función parametrizable

Currying - Currificación

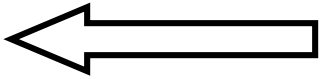
- Ejemplo:

```
(+ 5 4) ⇒  9

(define add5 (lambda (n) (+ 5 n)))
o
(define (add5 n) (+ 5 n))

(add5 4) ⇒ 9
```

```
(define (curried+ m)
    (lambda(n) (+ m n)))
```



Versión currificada de +

```
(curried+ 5) ⇒ (lambda(n) (+ 5 n))

((curried+ 5) 7) ⇒ 12

(define add5 (curried+ 5))
```

Currying - Currificación

- Ejemplo:

```
;member-c:: A -> ((list-of A) -> Bool)
(define (member-c item)
  (letrec ([helper (lambda (ls)
                     (cond ((null? ls)              #f)
                           ((equal? (car ls) item) ls)
                           (else   (helper (cdr ls)))))])
    helper))
```

member currificado

```
(define (member e l) ((member-c e) l))
```

member redefinido

```
(member-c 5) ⇒ (lambda (ls)
                 (cond ((null? ls) #f)
                       ((equal? (car ls) 5) ls)
                       (else (helper (cdr ls)))))
```

member-c(5):Lista->Bool

```
((member-c 5) '(1 2 3)) ⇒ #f
```

```
(define esta-el-5? (member-c 5))
(esta-el-5? '(1 2 5 3))        ⇒      #t
```

esta-el-5?:Lista->Bool

Conclusiones Currificación

- Redefine F/N (= M+K) argumentos como F'/M->λ/K
- De forma general:

  F/N ≡ F'/1-> λ1/1->... λn-1/1

- Técnica de generación automática de código por parametrización