



UNIVERSIDAD TÉCNICA  
FEDERICO SANTA MARÍA

DEPARTAMENTO  
DE INFORMÁTICA

---

# LENGUAJES DE PROGRAMACIÓN

**Equipo de Profesores:**

**Jorge Díaz Matte – José Luis Martí Lara – Rodrigo Salas Fuentes**

---

# Unidad 6

## Programación Funcional y Scheme

6.1 Introducción a la Programación Funcional

6.2 Introducción al Lenguaje de Programación Scheme

6.3 Condicionales

6.4 Recursión

6.5 Asignación

6.6 Ligado de Variables

6.7 Otras operaciones en Scheme

## 6.1 Introducción a la Programación Funcional

# Programación Funcional

- Paradigma diferente a los imperativos, que se aleja de la máquina de von Neumann.
- Basado en funciones matemáticas (notación funcional *lambda* de Church).
- No existen realmente arquitecturas de computadores que permitan la eficiente ejecución de programas funcionales.
- LISP es el primer lenguaje funcional, del cual derivan **Scheme**, Common LISP, ML y Haskell.
- La programación funcional ha influenciado muchos lenguajes modernos.

# Funciones Matemáticas

- Una función es una proyección de un conjunto dominio a otro que es el rango:  $f: D \rightarrow R$
- La evaluación de funciones está controlada por recursión y condiciones (los lenguajes imperativos lo hacen normalmente por secuencias e iteraciones).
- Funciones matemáticas entregan siempre el mismo valor para el mismo conjunto de argumentos y, por lo tanto, no tiene efectos laterales.

# Funciones Matemáticas: ejemplos

## MATEMÁTICAS

- Definición de una función:  $\text{cubo}(x) \equiv x * x * x$ , con  $x$  real
- Aplicación de la función:  $\text{cubo}(2.0) \rightarrow 8.0$

## NOTACIÓN LAMBDA DE CHURCH

- Separa definición de la función de su nombre
- Definición de una función:  $\lambda(x) x * x * x$
- Aplicación de la función:  $(\lambda(x) x * x * x) (2.0) \rightarrow 8.0$

# Formas Funcionales (funciones de primer orden)

Toman funciones como parámetros y/o producen funciones como resultado:

- Composición de Funciones:  $h \equiv f \circ g$  entonces  $h(x) \equiv f(g(x))$
- Construcción - lista de funciones que se aplican a un mismo argumento:  
 $[f, g](x)$  produce  $(f(x), g(x))$
- Aplicación a todo - una misma función se aplica a una lista de argumentos:  $\alpha(f, (x, y, z))$  produce  $(f(x), f(y), f(z))$

# Fundamentos de la Programación Funcional

- La programación funcional no usa variables ni asignación.
- La repetición debe ser lograda con recursión.
- Un [programa](#) consiste en la definición de funciones y la aplicación de éstas.
- La [ejecución del programa](#) no es nada más que la evaluación de funciones.
- La [transparencia referencial](#) se refiere a que la evaluación de una función siempre produce el mismo resultado.



# Lenguajes Funcionales

- El lenguaje provee algunas funciones básicas (núcleo), que son primitivas para construir funciones más complejas.
- Se definen algunas estructuras para representar datos de los parámetros y resultados de las funciones.
- Normalmente se implementan mediante interpretadores, pero también se pueden compilar.

## 6.2 Introducción al Lenguaje de Programación Scheme

# Scheme (1)

## Orígenes:

- Desarrollado en el MIT a mediados del '70, por Guy L. Steele y Gerald J. Sussmann.
- Es un dialecto de LISP (McCarthy, 1958).
- Usado inicialmente para enseñanza de programación.

## Scheme (2)

Características:

- Pequeño, con sintaxis y semántica simple.
- Nombres tienen sólo ámbito estático (a diferencia de LISP).
- Funciones son entidades de primera clase, y por lo tanto se tratan como cualquier valor.
- Tiene recolección automática de basura.

## Scheme (3)

- Estándares:
  - Revised Report on the Algorithmic Language Scheme  
(<http://www.r6rs.org>)
  - 1178-1990 - IEEE Standard for the Scheme Programming Language  
(<https://ieeexplore.ieee.org/document/159138>)

# Scheme: Ambiente Interactivo

- Corresponde al ciclo: leer, evaluar e imprimir (denominado REPL).
- El sistema entrega un prompt, se ingresa la expresión, el sistema evalúa y entrega el resultado. Ej.:  
    “Hola Scheme” => “Hola Scheme”  
    *(Toda constante evalúa en la misma constante)*
- Es posible cargar y salvar en un archivo para facilitar el proceso de desarrollo.

# Scheme: Identificadores

- Corresponden a palabras claves, variables y símbolos, que no son sensibles a mayúsculas.
- Se forman de:
  - mayúsculas y minúsculas ['A' .. 'Z', 'a' .. 'z']
  - dígitos ['0' .. '9']
  - caracteres ['?', '!', '.', '+', '-', '\*', '/', '<', '=', '>', ':', '\$', '%', '^', '&', '\_', '~']
- Identificadores no pueden comenzar un número.
  - Válidos: X3, ?\$!!!, Abcd, AbcD
  - No lo es: 8id

# Scheme: Constantes básicas

- **String**: se escribe usando comillas dobles. Ej.: “Un string es sensible a Mayúsculas”
- Un **caracter** precede de `#\`. Ej.: `#\a`
- Un **número** pueden ser enteros, fraccionarios, punto flotante y en notación científica. Ejs.: `-365`, `1/4`, `23.46`, `1.3e27`
- **Números complejos** en coordenadas rectangulares y polares. Ejs.: `2.7-4.5i`   `+3.4@-0.5`
- **Booleanos** son los valores `#f` (falso) y `#t` (verdadero)



# Scheme: Funciones Aritméticas

- Los nombres  $+$ ,  $-$ ,  $*$  y  $/$  son nombres reservados para las operaciones aritméticas.

- Funciones se escriben como listas en **notación prefija**:

$(+ \ 1/2 \ 1/2)$	$\Rightarrow 1$
$(- \ 2 \ (* \ 4 \ 1/3))$	$\Rightarrow 2/3$
$(/ \ (* \ 6/7 \ 7/2) \ (- \ 4.5 \ 1.5))$	$\Rightarrow 1.0$

# Scheme: Listas

- Las listas se escriben con paréntesis redondos: `(a b c d)`
- Listas contienen elementos de cualquier tipo, y se pueden anidar:  
`(lambda (x) (* x x))`
- Una función se escribe como una lista en notación prefija, correspondiendo el primer elemento de la lista a la función y los siguientes a los argumentos: `(+ 3 14) => 17`

# Scheme: Listas

- Toda lista es evaluada, salvo que se especifique lo contrario mediante citación simple (*quote*):

(quote (a b c d)) => (a b c d)

'(a b c d) => (a b c d)

(a b c d) => Error

- Al no usar “quote”, *Scheme* trata de evaluar considerando en el ejemplo a **a** como variable de función.

# Scheme: Listas

Operadores básicos:

- **car** devuelve el primer elemento de la lista:

`(car '(a b c d)) => a`

- **cdr** devuelve el resto de la lista (sin el primer elemento):

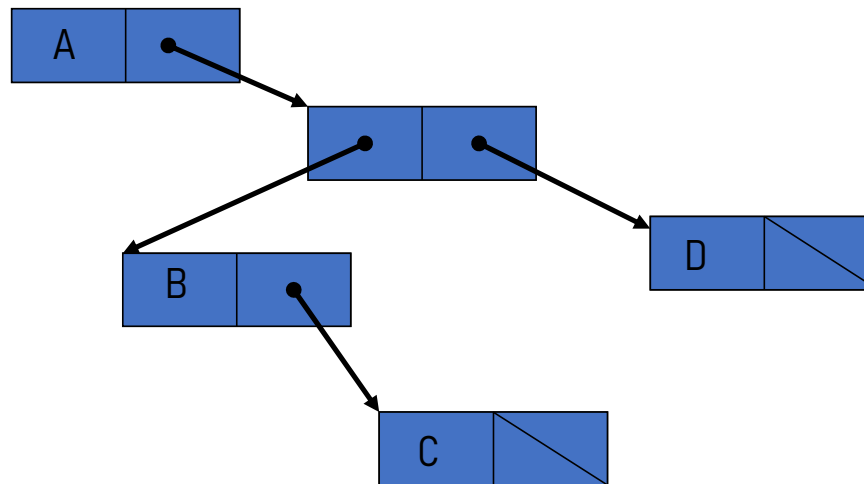
`(cdr '(a b c d)) => (b c d)`

Observación: *first* y *rest* se puede usar en lugar de *car* y *cdr*, respectivamente

# Scheme: Listas

## Estructura de una Lista

(A (B C) D)



# Scheme: Listas

Constructores:

- **cons** construye una nueva lista cuyo *car* y *cdr* son los dos argumentos:

`(cons 'a '(b c d))`  $\Rightarrow$  `(a b c d)`

`(cons (car '(a b c))(cdr '(a b c)))`  $\Rightarrow$  `(a b c)`

- **list** construye una lista con todos los argumentos:

`(list 'a 'b 'c 'd)`  $\Rightarrow$  `(a b c d)`

`(list)`  $\Rightarrow$  `()`

- **append** fusiona dos listas en una sola:

`(append '(a b) '(c d))`  $\Rightarrow$  `(a b c d)`

# Scheme: let

- **let** permite definir variables que se ligan a un valor en la evaluación de expresiones.
- Sintaxis:

**(let ((var1 val1) ... ) exp1 exp2 ... )**

- Ejemplo:

**(let ((x 2) (y 3))  
 (\* (+ x y) (- x y)))**      => -5

Importante:  
las variables sólo  
tienen ámbito local

# Scheme: expresiones lambda

- Permite crear un nuevo procedimiento

- Sintaxis:

`(lambda (var1 var2 ... ) exp1 exp2 ... )`

- Ejemplo:

`((lambda (x) (* x x)) 3)`       $\Rightarrow 9$

¡Una expresión lambda es un  
objeto tipo procedimiento  
que no tiene nombre!



# Scheme: expresiones lambda

Ejemplo:

```
(let ((square (lambda (x) (* x x))))  
      (list (square 2)  
            (square 3)  
            (square 4)  
            )  
      )  
  
=> (4 9 16)
```

# Scheme: relación entre let y lambda

- Nótese que:

`(let ((var1 val1) ... (varm valm)) exp1 ... expn)`

equivale a:

`((lambda (var1 ... varm) exp1 ... expn) val1 ... valm)`

# Scheme: especificación de parámetros formales

- Lista propia de parámetros ( $\text{var}_1 \text{var}_2 \dots \text{var}_n$ )

`((lambda (x y) (list x y)) 1 2)`  $\Rightarrow$  `(1 2)`

- Parámetros único  $\text{var}_r$

`((lambda x (list x)) 1 2)`  $\Rightarrow$  `((1 2))`

- Lista impropia de parámetros ( $\text{var}_1 \text{var}_2 \dots \text{var}_n . \text{var}_r$ )

`((lambda (x . y) (list x y)) 1 2 3)`  $\Rightarrow$  `(1 (2 3))`

# Scheme: definiciones de nivel superior (1)

- Las variables definidas con **let** y **lambda** son sólo visibles en el cuerpo de las expresiones (local).
- El procedimiento **define** permite definir variables de nivel superior (global).
- Definiciones de nivel superior permiten visibilidad en cada expresión donde no sean escondidas por otro ligado. Ej.: una variable definida con el mismo nombre mediante **let** oculta a las de nivel superior.

## Scheme: definiciones de nivel superior (2)

- Uso de define:

(define pi 3.1416)  $\Rightarrow$  pi

(define square (lambda (x) (\* x x)))  $\Rightarrow$  square

(square 3)  $\Rightarrow$  9

(let ((x 2)(square 4)) (\* x square))  $\Rightarrow$  8

## Scheme: definiciones de nivel superior (3)

- La forma: `(define var0 (lambda (var1 ... varn) e1 ...))`  
se puede abreviar como: `(define (var0 var1 ... varn) e1 ...)`
- Ejemplo: las siguientes expresiones son equivalentes  
`(define square (lambda (x) (* x x)))`  
`(define (square x) (* x x))`

## 6.3 Condicionales

# Expresiones condicionales

- En Scheme también es posible condicionar la realización de determinada tarea.
- Sintaxis: `(if test consecuencia alternativa)`
- Ejemplo:

```
(define (abs n) (if (> n 0)
                    n
                    (- 0 n)))
```

```
(abs -27)    => 27
```



# Expresiones condicionales Múltiples

- Scheme provee expresiones que se evalúan condicionalmente.
- Sintaxis:

```
(cond
  (test1 exp1)
  (test2 exp2) ...
  (else expn)
)
```

- El uso de **else** es opcional, siendo equivalente su uso a colocar **#t**.

# Expresiones condicionales Múltiples

- Ejemplo:

```
(define abs2
  (lambda (x)
    (cond ((= x 0) 0)
          ((< x 0) (- 0 x))
          (else x)
    )
  )
)
```

# Predicados (1)

- Procedimientos para expresiones relacionales: =, <, >, <= y >=

Ejemplo: (= 3 4) => #f

- Procedimientos para expresiones lógicas: or, and y not

Ejemplo: (and (> 5 2) (< 5 10)) => #t

## Predicados (2)

- Lista nula: `null?` `(null? '())`  $\Rightarrow$  `#t`

- Argumentos equivalentes: `eqv?` `(eqv? 'a 'a)`  $\Rightarrow$  `#t`

- Ejemplo:

```
(define (reciproco n)
  (if (and (number? n) (not (= n 0)))
      (/ 1 n)
      "reciproco: división no válida")
  )
)
```

## Predicados (3)

- Cualquier objeto se interpreta como **#t**
- La lista nula '()' equivale a **#f** (sólo en Estándar IEEE)
- Se definen los predicados:
  - pair?** : verifica si es un par (lista propia o impropia)
  - number?** : verifica si es número
  - string?** : verifica si es un string

## 6.4 Recursión

# Tipos de Recursión (1)

- **Directa**: la función se invoca a sí misma.
- **Indirecta**: la función invoca a otra función, y quizá ésta a otra(s), que termina invocando a la primera.
- **Lineal**: existe una única invocación recursiva.
- **Múltiple**: existe más de una invocación recursiva.
  - **Anidada**: dentro de una invocación recursiva se tiene como parámetro otra invocación recursiva

## Tipos de Recursión (2)

- **de Cabeza**: la invocación recursiva se hace al principio, antes que el resto de las sentencias.
- **Intermedia**: las sentencias aparecen y después de la invocación recursiva.
- **de Cola**: la invocación recursiva se hace al final, después que el resto de las sentencias.



# Recursión Directa: ejemplos en Scheme (1)

```
(define length  
  (lambda (ls)  
    (if (null? ls)  
        0  
        (+ 1 (length (cdr ls))))))
```

=> length

```
(length '(a b c d))          => 4
```

## Recursión Directa: ejemplos en Scheme (2)

;; El siguiente procedimiento busca x en la lista ls,  
;; devuelve el resto de la lista después de x o ()

```
(define memv  
  (lambda (x ls)  
    (cond ((null? ls) ())  
          ((eqv? x (car ls)) (cdr ls))  
          (else (memv x (cdr ls))))))
```

=> memv

(memv 'c '(a b c d e))      => (d e)

## Recursión Directa: ejemplos en Scheme (3)

;; El siguiente procedimiento devuelve la lista equivalente a ls eliminando  
;; toda ocurrencia de x

```
(define remv  
  (lambda (x ls)  
    (cond ((null? ls) '())  
          ((eqv? x (car ls)) (remv x (cdr ls)))  
          (else (cons (car ls) (remv x (cdr ls)))))))
```

=> remv

```
(remv 'c '(a b c d e c r e d)) => (a b d e r e d)
```

# Recursión de Cola

- Es un tipo de recursión directa.
- Cuando un llamado a procedimiento aparece al final de una expresión lambda, es un **llamado de cola** (no debe quedar nada por evaluar de la expresión lambda, excepto retornar el valor del llamado).
- **Recursión de cola** es cuando un procedimiento hace un llamado de cola hacia si mismo, o indirectamente a través de una serie de llamados de cola hacia si mismo.

Son llamados de cola a f:

```
(lambda () (if (g) (f) #f))  
(lambda () (or (g) (f)))
```

pero no lo son respecto a g:

```
(lambda () (if (g) (f) #f))  
(lambda () (or (g) (f)))
```

# Recursión de Cola: propiedad

- Scheme trata las llamadas de cola como un **goto** o salto de control (jump).
- Por lo tanto, se pueden hacer un número indefinido de llamados de cola **sin causar overflow** del stack.
- Es recomendable transformar algoritmos que producen mucho anidamiento en la recursión a uno que sólo use recursión de cola.

# Recursión de Cola: Ejemplos

Recursión  
Simple:

```
(define factorial  
  (lambda (n)  
    (let fact ((i n))  
      (if (= i 0)  
          1  
          (* i (fact (- i 1)))))))
```

$$n! = n * (n-1)! \\ 0! = 1$$

---

Recursión  
de Cola:

```
(define factorial  
  (lambda (n)  
    (let fact ((i n) (a 1))  
      (if (= i 0)  
          a  
          (fact (- i 1) (* a i)))))
```

$$n! = n * (n-1) * (n-2) * \dots 2 * 1$$

# Recursión de Cola: Ejemplos

**fib(n) = fib(n-1) + fib(n-2)**  
**fib(0) = 0 y fib(1) = 1**

Recursión  
Simple:

```
(define fibonacci  
  (lambda (n)  
    (let fib ((i n))  
      (cond ((= i 0) 0)  
            ((= i 1) 1)  
            (else (+ (fib (- i 1)) (fib (- i 2))))))))
```

Recursión  
de Cola:

```
(define fibonacci1  
  (lambda (n)  
    (if (= n 0)  
        0  
        (let fib ((i n) (a1 1) (a0 0))  
          (if (= i 1)  
              a1  
              (fib (- i 1) (+ a1 a0) a1))))))
```

## 6.5 Asignación



# Asignación

- **let** permite ligar un valor a una (nueva) variable en su cuerpo (local), mientras que **define** permite ligar un valor a una (nueva) variable de nivel superior.
- Sin embargo, **let** y **define** no permiten cambiar el ligado de una variable ya existente, como lo haría una asignación.

# Asignación

- **set!** permite en Scheme re-ligar a una variable existente un nuevo valor, como lo haría una asignación.
- No establece un nuevo ligado, sino que cambia uno existente.
- Evaluaciones posteriores evalúan al nuevo valor.
- Son útiles para actualizar un estado y para crear estructuras recursivas.

# Asignación: Ejemplo

```
(define abcde '(a b c d e))  
; => abcde
```

```
abcde  
; => (a b c d e)
```

```
(set! abcde (cdr abcde))  
; => abcde
```

```
abcde  
; => (b c d e)
```

## Asignación: otro Ejemplo (1/2)

;;; **haga-stack**: es un procedimiento que permite crear un stack  
;;; que tiene las operaciones: **vacio?** , **push!** , **pop!** y **tope!**

```
(define haga-stack
  (lambda ()
    (let ((st '()))
      (lambda (op . args)
        (cond
          ((eqv? op 'vacio?)(null? st))
          ((eqv? op 'push!) (begin (set! st (cons (car args) st))) st)
          ((eqv? op 'pop!)  (begin (set! st (cdr st))) st)
          ((eqv? op 'tope!) (car st))
          (else              "operacion no valida")
        ))))
  ; => haga-stack
```

## Asignación: otro Ejemplo (2/2)

(define st (haga-stack)) => st

(st 'vacio?) => #t

(st 'push! 'perro) => (perro)

(st 'push! 'gato) => (gato perro)

(st 'push! 'canario) => (canario gato perro)

(st 'tope!) => canario

(st 'vacio?) => #f

(st 'pop!) => (gato perro)

## 6.6 Ligado de Variables

# Expresión Lambda

- Permite crear procedimientos, cuyo cuerpo se evalúa secuencialmente.
- En el momento de la evaluación se ligan los parámetros formales a los actuales y las variables libres a sus valores.
- Parámetros formales se especifican en tres formas: lista propia, lista impropia o variable única.

`((lambda (x y) (+ x y)) 3 4)`  $\Rightarrow$  7

`((lambda (x . y) (list x y)) 3 4)`  $\Rightarrow$  (3 (4))

`((lambda x x) 3 4)`  $\Rightarrow$  (3 4)

# Ligado de Referencias a una Variable

- Es un error evaluar una referencia a una variable de nivel superior antes de definirla.
- No lo es que una referencia a una variable aparezca dentro de una expresión no evaluada.

<code>(define x 'a)</code>	<code>=&gt; x</code>
<code>(list x x)</code>	<code>=&gt; (a a)</code>

<code>(let ((x 'b)) (list x x))</code>	<code>=&gt; (b b)</code>
--	--------------------------

<code>(define f (lambda (x) (g x)))</code>	<code>=&gt; f</code>
<code>(define g (lambda (x) (+ x x)))</code>	<code>=&gt; g</code>
<code>(f 3)</code>	<code>=&gt; 6</code>



## Ligado Local: let

- Cada variable se liga al valor correspondiente.
- Las expresiones de valor en la definición están fuera del ámbito de las variables.
- No se asume ningún orden particular de evaluación de las expresiones del cuerpo.
- Se recomienda su uso para valores independientes y donde no importa orden de evaluación.

## Ligado Local: `let`\*

- Similar a `let`, donde se asegura que expresiones se evalúan de izquierda a derecha.
- Cada expresión está dentro del ámbito de las variables de la izquierda.
- Se recomienda su uso si hay una dependencia lineal entre los valores o el orden de evaluación es importante.

## Ligado Local: let\*

```
(let ((x 1) (y 2))  
    (let ((x y) (y x))  
        (list x y)))
```

=> (2 1)

```
(let ((x 1) (y 2))  
    (let* ((x y) (y x))  
        (list x y)))
```

=> (2 2)

## Ligado Local: letrec (1)

- Similar a `let`, excepto que todos los valores están dentro del ámbito de todas las variables (permite definición de procedimientos mutuamente recursivos).
- El orden de evaluación no está especificado.
- Se recomienda su uso si hay una dependencia circular entre las variables y sus valores, y el orden de evaluación no es importante.

## Ligado Local: letrec (2)

- Definiciones son también visibles en los valores de las variables
- Se usa principalmente para definir expresiones lambda
- Existe la restricción que cada valor debe ser evaluable sin necesidad de evaluar otros valores definidos (expresiones lambda lo cumplen)

# Ligado Local: letrec

*;;; letrec hace visible las variables dentro de los valores definidos,  
;;; permitiendo definiciones recursivas con ámbito local*

```
(letrec ((suma (lambda (ls)
                  (if (null? ls)
                      0
                      (+ (car ls) (suma (cdr ls))))
                )
          ))
  (suma '(1 2 3 4 5 6))
)
```

# Ligado Local: letrec

```
(letrec ((suma (lambda (x)
                  (if (zero? x)
                      0
                      (+ x (suma (- x 1))
                          )))))
  (suma 10))
```

=> 55

```
(letrec ((f (lambda () (+ x 2)))
          (x 1))
```

(f))

=> 3

¡Es válido!

```
(letrec ((y (+ x 2))
          (x 1))
```

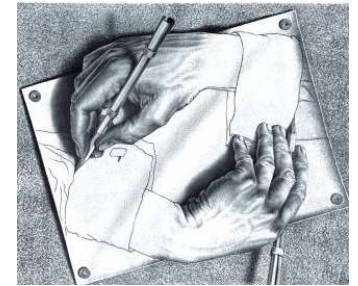
y)

=> error

¡No es válido!

# Ligado Local: letrec

```
(letrec
  (
    (par? (lambda (x)
              (or (= x 0)(impar? (- x 1))))
    )
    (impar? (lambda (x)
              (and (not (= x 0))(par? (- x 1))))
    )
  )
  (list (par? 20) (impar? 20))
)
=> (#t #f)
```



Recursión  
Mutua



# Ligado Local: letrec y let con nombre

- La expresión let con nombre:

```
((lambda(val)
  (let nombre ((var val)) exp1 exp2 ...))
```

- equivale a la expresión letrec:

```
((letrec ((nombre
  (lambda (var) exp1 exp2 ...)))
  nombre
)
val ...)
```

Ejemplo de let con nombre:  
(comparar con ejercicio de ppt 61)

```
((lambda (ls)
  (let suma ((l ls))
    (if (null? l)
        0
        (+ (car l) (suma (cdr l)))
    )
  )
)
'(1 2 3 4 5 6)
)
```

=> 21

## 6.7 Otras operaciones en Scheme

# Igualdad - Equivalencia

- $(eq? \text{ } obj_1 \text{ } obj_2)$   
*retorno: #t si son idénticos*
- $(eqv? \text{ } obj_1 \text{ } obj_2)$   
*retorno: #t si son equivalentes*
- $(equal? \text{ } obj_1 \text{ } obj_2)$   
*retorno: #t si tienen la misma estructura y contenido*
- $eqv?$  es similar a  $eq?$ , salvo que no es dependiente de la implementación, pero es algo más costoso.
- $eq?$  no permite comparar en forma fiable números.
- $equal?$  es similar a  $eqv?$ , salvo que se aplica también para strings, pares y vectores.

# Igualdad - Equivalencia: ejemplos

(eq? 'a 'a) => #t

(eq? 3.1 3.1) => #f

(eq? (cons 'a 'b) (cons 'a 'b)) => #f

(eqv? 'a 'a) => #t

(eqv? 3.1 3.1) => #t

(eqv? (cons 'a 'b) (cons 'a 'b)) => #f

(equal? 'a 'a) => #t

(equal? 3.1 3.1) => #t

(equal? (cons 'a 'b) (cons 'a 'b)) => #t

# Listas asociativas

- Una lista asociativa es una lista propia cuyos elementos son pares, donde cada tiene la forma (*clave valor*).
- Las asociaciones son útiles para almacenar información (*valor*) relacionada con un objeto (*clave*).

(assq *obj alist*)

retorno: *primer elemento de alist cuyo car es equivalente a obj, sino #f*

(assv *obj alist*)

*ídem*

(assoc *obj alist*)

*ídem*

# Listas asociativas: ejemplo

```
(define e '((a 1) (b 2) (c 3)))
```

```
(assq 'a e) ⇒ (a 1)
```

```
(assq 'b e) ⇒ (b 2)
```

```
(assq 'd e) ⇒ #f
```

# eval

`(eval obj)`

retorno: evaluación de *obj* como programa Scheme

- *obj* debe ser un programa válido de Scheme.
- El ámbito actual no es visible a *obj*, comportándose éste como si estuviera en un nivel superior de otro ambiente.
- No pertenece al estándar de ANSI/IEEE.

`(eval 3)`                       $\Rightarrow 3$

`(eval '(+ 3 4))`               $\Rightarrow 7$

`(eval (list '+ 3 4))`         $\Rightarrow 7$





# apply

`(apply proc obj... lista)`

retorno: resultado de aplicar *proc* a los valores de *obj...* y a los elementos de la *lista*

- `apply` invoca *proc* con *obj* como primer argumento ..., y los elementos de lista como el resto de los argumentos.
- Es útil cuando algunos o todos los argumentos de un procedimiento están en una lista.

```
(apply + `(5 -1 3 5))           => 12
```

```
(apply min 5 1 3 `(5 -1 3 5))  => -1
```

# map

```
(map proc lista1 lista2 ...)  
retorno: lista de resultados
```

- Las listas deben ser del mismo largo.
- *proc* debe aceptar un número de argumentos igual al número de listas.
- *map* aplica repetitivamente *proc* tomando como parámetros un elemento de cada lista.

```
(map (lambda (x y) (sqrt (+(* x x) (* y y))))  
      '(3 5)  
      '(4 12))
```

```
=> (5 13)
```

# do

```
(do ((var val nuevo) ...)
     (test res ...) exp...)
retorno: valor de último res
```

- Permite una forma iterativa simple.
- Las variables *var* se ligan inicialmente a *val*, y son re-ligadas a *nuevo* en cada iteración posterior.
- En cada paso se evalúa *test*.
  - *#t*: se termina evaluando en secuencia *res* ... y retornando valor de última expresión de *res* ... .
  - *#f*: se evalúa en secuencia *exp* ... y se vuelve a iterar re-ligando variables a nuevos valores.

## do: ejemplo

```
(do ((var val nuevo) ...)
    (test res ...) exp...)
retorno: valor de último res
```

```
(define factorial
  (lambda (n)
    (do ((i n (- i 1))           ;; variable i
        (a 1 (* a i)))         ;; variable a
        ((zero? i) a))))       ;; test
```

# filter

```
(filter proc lista)  
retorno: lista de resultados
```

- Retorna una lista con los elementos que cumplan con el predicado *proc*.
- *proc* es un procedimiento que recibe un solo argumento. Si retorna verdadero, el elemento se mantiene en la lista. Si retorna falso, se elimina.

```
(filter odd? '(1 2 3 4 5 6))
```

```
=> (1 3 5)
```

# Evaluación Perezosa

(delay exp)

retorno: una promesa

(force promesa)

retorno: resultado de forzar la promesa

- *delay* con *force* se usan juntos para permitir una evaluación perezosa, ahorrando computación.
- La primera vez que se fuerza la promesa se evalúa la expresión *exp*, memorizando su valor; forzados posteriores retornan el valor memorizado.

# Evaluación Perezosa: ejemplo

```
(define stream-car          ;; define un stream infinito de números naturales  
  (lambda (s) (car (force s))))
```

```
(define stream-cdr  
  (lambda (s) (cdr (force s))))
```

```
(define contadores  
  (let prox ((n 1))  
    (delay (cons n (prox (+ n 1))))))
```

```
(stream-car contadores)
```

```
=> 1
```

```
(stream-car (stream-cdr contadores))
```

```
=> 2
```

# Unidad 6

## Programación Funcional y Scheme

FIN

- 6.1 Introducción a la Programación Funcional
- 6.2 Introducción al Lenguaje de Programación Scheme
- 6.3 Condicionales
- 6.4 Recursión
- 6.5 Asignación
- 6.6 Ligado de Variables
- 6.7 Otras operaciones en Scheme