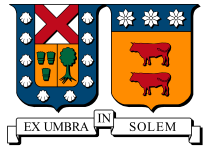


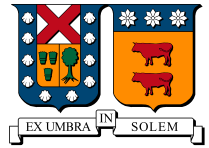
Programación Funcional y Lenguaje Scheme

Roberto Díaz

roberto.diazu@usm.cl

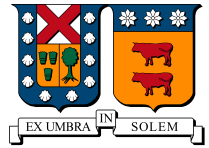
Universidad Técnica Federico Santa María
Departamento de Informática – Santiago, Chile

2021-2



Introducción a la Programación Funcional

Programación Funcional



Paradigma diferente a los imperativos, que se aleja de la máquina de von Neumann.

Basado en funciones matemática (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 de **dominio** a otro que es el **rango**

$$f : D \rightarrow R$$

La evaluación de funciones está controlada por recursión y condiciones (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 Simples

Matemáticas:

- Definición de una función:

$$cubo(x) \equiv x * x * x \text{ con } x \in \mathbb{R}$$

- Aplicación de la función:

$$cubo(2.0) \rightarrow 8.0$$

La **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:

$$\{\lambda(x)\ x * x * x\}(2.0) \rightarrow 8.0$$

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

- **Composición de Funciones**

$$h \equiv f \circ g \text{ entonces } h(x) \equiv f(g(x))$$

- **Aplicación a todo**: una misma función se aplica a una lista de argumentos

$$\alpha(f,(x,y,z)) \text{ produce } (f(x),f(y),f(z))$$

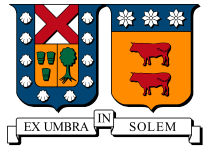
- La Programación Funcional pura no usa variables ni asignación
- Repetición debe ser lograda con recursión.
- Un **programa** consiste en la definición de funciones y la aplicación de éstas.
- La **ejecución del programa** no es nada más que la evaluación de funciones.
- La **transparencia referencial** 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.



Introducción al Lenguaje Scheme

Scheme
Origen

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
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.

Ambiente Interactivo

Corresponde al ciclo: **leer**, **evaluar** e **imprimir** (denominado **REPL**).

El sistema entrega una línea de comandos, se ingresa la expresión, el sistema evalúa y entrega el resultado.
Ejemplo:

```
"Hola Scheme"
  => "Hola Scheme"
```

Toda constante evalúa en la misma constante.

Es posible cargar y guardar en un archivo para facilitar el proceso de desarrollo.

Ambiente a Usar: Dr. Racket (`racket-lang.org`)

Identificadores

Corresponden a **palabras claves**, **variables** y **símbolos**. Se forman de:

- Mayúsculas y Minúsculas
  `['A'...'Z','a'...'z']`
- Dígitos
  `[0 ... 9]`
- Carácteres
  `[? ! . + - * / < = > : $ % ^ & _ ~]`

Constantes Básicas

- **String**: se escribe usando citado doble

```
"Un string es sensible a Mayusculas"
```

- **Carácter**: precede de #\

```
#\a
```

- **Número**: pueden ser enteros, fraccionarios, punto flotante y en notación científica

```
-365
1/4
23.46
1.3e27
```

- **Números complejos**: en coordenadas rectangulares o polares

```
2.7-4.5i
+3.4@-0.5
```

- **Booleanos**: son los valores `#f` (falso) y `#t` (verdadero)

Sintaxis

La sintaxis de Scheme consiste en listas de símbolos que van entre paréntesis redondos

```
(a b c d)
```

Estas 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
```

Funciones Aritméticas

Los nombres +, −, ∗ y / son reservados para las operaciones aritméticas.

Funciones se escriben como listas en notación prefija:

```
(+ 1/2 1/2)
  =>
(- 2 (* 4 1/3))
  =>
(/ (* 6/7 7/2) (- 4.5 1.5))
  =>
```

Funciones Aritméticas

Los nombres +, −, ∗ y / son 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
```

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` como nombre de función.

Listas
Operadores Básicos

car devuelve el primer elemento de la lista

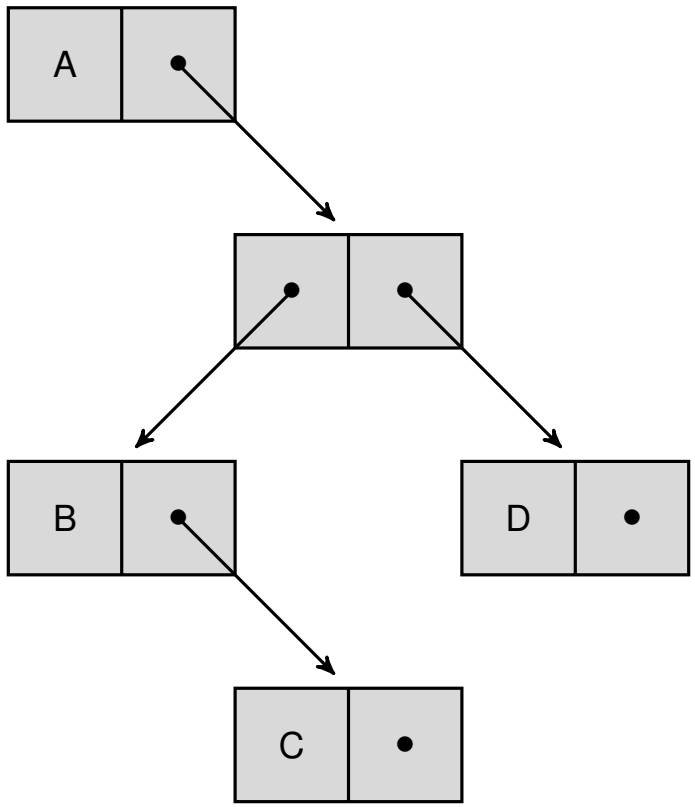
```
(car '(a b c d))
  => a
```

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

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

Listas
Estructura

```
(a (b c) d)
```



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

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

Variables y Expresiones `let`

`let` permite definir variables que se ligan a un valor en la evaluación de expresiones. Variables son sólo visibles dentro del cuerpo de `let` (ámbito local)

Sintaxis:

```
(let ((var1 val1) (var2 val2) ... ) exp1 exp2 ... )
```

Ejemplo:

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

Expresiones `lambda`

Permite crear un nuevo procedimiento. Una expresión lambda es un objeto tipo procedimiento que no tiene nombre.

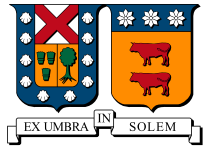
Sintaxis:

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

Ejemplo:

```
(lambda (x) (* x x))
  => #<procedure>
((lambda (x) (* x x)) 3)
  => 9
```

Expresiones `lambda`
Ejemplo




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

Relación entre `let` y `lambda`

Nótese que:

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

equivale a:

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

Especificación de Parámetros Formales

Lista propia de parámetros `(var1 var2 ... varn)`

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

Parámetro único `varu`

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

Lista impropia de parámetros `(var1 var2 ... varn . varr)`

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

Definiciones de nivel superior

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
Ejemplo: una variable definida con el mismo nombre mediante `let` oculta a las de nivel superior

Definiciones de nivel superior
Ejemplo

```
(define pi 3.1416)
  => pi
pi
  => 3.1416
(define square (lambda (x) (* x x)))
  => square
(square 3)
  => 9
(let ((x 2)(square 4)) (* x square))
  => 8
```

Definición de funciones
Abreviación

La forma:

```
(define var0 (lambda (var1 ... varn ) exp1 exp2 ...))
```

Se puede abreviar como:

```
(define (var0 var1 ... varn ) exp1 exp2 ...)
```

Ejemplo:

```
(define square (lambda (x) (* x x))))
```

Es equivalente a:

```
(define (square x) (* x x))
```

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
    (- n)
  )
)

(abs -27)
  => 27
```

Predicados

Procedimientos para **expresiones relacionales**

```
(= 3 4)
  => #f
(< 3 4)
  => #t
(> 3 4)
  => #f
(<= 3 4)
  => #t
(>= 3 4)
  => #f
```

Procedimientos para **expresiones lógicas**

```
(not (< 3 4))
  => #f
(and (> 5 2) (< 10 5))
  => #f
(or (> 5 2) (< 10 5) )
  => #t
```

Otros predicados

Lista nula o vacía: `null?`

```
(null? '())
  =>#t
```

Prueba de tipo:

```
(complex? 3+4i)
  =>#t
(real? 3)
  =>#t
(rational? 6/10)
  =>#t
(integer? 3)
  =>#t
(char? #\a)
  =>#t
(string? "hola mundo")
  =>#t
(list? '(2 3 4))
  =>#t
(pair? '(2 3))
  =>#t
```

Otros predicados

Equivalencia:

- = prueba si dos o más **números** son equivalentes
- `eq?` prueba solo si dos argumentos son el mismo objeto en memoria
- `eqv?` prueba además si dos argumentos tienen el mismo valor
- `equal?` prueba además si son iguales cada elemento de dos listas con `eqv?`

```
(define x '(2 3))
(define y '(2 3))
(eq? x y)
  => #f
(equal? x y)
  => #t
(define y x)
(eq? x y)
  => #t
(define y '(3 2))
(equal? x y)
  => #f
```

Expresión condicional múltiple

Scheme provee expresiones que se evalúan condicionalmente usando **`cond`**.

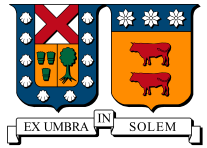
Sintaxis:

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

El uso de `else` es opcional, siendo equivalente su uso a colocar `#t`

Expresión condicional múltiple
Ejemplo

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



Recursión en Scheme

Recursión

Una **función recursiva** es aquella que se invoca a sí misma.

Función que realiza la recursión:

- **Directa**: la función se invoca a sí misma.
- **Indirecta**: la función invoca a otra función, y posiblemente ésta a otras, que terminan invocando a la primera.

Cantidad de invocaciones recursivas:

- **Lineal**: existe una única invocación recursiva.
- **Múltiple**: existe más de una invocación recursiva.
- **Anidada**: dentro de una invocación recursiva, se pasa como parámetro otra invocación recursiva.

Recursión

Posición de la invocación recursiva:

- **de Cabeza**: la invocación recursiva se hace al principio, antes que el resto de las sentencias de proceso (puede haber una condicional antes)
- **Intermedia**: otras sentencias aparecen antes y después de la invocación recursiva
- **de Cola**: la invocación recursiva se hace al final, después que el resto de las sentencias

Recursión Directa
Ejemplo

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

(length '(a b c d))
```

Recursión Directa
Ejemplo

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

(length '(a b c d))
=> (+ 1 (length '(b c d)))
```

Recursión Directa
Ejemplo

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

(length '(a b c d))
=> (+ 1 (length '(b c d)))
       => (+ 1 (length '(c d)))
```

Recursión Directa
Ejemplo

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

(length '(a b c d))
=> (+ 1 (length '(b c d)))
       => (+ 1 (length '(c d)))
              => (+ 1 (length '(d)))
```

Recursión Directa
Ejemplo

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

(length '(a b c d))
=> (+ 1 (length '(b c d)))
       => (+ 1 (length '(c d)))
              => (+ 1 (length '(d)))
                     => (+ 1 (length '()))
```

Recursión Directa
Ejemplo

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

(length '(a b c d))
=> (+ 1 (length '(b c d)))
       => (+ 1 (length '(c d)))
              => (+ 1 (length '(d)))
                     => (+ 1 (length '()))
                            => 0
```

Recursión Directa
Ejemplo

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

(length '(a b c d))
=> (+ 1 (length '(b c d)))
       => (+ 1 (length '(c d)))
              => (+ 1 (length '(d)))
                     => (+ 1 (length '()))
                            => 0
```

Recursión Directa
Ejemplo

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

(length '(a b c d)) => 4
=> (+ 1 (length '(b c d))) => 4
        => (+ 1 (length '(c d))) => 3
                => (+ 1 (length '(d))) => 2
                        => (+ 1 (length '())) => 1
                                => 0 => 0
```

Recursión Directa
Ejemplo procedimiento memv

```
;; El siguiente procedimiento busca x en la lista ls,
    devuelve el resto de la lista despues 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
Ejemplo procedimiento `remv`

```
;; 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

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)

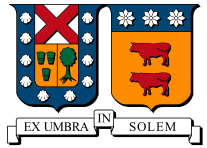
Recursión de cola es cuando un procedimiento hace un llamado de cola hacia si mismo, o indirectamente a través de una serie de llamados de cola hacia si mismo.

Recursión de Cola
Ejemplo

Los siguientes ejemplos son llamados de cola a f, pero no a g

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

(lambda () (or (g) (f)))
```

Propiedad de la recursión de cola



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
Ejemplo Factorial

Ejemplo sin recursión de cola

```
(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))
        )
    )
  )
  => factorial

(factorial 5)
  => 120
```

Ejemplo con recursión de cola

```
(define factorial1
  (lambda (n)
    (let fact ((i n) (a 1))
      (if (= i 0)
        a
        (fact (- i 1) (* a i))
      )
    )
  )
)
  => factorial1

(factorial1 5)
  => 120
```

Ejemplo sin recursión de cola

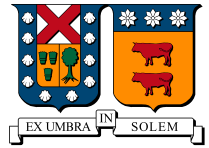
```
(define fibonacci
  (lambda (n)
    (let fib ((i n))
      (cond ((= i 0) 0)
        ((= i 1) 1)
        (else (+ (fib (- i 1)) (fib (- i 2))))
      )
    )
  )
)
  => fibonacci

(fibonacci 20)
  => 6765
```

Ejemplo con 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)
        )
      )
    )
  )
)
  => fibonacci1

  (fibonacci 20)
    => 6765
```

Recursión de Cola
Ejemplo Factorizar

```
(define factorizar
  (lambda (n)
    (let fact ((n n) (i 2))
      (cond
        ((> i n) '())
        ;;No recursiva de cola
        ((integer? (/ n i)) (cons i (fact (/ n i) i)))
        ;;Recursiva de cola
        (else (fact n (+ i 1)))
))))
  => factorizar

(factorizar 120)
  =>  (2 2 2 3 5)
(factorizar 37897)
  => (37897)
(factorizar 1)
  => ()
```



Asignación

Asignación

`let` permite ligar un valor a una (nueva) variable en su cuerpo (local), como `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.

`set!` permite en Scheme re-ligar a una variable existente un nuevo valor, como lo haría una asignación. `set!` no establece un nuevo ligado, sino que cambia uno existente.

Asignación
Ejemplo

```
(define a 0)
a
  => 0

(set! a 1)
a
  => 1
```

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

(set! abcde (cdr abcde))
abcde
  => (b c d e)
```

Asignación
Ejemplo Contador

```
;; Definicion de un contador usando variable de nivel
    superior
(define contador 0)
  => contador
(define cuenta
  (lambda ()
    (set! contador (+ contador 1))
    contador
  )
)
  => cuenta

(cuenta)
  => 1
(cuenta)
  => 2
```

Asignación
Ejemplo Contador

```
;; La siguiente solucion usando let define una variable que
    no es visible fuera de su definicion
(define cuenta1
  (let ((cont 0))
    (lambda ()
      (set! cont (+ cont 1))
      cont
    )
  )
)
  => cuenta1

(cuenta1)
  => 1
(cuenta1)
  => 2
```

Asignación
Ejemplo Contador

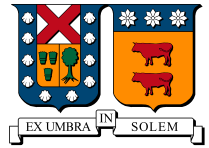
```
(define hacer-contador
  (lambda ()
    (let ((cont 0))
      (lambda ()
        (set! cont (+ cont 1))
        cont
))))
  => hacer-contador

(define cont1 (hacer-contador)) => cont1
(define cont2 (hacer-contador)) => cont2

(cont1)
  => 1
(cont2)
  => 1
(cont1)
  => 2
(cont1)
  => 3
(cont2)
  => 2
```

Asignación
Ejemplo Stack

```
;;; 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
```

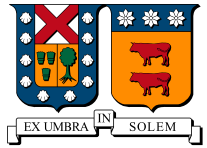
Asignación
Ejemplo Stack

```
(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)
```



Ligado de Variables en Scheme

Referencia a una Variable



Cualquier identificador no citado en una expresión es una **palabra clave** o una **referencia 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.

Referencia a una Variable
Ejemplo

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

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

;; g no ha sido definida
(define f (lambda (x) (g x)))
  => f
;; definicion de g
(define g (lambda (x) (+ x x)))
  => g
(f 3)
  => 6
```

Lambda

```
(lambda formales exp1 exp2 ...)
  => retorno: un procedimiento
```

Permite crear procedimientos. 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
- Variable única

Lambda
Ejemplo

Lista propia

```
((lambda (x y) (+ x y)) 3 4)
  => 7
```

Lista impropia

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

Variable única

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

Ligado Local

Formas más conocidas de ligado local son:

- `let`
- `let*`
- `letrec`

Ligado local
let

```
(let ((var val)...) exp1 exp2 ...)
  retorno: valor de ultima expresion
```

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` y Definición de Variables

Procedimientos recursivos se han definido con variables de nivel superior.

Hasta ahora se ha visto una forma let para definir variables locales, sólo visibles en el cuerpo, por tanto no son visibles por las variables.

Ejemplo:

```
;;;suma no es visible dentro de la expresion lambda
  (let
    ((suma
      (lambda (ls)
        (if (null? ls)
          0
          (+ (car ls) (suma (cdr ls)))
        )
      )
    ))
    (suma '(1 2 3 4 5 6))
  )
```

Ligado Local
Recursión con `let`

```
;;; se puede solucionar el problema entregando suma como
    argumento a la expresion lambda, siendo la solucion mas
    complicada
(let
  ((suma
    (lambda (suma ls)
      (if (null? ls)
      0
      (+ (car ls) (suma suma (cdr ls)))
      )
    )
  ))
  (suma suma '(1 2 3 4 5 6))
)
  => 21
```

Ligado local
let*

```
(let* ((var val) ...) exp1 exp2 ...)
  => retorno: valor de ultima expresion
```

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
letrec

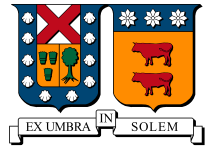
```
(letrec ((var val) ...) exp1 exp2 ...)
  => retorno: valor de ultima expresion
```

Similar a `let` excepto que todos los valores están dentro del ámbito de todas las variables (permite definición de procedimientos mutuamente recursivos).

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

Definición `letrec`



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
Ejemplo

```
(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)

(letrec ((suma
          (lambda (ls)
            (if (null? ls)
                0
                (+ (car ls) (suma (cdr ls)))
                )
            ))
  (suma '(1 2 3 4 5 6))
  )
=> 21
```

Ligado local
Ejemplo `letrec`

El siguiente código es válido

```
(letrec ( (f (lambda () (+ x 2)))
          (x 1)
        )
(f))
=> 3
```

El siguiente código no es válido

```
(letrec ( (y (+ x 2))
          (x 1)
    )
  y)
=> error
```

Recursión Mutua
Ejemplo

```
(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)
```

Equivalencia entre `letrec` y `let` con nombre

La expresión `let` con nombre

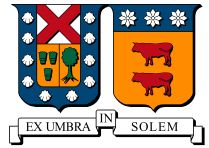
```
(let nombre ((var val) ...) exp1 exp2 ...)
```

Equivale a la expresión `letrec`

```
((letrec (
    (nombre
      (lambda (var ...) exp1 exp2 ...)
    ))
  nombre ;; fin de cuerpo de letrec
  )
val ...)
```

Definición `let` con nombre

```
(let suma ((ls '(1 2 3 4 5 6)))
  (if (null? ls)
    0
    (+ (car ls) (suma (cdr ls))))
  )
)
=> 21
```



Otras Operaciones en Scheme

Evaluación

```
(eval obj)
=> retorno: evaluacion 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.

Evaluación
Ejemplo

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

do

```
(do ((var val nuevo) ...)
  (test res ...) exp ...)
=> retorno: valor de ultimo 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

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

```
(define divisores
  (lambda (n)
    (do ((i 2 (+ i 1))                   ;;variable i
      (ls '()                            ;;variable ls
        (if (integer? (/ n i))
          (cons i ls)
          ls)))
      ((>= i n) ls))))                   ;;test
```

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.

La `lista` es obligatoria, pero puede ser la lista vacía.

apply
Ejemplo

```
(+ 3 4)
=> 7
((lambda (x) x) 5)
=> 5
(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
```

`proc` debe aceptar un número de argumentos igual al número de listas.

`map` aplica repetitivamente `proc` tomando como parámetros un elemento de cada lista.

Las listas deben ser del mismo largo.

map
Ejemplo

```
(map (lambda (x) (* x x))
   '(1 2 3 4)
)
   => (1 4 9 16)
```

```
(map (lambda (x y) (sqrt (+ (* x x) (* y y))))
   '(3 5)
   '(4 12)
)
   => (5 13)
```

filter

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

`filter` 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.

Ejemplo:

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

Evaluación Perezosa

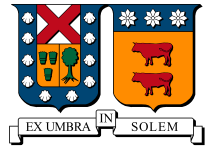
```
(delay exp)
=> retorno: una promesa

(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.

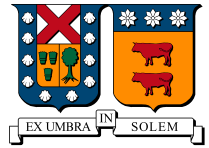
Evaluación Perezosa
Ejemplo

```
;;; define un stream infinito de numeros naturales
(define stream-car
  (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
```



Otros Objetos en Scheme

Listas Asociativas



Una lista asociativa (`alist`) es una lista propia cuyos elementos son pares de la forma `(clave valor)`.

Son útiles para almacenar información (valor) relacionada con un objeto (clave).

Listas Asociativas
Operadores

```
(assq obj alist)
(assv obj alist)
(assoc obj alist)
```

Los operadores retornan el primer elemento de `alist` cuyo `car` es equivalente a `obj`, sino `#f`. Usan `eq?`, `eqv?` y `equal?` respectivamente.

Listas Asociativas
Ejemplo

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

(assv 'a e)
  => (a 1)
(assv 'b e)
  => (b 2)
(assv 'd e)
  => #f
```

Vectores

Forma más conveniente y eficiente de listas en algunas aplicaciones.

A diferencia de listas, el acceso a un elemento se hace en tiempo constante.

Al igual que strings, se referencian con base 0.

Elementos pueden ser de cualquier tipo.

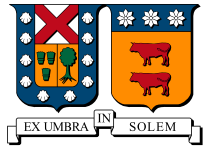
Se escriben como lista, pero precedidos por #.

```
(vector a b c)
```

Vectores
Operadores

```
(vector obj ...)
  => retorno: un vector con objetos obj ...
(make-vector n)
  => retorno: un vector de largo n
(vector-length vector)
  => retorno: largo de vector
(vector-ref vector n)
  => retorno: el elemento n de vector
(vector-set! vector n obj)
  => sin retorno: cambia el elemento n por obj en vector
```

Referencias



- Capítulo XV de [Sebesta, 2011]