4.3 Recursión en Scheme

Recursión simple y asignación

Recursión Simple

- Un ***procedimiento recursivo*** es aquel se aplica a si mismo.
- **Ejemplo**:

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

; => length

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

Ejemplo1: Procedimiento memv

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

(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)
```

Ejemplo2: 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 )
```

Ejemplo: Copiar Árbol

```
;;; tree-copy: copia en forma recursiva el arbol tr

(define tree-copy
    (lambda (tr)
       (if (not(pair? tr))
              tr
              (cons (tree-copy (car tr))
                           (tree-copy (cdr tr)))
       )
    )
)
; =>tree-copy

(tree-copy '(a (b c) d (d (e f(h)))))
; =>(a (b c) d (d (e f (h))))
```

Ejemplo de Recursión (1/3)

```
;;; abs-all: genera una lista con los valores
;;; absolutos de la lista ls

(define abs-all
       (lambda (ls)
              (if (null? ls)
                     ()
                     (cons (abs (car ls)) (abs-all (cdr ls)))
              )
       )
)
; => abs-all

(abs-all '(3 7 -10 5 -8))  ; => (3 7 10 5 8)
```

Ejemplo de Recursión (2/3)

```
;;; abs-all1: genera una lista con los valores
;;; absolutos de la lista ls, usando el procedimiento map

(define abs-all1
       (lambda (ls) (map abs ls))
)

; => abs-all1


(abs-all1 '(3 7 -10 5 -8)) ; => (3 7 10 5 8)

;;; que se puede expresar directamente como:

(map abs '(3 7 -10 5 -8))  ; => (3 7 10 5 8)
```

Ejemplo de Recursión (3/3)

```
;;; map1: implementa la funcion map

(define map1
       (lambda (proc ls)
          (if (null? ls)
              ()
              (cons (proc (car ls)) (map1 proc (cdr ls)))
          )
       )
)
; =>map1

(map1 abs '(3 7 -10 5 8))  ; => (3 7 10 5 8)
```

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 (global).
- Sin embargo, **let** y **define** no permiten cambiar el ligado de una variable ya existente, como lo haría una asignación.
  - Son constantes!
- **set!** Permite en *Scheme* re-ligar a una variable existente un nuevo valor, como lo haría una asignación.

Ejemplo: Asignación

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

abcde
; => (a b c d e)


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

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

Ejemplo: Ecuación Cuadrada (1/2)

```
(define raices-cuadradas
       (lambda (a b c)
           (let ((menosb 0) (raiz 0)
                     (divisor 1) (raiz1 0) (raiz2 0))

              (set! menosb (- 0 b))
              (set! divisor (* 2 a))
              (set! raiz (sqrt (- (* b b) (* 4 (* a c)))))
              (set! raiz1 (/ (+ menosb raiz) divisor))
              (set! raiz2 (/ (- menosb raiz) divisor))

              (cons raiz1 raiz2)
           )
       )
)
; => raices-cuadradas
```

Ejemplo: Ecuación Cuadrada (2/2)

```
(define raices-cuadradas1
       (lambda (a b c)
              (let  ( (menosb (- 0 b))
                      (raiz (sqrt (- (* b b) (* 4 (* a c)))))
                      (divisor (* 2 a))
                    )
                    (let ((raiz1 (/ (+ menosb raiz) divisor))
                          (raiz2 (/ (- menosb raiz) divisor)))
                          (cons raiz1 raiz2)))))

; => raices-cuadradas1


(raices-cuadradas 2 -4 -30)       ; => (5 . -3)

(raices-cuadradas1 2 -4 -30)      ; => (5 . -3)
```

Ejemplo: Contador (1/3)

```
;;; 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
```

Ejemplo: Contador (2/3)

```
;;; 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
```

Ejemplo: Contador (3/3)

```
(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
```

Ejemplo: Evaluación Perezosa (1/2)

```
;;; lazy: evaluacion perezosa de la expresion t, la cual
;;; es solo evaluada la primera vez que se invoca.

(define lazy
    (lambda (t)
       (let ((val #f) (flag #f))
              (lambda ()
                 (if (not flag)
                     (begin (set! val (t))(set! flag #t))
                 )
                 val
              )
       )
    )
)
; => lazy
```

Ejemplo: Evaluación Perezosa (2/2)

```
(define imprime
    (lazy (lambda ()
              (display "Primera vez!")
              (newline)
              "imprime: me llamaron"
          )
     )
)
; => imprime

(imprime)
Primera vez!
; => "imprime: me llamaron"

(imprime)
; => "imprime: me llamaron"
```

Ejemplo: Stack (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
```

Ejemplo: Stack (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?)               ; => ()
(st 'pop!)                 ; => (gato perro)
```

4.4 Aspectos más Avanzados sobre Scheme

Otras formas de definición de variables y recursión de cola

Formas `let` para 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
- **Ejemplo**:

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

¡No es visible!

Recursión con `let`

```
;;; se puede solucionar el problema entregando
;;; suma como argumento a la expresion lambda,
;;; siendo la solución más 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
```

Aplicación de `letrec`

```
;;; letrec hace visible las variables dentro
;;; de los valores definido, 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))
)
```

Ejemplo de Recursión Mutua

```
(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 ())
```

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)

Ejemplo de `letrec`

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

¡Es válido!

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

¡No es válido!

Definición `let` con nombre

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

;=> 21
```

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 ...)
```

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, ó indirectamente a través de una serie de llamados de cola hacia si mismo.

Ejemplo: Llamado de Cola

- Lo son los llamados a f:

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

- No lo son respecto 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.

Ejemplo1: Factorial (1/2) (sin recursión de cola)

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

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

(factorial 5)           ; => 120
```

Ejemplo1: Factorial (2/2) (con recursión de cola)

n! = n * (n-1) * (n-2) * ... 2 * 1

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

(factorial1 5)          ; => 120
```

Ejemplo2: Fibonacci (1/2) (sin recursión de cola)

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

```
(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
```

Ejemplo2: Fibonacci (2/2) (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

(fibonacci1 20)            ; => 6765
```

Ejemplo de Factorización

```
(define factorizar
    (lambda (n)
       (let fact ((n n) (i 2))
          (cond
              ((>= i n) '())
              ((integer? (/ n i)) (cons i (fact (/ n i) i)))
              (else (fact n (+ i 1)))
))))
;=> factorizar
```

No recursiva de cola

Recursiva de cola

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

Continuaciones

- Evaluación de expresiones ***Scheme*** implica dos decisiones:
  - ¿Qué evaluar?
  - ¿Qué hacer con el valor de la evaluación?
- Se denomina ***continuación*** al punto de la evaluación donde se tiene un valor y se está listo para seguir o terminar.

Procedimiento call-with-current-continuation

- Procedimiento que se puede abreviar como `call/cc`
- `call/cc` se pasa como argumento formal a un procedimiento `p`
- `call/cc` obtiene la continuación actual y se lo pasa como argumento actual a `p`.

Procedimiento call-with-current-continuation

- La continuación se representa por **k**, donde cada vez que se aplica a un valor, éste retorna este valor a la continuación de la aplicación de **`call/cc`**.
- Este valor se convierte en el valor de la aplicación de **`call/cc`**.
- Si **p** retorna sin invocar k, el valor retornado por **p** es el valor de la aplicación de **`call/cc.`**

Ejemplo de Continuación

```
(define call/cc call-with-current-continuation)
;=> Value: call/cc

(call/cc (lambda (k) (* 4 5)))
;=> Value: 20


(call/cc (lambda (k) (* 5 (+ 2 (k 7)))))
;=> Value: 7
```

Ejemplo 1: call/cc

```
(define producto
   (lambda (ls)
       (call/cc
           (lambda (salir)
              (let prod ((ls ls))
                 (cond
                     ((null? ls) 1)
                     ((= (car ls) 0) (salir 0))
                     (else (* (car ls)(prod (cdr ls))))
              ))))))
;=> Value: producto

(producto '())                          ;=> Value: 1
(producto '(1 2 3 4 5))                 ;=> Value: 120
(producto '(2 4 5 0 5 6 8 3 5 7))       ;=> Value: 0
```

Ejemplo 2: call/cc

```
(define list-length
   (lambda (ls)
       (call/cc
          (lambda (retorno)
              (letrec
                  ((len (lambda (ls)
                          (cond
                            ((null? ls) 0)
                            ((pair? ls) (+ 1 (len (cdr ls))))
                            (else (retorno #f))
                  ))))
                  (len ls)
              ))))))
;=> Value: list-length

(list-length '(a b c d e))        ;=> Value: 5
(list-length '(a . b))            ;=> Value: ()
```

Ejemplo de memorizar una continuación

```
(define retry #f)
;=> retry

(define factorial
   (lambda (x)
       (if (= x 0)
           (call/cc (lambda (k) (set! retry k) 1))
           (* x (factorial (- x 1)))
)))
;=> factorial

(factorial 4)       ;=> 24
retry               ;=>  #[continuation 13]
(retry 2)           ;=> 48
```