LENGUAJES DE PROGRAMACIÓN

Equipo de Profesores:
Jorge Díaz Matte - José Luis Martí Lara
Wladimir Ormazabal Orellana

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.

Programación Funcional

- La programación funcional pura **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.

Funciones Matemáticas

- Un función es una proyección de un conjunto dominio a otro que es el rango: f: D → 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: cubo(x) ≡ x*x*x, con x real
- Aplicación de la función: cubo(2.0) ⟶ 8.0

NOTACIÓN LAMBDA DE CHURCH

- Separa definición de la función de su nombre
- Definición de una función: λ(x) x*x*x
- Aplicación de la función: (λ(x) x*x*x) (2.0) ⟶ 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: α (f, (x, y, z)) produce (f(x), f(y), f(z))

Principios claves de la Programación Funcional

- Las funciones son el elemento básico de abstracción y reutilización de código. Los programas se construyen mediante una composición de funciones.

- Las funciones son un elemento de primer orden en el lenguaje. Pueden guardarse en estructuras de datos, pasarse como argumentos y devolverse desde otras funciones.

- Las funciones deben ser puras, es decir, siempre devolver las mismas salidas para las mismas entradas y no tener efectos secundarios.
  - Son útiles porque se pueden ejecutar en cualquier orden y en paralelo, cuando no hay dependencias de parámetros entre ellas.

Principios claves de la Programación Funcional

- Programar considerando objetos inmutables, siendo la inmutabilidad la propiedad de un objeto de no cambiar su estado.
  - La inmutabilidad libera de pensar en los cambios sufridos por los objetos a lo largo de la ejecución de un programa.
  - Los objetos inmutables son automáticamente seguros en hilos (*thread-safe*), ya que pueden ser accedidos de manera concurrente sin consecuencias, al no poder modificarse.

- A través del código se debe asegurar la transparencia referencial, es decir, que cualquier expresión se pueda sustituir por su valor sin que esto altere el comportamiento del programa → tener funciones puras y objetos inmutables.

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.
- Funciones son entidades de primera clase, y por lo tanto se tratan como cualquier valor.
- Tiene recolección automática de basura.

- En estricto rigor, es un lenguaje de programación funcional “impuro”, pues sus estructuras de datos no son inmutables.

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 pronto, 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)                    => 1
(- 2 (* 4 1/3))                => 2/3
(/ (* 6/7 7/2) (- 4.5 1.5))    => 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):

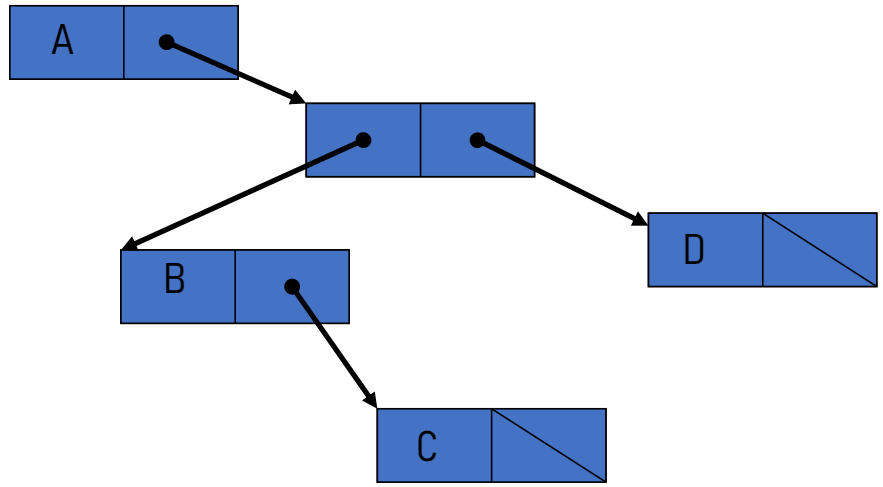
  (cdr '(a b c d)) => (b c d)

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

Scheme: Listas

Estructura de una Lista

(A (B C) D)



Observación: cada nodo es un “par”

Scheme: Listas

Constructores:

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

```
(cons ´a ´(b c d))                  => (a b c d)
(cons (car ´(a b c))(cdr ´(a b c))  => (a b c)
```

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

```
(list ´a ´b ´c ´d)                  => (a b c d)
(list)                              => ()
```

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

```
(append ´(a b) ´(c d))              => (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) => 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** ((var$_1$ val$_1$) … (var$_m$ val$_m$)) exp$_1$ … exp$_n$)

equivale a:

((**lambda** (var$_1$ … var$_m$) exp1 … exp$_n$) val$_1$ … val$_m$)

Scheme: especificación de parámetros formales

- Lista propia de parámetros (var_1 var_2 ... var_n)

  ((lambda (x y) (list x y)) 1 2) => (1 2)

- Lista impropia de parámetros (var_1 var_2 ... var_n . var_r)

  ((lambda (x . y) (list x y)) 1 2 3) => (1 (2 3))

- Parámetros único var_r

  ((lambda x (list x)) 1 2) => ((1 2))

Scheme: ámbito

- Las variables definidas con let y lambda son sólo visibles en el cuerpo de las expresiones (ámbito local).

- El procedimiento define permite definir variables de nivel superior (ámbito 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 (1)

- Uso de **define**:

```
(define pi 3.1416)                          => pi

(define square (lambda (x) (* x x))))      => square

(square 3)                                  => 9

(let ((x 2)(square 4)) (* x square))        => 8
```

Scheme: definiciones de nivel superior (2)

- La forma: (**define** var_0 (**lambda** (var_1 ... var_n) e_1 ...))
  se puede abreviar como: (**define** (var_0 var_1 ... var_n) e_1 ...)

- **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 exp_n)
)
```

- 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 lógicos: or, and y not

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

Predicados (2)

- Lista nula: null? (null? ´()) => #t
- Argumentos equivalentes: eqv? (eqv? ´a ´a) => #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, es decir si una lista
  - 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) * ... 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 lista '(a b c d e))
; => lista

lista
; => (a b c d e)


(set! lista (cdr lista))
; => lista

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

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)          => 7

((lambda (x . y) (list x y)) 3 4)     => (3 (4))

((lambda x x) 3 4)                    => (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.

```
(define x 'a)                          => x
(list x x)                             => (a a)

(let ((x 'b)) (list x x))              => (b b)

(define f (lambda (x) (g x)))          => f
(define g (lambda (x) (+ x x)))        => g
(f 3)                                  => 6
```

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

Ámbito de **suma**

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

suma es visible en el interior de la lambda

Ámbito de **suma**

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

suma no es visible en el interior de la lambda

Ligado Local: letrec

- 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 ...)
```

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

=> 21
```

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))
```

¡Es válido!

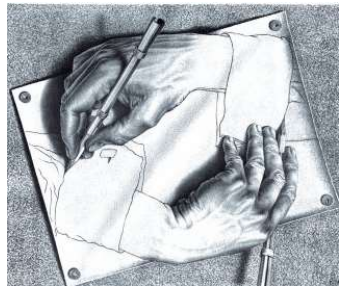
=> 3

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

¡No es válido!

=> error

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

6.7 Otras operaciones en Scheme

Igualdad - Equivalencia

- (eq? obj_1 obj_2)
  retorno: #t *si son idénticos*
- (eqv? obj_1 obj_2)
  retorno: #t *si son equivalentes*
- (equal? obj_1 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) => #t

(**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)              => 3

(eval '(+ 3 4))       => 7

(eval (list '+ 3 4))  => 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* $lista_1$ $lista_2$ ...)
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)
```

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
```

do: evitar!!

(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
```

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