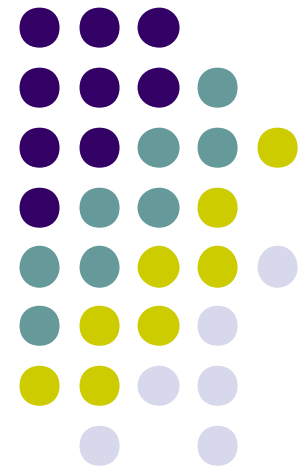


4.5 Ligado de Variables en Scheme

Referencia a variables, lambda,
formas let y asignación





a) Referencia a una Variable

✍ ***variable***

retorno: valor de la *variable*

✍ Cualquier identificador no citada en una expresión es una *palabra clave* o una *referencia a una variable*.

✍ Una *palabra clave* tiene un ligado léxico; sino es una *referencia a una variable*.



Ligado de Referencias a Variables

- ✍ 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.

Ejemplo de Referencias a Variables



```
list
```

```
=> #<procedure>
```

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

```
=> x  
=> (a a)
```

```
(let ((x 'b)) (list x x))
```

```
=> (b b)
```

```
(define f (lambda (x) (g x)))  
(define g (lambda (x) (+ x x)))  
(f 3)
```

```
=> f  
=> g  
=> 6
```



b) 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 o impropia o variables única
- ✍ Cuerpo se evalúa secuencialmente.

Ejemplo de Lambda



`((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)



c) Ligado Local

✍ Formas más conocidas de ligado local son:

✍ let

✍ let*

✍ letrec



Forma `let`

- ✍ `(let ((var val)...) exp1 exp2 ...)`
retorno: valor de última expresión
- ✍ 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.



Forma `let*`

- ✍ `(let* ((var val)...) exp1 exp2 ...)`
retorno: valor de última expresión
- ✍ 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.



Forma `letrec`

- ✍ `(letrec ((var val)...) exp1 exp2 ...)`
retorno: valor de última expresión
- ✍ 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.



Ejemplo de ligado local

```
(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 (x)  
                  (if (zero? x)  
                      0  
                      (+ x (suma (- x 1)))))))  
  (suma 10))  
                                     ;=> 55
```



d) Definición de Variables

- ✍ `(define var exp)`
retorno: *no especificado*
- ✍ Normalmente se usan como **definiciones de nivel superior**, i.e. fuera de cualquier forma `let` o `lambda`, siendo en este caso visible en cualquier expresión donde no sea sombreada por una variable local.
- ✍ También están permitidas la **definiciones internas** al comienzo del cuerpo de una expresión `lambda` o derivada.



Definiciones Internas

✍ La expresión:

```
(lambda formales  
  (define var val) ...  
  exp1 exp2 ...)
```

✍ Se transforma en:

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

Definiciones Abreviadas para Procedimientos



✍ **(define** (var₀ var₁ ...) exp₁ exp₂ ...) **retorno:** *no especificado*

✍ **(define** (var₀ . var_r) exp₁ exp₂ ...) **retorno:** *no especificado*

✍ **(define** (var₀ var₁ var₂ ... var_n .var_r)
exp₁ exp₂ ...) **retorno:** *no especificado*



Ejemplo de Definiciones

```
(define x 3)  
x          => 3
```

```
(define f (lambda (x) (+ x 1)))  
          => f
```

```
(let ((x 2))  
  (define f (lambda (y) (+ y x)))  
  (f 3))  
          => 5
```

```
(f 3)      => 4
```

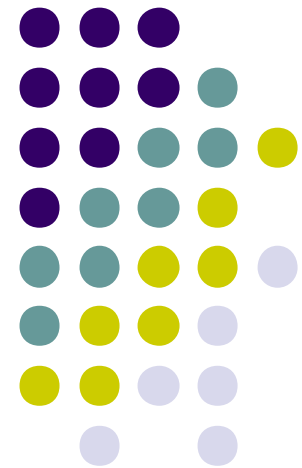


e) Asignación

- ✍ `(set! var exp)`
retorno: *no especificado*
- ✍ Cambia el valor ligado a la variable se cambia al valor de la expresión.
- ✍ Evaluaciones posteriores evalúan al nuevo valor.
- ✍ **set!** no establece un nuevo ligado, sino que cambia uno existente.
- ✍ Son útiles para actualizar un estado y para crear estructuras recursivas.

4.6 Operaciones de Control en Scheme

Constantes, citaciones,
secuenciación, condicionales y
formas repetitivas





a) Constantes

✍ ***constante***

retorno: *constante*

- ✍ Cualquier constante evalúa hacia si mismo.
- ✍ Son constantes los números, booleanos, caracteres y strings.
- ✍ Las constantes son inmutables (no se pueden cambiar con asignación)



b) Citaciones

✍ `(quote obj)` (ó ``obj`)

retorno: `obj`

✍ ``obj` es equivalente a `(quote obj)`

✍ `Quote` inhibe la evaluación de `obj`

✍ Citación no es necesaria para una constante



Cuasi- Citación

✍ (quasiquote *obj*) (ó ``obj`)
retorno: *ver explicación*

✍ (unquote *obj*) (ó `,obj`)
retorno: *ver explicación*

✍ (unquote-splicing *obj*) (ó `,@obj`)
retorno: *ver explicación*

Splice ✍ empalmar



Expresiones Cuasi-citadas

- ✍ `quasiquote` es similar a `quote`, salvo que permite descitar parte del texto (usando `unquote` o `unquote-splicing`).
- ✍ Dentro de una expresión cuasi-citada se permite la evaluación de expresiones que han sido descitadas, cuyo resultado es el que aparece en el texto.
- ✍ `unquote` o `unquote-splicing` son sólo válidos dentro de un `quasiquote`.

Ejemplos de Citaciones



```
`(* 2 3)
```

```
=> (+ 2 3)
```

```
`(+ 2 3)
```

```
=> (+ 2 3)
```

```
`((+ 2 ,(* 3 4))
```

```
=> (+ 2 12)
```

```
(let ((a 1) (b 2))
```

```
  `(,a . ,b))
```

```
=> (1 . 2)
```

```
`(list ,(list 1 2 3))
```

```
=> (list (1 2 3))
```

```
`(list ,@(list 1 2 3))
```

```
=> (list 1 2 3)
```

```
`',(cons `a `b)
```

```
=> `,(cons `a `b)
```

```
`',(cons `a `b)
```

```
=> `(a . b)
```



c) Aplicación de Procedimientos

✍ **(*proc exp ...*)**

retorno: *resultado de aplicar el valor de proc a los valores de exp ...*

- ✍ Aplicación de procedimientos o funciones es la estructura básica de Scheme
- ✍ Orden de evaluación de expresiones no está especificado, pero se hace secuencialmente.



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.

Ejemplos de Aplicación de Procedimientos



`(+ 3 4)` \Rightarrow 7

`((lambda (x) x) 5)` \Rightarrow 5

`(apply + '(5 -1 3 5))` \Rightarrow 12

`(apply min 5 1 3 '(5 -1 3 5))` \Rightarrow -1



d) Secuenciación

- ✍ **(begin *exp1 exp2 ...*)**
retorno: *resultado de última expresión*
- ✍ Expresiones se evalúan de izquierda a derecha.
- ✍ Se usa para secuenciar asignaciones, E/S y otras operaciones que causan efectos laterales.
- ✍ Los cuerpos de `lambda`, `let`, `let*` y `letrec`, como también las cláusulas de `cond`, `case` y `do`, se tratan como se tuvieran implícitamente un `begin`.

Ejemplo de Secuenciación



```
(define swap-pair!  
  (lambda (x)  
    (let ((tmp (car x)))  
      (set-car! x (cdr x))  
      (set-cdr! x tmp)  
      x)))
```

```
(swap-pair! (cons 'a 'b))      ;=> (b . a)
```



e) Condicionales

✍ `(if test consec alt)`

`(if test consec)`

retorno: el valor de `consec` o `alt`
según valor de `test`

✍ Si no se especifica alternativa y `test` evalúa en falso, resultado no está especificado.



not

✍ **(not obj)**

retorno: #t si obj es falso,
 sino #f

✍ **not** es equivalente a:

(lambda (x) (if x #f #t))

✍ El estándar ANSI/IEEE permite que **()** y **#f** sean equivalente, retornando **#t** para **(not `())**

and



✍ **(and *exp* ...)**

retorno: ver explicación

- ✍ Evalúa expresiones de izquierda a derecha y se detiene tan pronto alguna evalúa en falso.
- ✍ Se retorna el valor de la última evaluación.



or

✍ **(or *exp* ...)**

retorno: ver explicación

- ✍ Evalúa expresiones de izquierda a derecha y se detiene tan pronto alguna evalúa en verdadero.
- ✍ Se retorna el valor de la última evaluación.



cond (1/2)

✍ `(cond cla1 cla2 ...)`

retorno: ver explicación

✍ Cada cláusula, excepto la última, tiene la forma:

✍ `(test)`

✍ `(test exp1 exp2 ...)`

✍ `(test => exp)`

✍ La última cláusula tiene alguna de las formas anteriores o:

`(else exp1 exp2 ...)`



cond (2/2)

- ✍ Las cláusulas se evalúan en orden hasta que alguna evalúa en verdadero, o hasta que se agoten
- ✍ Si el **test** de ✍ evalúa verdadero, el valor de **test** es retornado.
- ✍ Si el **test** de ✍ evalúa verdadero, el valor de la última expresión es retornado.
- ✍ Si el **test** de ✍ evalúa verdadero, es aplicar el procedimiento de la expresión a **test** (un argumento)
- ✍ Si todas las cláusulas evalúan en falso:
 - ✍ si hay else, se retorna valor de la última expresión
 - ✍ si no hay else, valor no está especificado

Ejemplo de cond



```
(define nota
  (lambda (x)
    (cond
      ((not (symbol? x)))
      ((assq x `((juan . 68)(maria . 98)
                  (diego . 45)(lucia . 55)))
       => cdr)
      (else 0))))
```

(nota 40)	=> #t
(nota `maria)	=> 98
(nota `anastasio)	=> 0



case

✍ **(case exp_0 cla_1 cla_2 ...)**
retorno: ver explicación

✍ Cada cláusula tiene la forma:
 $((clave \dots) exp_1 exp_2 \dots)$

✍ Siendo las claves datos diferentes. Se puede poner al final una cláusula que calza todo:

$(\mathbf{else} exp_1 exp_2 \dots)$

✍ **exp_0** se evalúa, luego se busca alguna clave que calce con este valor; el resultado es el valor de la última expresión.

Ejemplo de case



```
(define f
  (lambda (x)
    (case x
      ((1 3 5 7 9) `impar)
      ((0 2 4 6 8) `par)
      (else `fuera-de-rango))))
```

```
(f 3)      => impar
(f 8)      => par
(f 14)     => fuera-de-rango
```



f) Recursión e Iteración

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

retorno: Valor de última expresión

- ✍ Es un constructor general de iteración y recursión.
- ✍ Las variables **var** ... son visibles dentro del cuerpo, como en la forma común de **let**.
- ✍ **nombre** es ligado a un procedimiento que puede iterar o llamarse recursivamente.
- ✍ En un nuevo llamado, los argumentos son los nuevos valores de las variables **var** ...




Ejemplo de let con nombre

```
(define divisores                ;;; sin recursión de cola
  (lambda (n)
    (let div ((i 2))
      (cond
        ((>= i n) '())
        ((integer? (/ n i)) (cons i (div (/ n i) i)))
        (else (div n (+ i 1)))))))
```






```
(define divisores                ;;; con recursión de cola
  (lambda (n)
    (let div ((i (- n 1)) (ls '()))
      (cond
        ((<= i 1) '())
        ((integer? (/ n i)) (div i (cons i ls)))
        (else (div (- i 1) ls))))))
```



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*. Si evalúa como
 -  **#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.

Ejemplo de do



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




for-each

- ✍ **(for-each *proc* *lista*₁ *lista*₂ ...)**
retorno: *no especificado*
- ✍ Similar a **map**, pero no crea ni retorna una lista con los resultados.
- ✍ En cambio, si garantiza orden de aplicación de ***proc*** a los elementos de las listas de izquierda a derecha.

Ejemplo de for-each



```
(define comparar
  (lambda (ls1 ls2)
    (let ((cont 0))
      (for-each
        (lambda (x y)
          (if (= x y)
              (set! cont (+ cont 1))))
        ls1 ls2)
      cont)))
```

```
(comparar '(1 2 3 4 5 6) '(2 3 3 4 7 6))
=> 3
```



g) Mapping

✍ `(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.
- ✍ No está especificado el orden en que se aplica ***proc*** a los elementos de las listas.

Ejemplo de map



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



h) Evaluación Retardada

✍ **(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 ***exp***, memorizando su valor; forzados posteriores retornan el valor memorizado.



Ejemplo de Evaluación Retardada

;;; define un stream infinito de números 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
```



i) Evaluación

✍ **(eval *obj*)**

retorno: *evaluación de **obj***
 como un 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.

Ejemplo de eval



`(eval 3)` \Rightarrow 3

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

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