Lenguajes de Programación
Programación funcional

Dr. Mauricio Araya Roberto Bonvallet

Primer Semestre 2014

Agenda

Índice

1. Introducción a la programación funcional

Concepto de función

En el paradigma imperativo:

- Una función es una secuencia de instrucciones
- El valor de retorno se obtiene a través de una serie de manipulaciones de estado (cambios en valores de variables)
- Existencia de estado implica posibles efectos laterales
- Consecuencia de la arquitectura de von Neumann:
  - Programas son datos en memoria
  - Unidad de procesamiento (CPU) está separada de la memoria
  - Instrucciones y datos son transmitidos de la memoria a la CPU
  - Resultados de operaciones son transmitidos de la CPU a la memoria

Concepto de función

En matemáticas:

- Una función es un mapeo de los elementos de un conjunto (dominio) a los de otro (rango): $f: A \to B$
- Generalmente descrita por una expresión o una tabla de valores
- Evaluación controlada por recursión y expresiones condicionales, a diferencia de la programación imperativa, que utiliza sentencias de repetición y de ejecución condicional
- Libre de efectos laterales: Para los mismos parámetros, cada evaluación de la función entrega el mismo resultado

Influencia del paradigma imperativo en la programación

- Lenguajes imperativos están diseñados para explotar la arquitectura del computador
- Apego a la arquitectura es una restricción innecesaria al proceso de desarrollo de software
- Lenguajes diseñados según otros paradigmas son impopulares debido a penalizaciones de rendimiento
- Máquina de Turing: Modelo matemático para estudiar capacidades y limitaciones del computador

2. Fundamentos de la programación funcional

Fundamentos de los lenguajes funcionales

- Objetivo: Emular las funciones matemáticas lo más posible. Esto conduce a un enfoque para resolver problemas que es fundamentalmente diferente de los métodos imperativos.
- No se usan variables ni asignaciones: El programador no debe preocuparse sobre el estado de las celdas en la memoria.
- Repetición a través de recursión.
- Transparencia referencial: No hay estado, no hay efectos laterales.
- Funciones son objetos de primera clase: Se pueden tratar como cualquier otro valor. Por ejemplo, se pueden pasar como parámetros a otras funciones.
- Un programa consiste de definiciones de funciones y aplicaciones de ellas.

Funciones

Funciones simples

- Ejemplo: cubo$(x) \equiv x * x * x$
- x representa cualquier valor del dominio, pero está fijo en un elemento específico durante la evaluación
- Durante la evaluación, cada aparición de un parámetro está ligada a un valor y es considerada constante

- Notación lambda para funciones anónimas permite referirse a una función sin necesidad de ligarla a un nombre; por ejemplo, la función "elevar al cubo" se representa así:

$$\lambda(x)\, x * x * x$$

- Evaluación de una función anónima:

$$\big(\lambda(x)\, x * x * x\big)(2) = 8$$

Funciones

Formas funcionales (funciones de mayor orden)

- Reciben funciones como parámetros, o entregan funciones como resultado. Por ejemplo:

  Composición

  $$\big(f \circ g\big)(x) = f\big(g(x)\big)$$

  Construcción

  $$\big[g, h, i\big](x) = \big(g(x), h(x), i(x)\big)$$

  Aplicar a todo

  $$\alpha\big(j, (x, y, z)\big) = \big(j(x), j(y), j(z)\big)$$

Comparación entre un programa imperativo y uno funcional

- Problema sencillo: Aplicar dos funciones a una lista de valores
- Programa imperativo:

  target $\leftarrow$ []
  for i **in** source **do**
  $t_1 \leftarrow g(i)$
  $t_2 \leftarrow f(t_1)$
  target.append(t_2)
  end for

- Distintas versiones funcionales del mismo programa:

  target $\leftarrow \alpha(f \circ g, \text{source})$
  target $\leftarrow \alpha\Big(\lambda(x)\, f\big(g(x)\big), \text{source}\Big)$
  target $\leftarrow \Big[f\big(g(i)\big)\ \forall i \in \text{source}\Big]$

- El programa imperativo describe los pasos necesarios para construir la lista resultado, mientras los programas funcionales describen relaciones matemáticas entre las listas fuente y resultado.

3. El lenguaje de programación Scheme

Introducción a Scheme

Características:

- Dialecto de LISP
- Sintaxis y semántica simples
- Nombres tienen ámbito estático
- No es puramente funcional: Es posible tener efectos laterales, cambiar asignaciones, alterar listas, etc., en caso de que sea conveniente.

Funcionamiento del intérprete de Scheme

- Entorno interactivo implementa un ciclo lectura-evaluación-escritura: El intérprete lee una expresión ingresada por el usuario, la evalúa y entrega el resultado. Las expresiones tienen estructura de lista.
- Se pueden cargar las expresiones desde un archivo para facilitar el desarrollo.
- Las llamadas a funciones se evalúan así:
  1. Se evalúa la función
  2. Se evalúan los parámetros, en ningún orden en particular;
  3. Se aplica la función sobre los parámetros.

Funcionamiento del intérprete de Scheme

Ejemplo de sesión interactiva

```
12
;--> 12
'(1 2 3 4)
;--> (1 2 3 4)
(+ 1 2 3)
;--> 6
(string-append "hola␣" "mundo")
;--> "hola mundo"
```

Sintaxis del lenguaje

Scheme utiliza la misma sintaxis para:

1. Listas:

   ```
   (1 2 3 4 5)
   ```

2. Llamadas a función:

   ```
   (max 43 -23 15 58 21)
   ```

3. Formas sintácticas:

   ```
   (if (< x 0) x (- x))
   ```

Constantes literales

- Valores literales pueden ser:
  - Números enteros, reales, racionales y complejos:

```
123456789
3.14159
22/7
+27.0-3.0i
5.0@3.0
```

  - Caracteres:

```
#\a
#\space
```

  - Booleanos:

```
#t
#f
```

  - Strings:

```
"Hola␣mundo"
```

- Al ser evaluadas en el intérprete, las constantes entregan su propio valor.

4. Listas

Listas

- Notación de listas:

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

- Los elementos pueden ser de cualquier tipo: La lista

```
(1.0 "Hola␣mundo" 3)
```

  contiene un número de punto flotante, una cadena de caracteres y un entero.

- Las listas se pueden anidar: La lista

```
(a (b c) (d e (f) g))
```

  tiene tres elementos; el segundo y el tercero son listas.

- Las listas se tratan como objetos inmutables: En general no se agregan, eliminan o modifican elementos de una lista. Las operaciones sobre listas entregan una nueva lista.

Representación interna de las listas

- Implementadas como listas enlazadas
- Los nodos se llaman *pares* o *celdas cons*
- El final de una lista se marca con la lista vacía

Figura 1: Representación de `(1 2 3 4 5)`

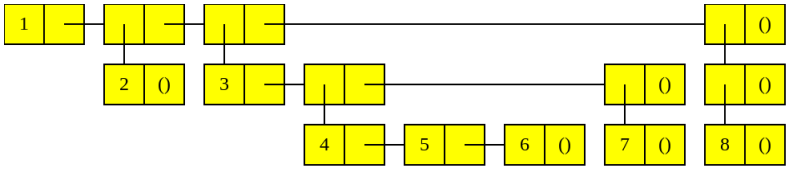

Figura 2: Representación de `(1 (2) (3 (4 5 6) (7)) ((8)))`

Representación interna de las listas

Celdas cons

- Las celdas cons son el tipo de datos estructurado fundamental de Scheme.
- El campo izquierdo de una celda se denomina el *car*, y el campo derecho, el *cdr*.
- En una lista enlazada propia, las celdas tienen:
  - Un valor en el car;
  - Una referencia a una lista en el cdr.

Celdas cons

- Las funciones `car` y `cdr` obtienen el car y el cdr de una celda.
- La función `cons` permite crear una celda cons definiendo su car y su cdr, y por lo tanto se puede usar para crear listas enlazadas.

```
(car '(1 2 3 4 5))
;--> 1
(cdr '(1 2 3 4 5))
;--> (2 3 4 5)
(cons 'a '(b c d))
;--> (a b c d)
```

Listas impropias

- Una lista es impropia cuando el cdr de su última celda no es la lista vacía.
- Las listas impropias se representan usando la notación de par con punto.

Listas impropias

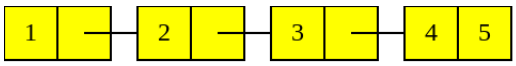

Figura 3: Representación de `(1 2 3 4 . 5)`

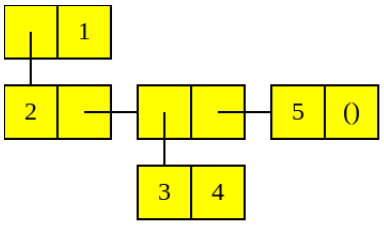

Figura 4: Representación de `((2 (3 . 4) 5) . 1)`

Listas impropias

5. Funciones

Llamadas a funciones

- Llamadas a funciones utilizan la misma notación que las listas:

```
(funcion arg$_1$ arg$_2$ ... arg$_N$)
```

- Para evitar que una lista sea evaluada como función, se utiliza el operador de citado:

```
(a b c d)
;error:  'a' no es una funcion
(quote (a b c d))
;--> (a b c d)
'(a b c d)
;--> (a b c d)
```

Funciones aritméticas

- Scheme provee las funciones aritméticas elementales `+`, `-`, `*` y `/`:

```
(+ 2 -3)
;--> -1
(* 1 2 3 4 5)
;--> 120
(+ (* 3 3) (* 4 4))
;--> 25
(- (/ 81 9) (*) (+))
;--> 8
```

- Notación prefija evita ambigüedad en la evaluación

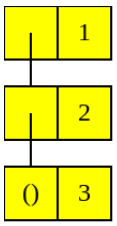

Figura 5: Representación de `((() . 3) . 2) . 1)`

Funciones anónimas

- La forma sintáctica `lambda` permite definir funciones anónimas.
- Sintaxis:

```
(lambda (arg$_1$ arg$_2$ ...) expr$_1$ expr$_2$ ...)
```

- Ejemplo:

```
(lambda (x y) (+ (* x x) (* y y)))
;--> #<procedure>
((lambda (x y) (+ (* x x) (* y y))) 3 4)
;--> 25
```

Sintaxis alternativas de lambda

- Guarda en args la lista de todos los argumentos:

```
(lambda args expr$_1$ expr$_2$ ...)
```

- Guarda en rest todos los argumentos adicionales:

```
(lambda (arg$_1$ ... arg$_N$ . rest) expr$_1$ expr$_2$ ...)
```

Ligados locales

- La forma sintáctica `let` establece ligados de nombres locales.
- Nombres ligados con `let` son visibles sólo en el cuerpo del `let`.
- Sintaxis:

```
(let ((var$_1$ val$_1$) (var$_2$ val$_2$) ...)
    expr$_1$ expr$_2$ ...)
```

- Ejemplo:

```
(let ((a 17) (b 31))
  (+ (* 1/2 a b) (* 1/3 a b)))
;--> 439.166666666667

((lambda (x y)
   (let ((x-squared (* x x))
         (y-squared (* y y)))
     (sqrt (+ x-squared y-squared))))
 5 12)
;--> 13.0
```

Definiciones de nivel superior

- La forma sintáctica `define` liga un nombre a una expresión , generalmente fuera del ámbito de expresiones `lambda`.
- Nombres ligados con `define` son visibles desde cualquier expresión, siempre que no sean ocultados por ligados locales.
- Sintaxis:

  ```
  (define nombre expr)
  ```

- Ejemplo:

```
(define n 25)

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

(square n)
;--> 625
```

6. Expresiones Condicionales

Expresiones condicionales simples

- La forma sintáctica `if` permite evaluar una expresión de manera condicional.
- Sintaxis:

  ```
  (if condicion consecuencia alternativa)
  ```

- Ejemplo:

```
(define signo
  (lambda (x)
    (if (>= x 0) 'positivo 'negativo)))

(signo -2)
;--> negativo

(signo 5)
;--> positivo
```

Expresiones condicionales múltiples

- La forma sintáctica `cond` permite evaluar una expresión usando distintos casos.
- Sintaxis:

```
(cond clausula$_1$ clausula$_2$ ...)
```

- Cada cláusula tiene alguna de estas formas:
  1. `(condicion)`: entrega el valor de `condicion`.
  2. `(condicion` $\texttt{expr}_1$ $\texttt{expr}_2$ `...)` : evalúa las expresiones y entrega el valor de la última de ellas.
  3. `(condicion => expr)` : se evalúa la expresión, que debe entregar una función de un argumento, que es aplicada al resultado de `condicion`.
- La última cláusula puede tener la forma:
  4. `(else` $\texttt{expr}_1$ $\texttt{expr}_2$ `...)`: cubre el caso en que ninguna de las cláusulas sea verdadera.

Ejemplo de uso de cond

```
(define income-tax
  (lambda (income)
    (cond
      ((<= income 10000) (* income .05))
      ((<= income 20000)
        (+ (* (- income 10000) .08)  500.00))
      ((<= income 30000)
        (+ (* (- income 20000) .13) 1300.00))
      (else
        (+ (* (- income 30000) .21) 2600.00)))))
```

Evaluación de valores de verdad

- Todos los objetos son considerados verdaderos, excepto `\#f`.
- Existen las formas sintácticas `and`, `or` y `not`.
- `and` y `or` hacen cortocircuito.
- En general, los predicados lógicos tienen nombres terminados en `?`.

Ejemplos de valores de verdad

```
(if 1 'true 'false)
;--> true
(if '() 'true 'false)
;--> true
(if #f 'true 'false)
;--> false
(not #t)
;--> #f
(not "false")
;--> #f
```

```
(or)
;--> #f
(or #f #t)
;--> #t
(or #f 'a #t)
;--> 'a
```

Ejemplos de predicados lógicos

```
(< 1 2 3 4 5)
;--> #t
(null? '())
;--> #t
(null? '(1 2 3))
;--> #f
(pair? '(8 . 2))
;--> #t
(eqv? 3 2)
;--> #f
(number? 15)
;--> #t
(list? "Veamos si soy una lista...")
;--> #f
```

7. Recursión

Recursión simple

- En Scheme, la iteración se hace usando recursión.
- La recursión es más general que construcciones tipo *for* o *while*, y elimina la necesidad de usar variables de control.
- Una función recursiva debe tener:
  1. Casos base
  2. Pasos de recursión
- Ejemplo:

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

(length '())
;--> 0

(length '(a b c))
;--> 3
```

Ligados locales de funciones recursivas

- `let` presenta problemas con la recursión: Los nombres ligados sólo son alcanzables en el cuerpo de la expresión `let`.

```
(let ((sum (lambda (ls)
    (if (null? ls)
        0
        (+ (car ls) (sum (cdr ls)))
    ))))
  (sum '(1 2 3 4 5)))
;-->Error
```

Ligados locales de funciones recursivas

- **Solución 1:** pasar la función como argumento a sí misma.

```
(let ((sum (lambda (sum ls)
    (if (null? ls)
        0
        (+ (car ls) (sum sum (cdr ls)))
    ))))
  (sum sum '(1 2 3 4 5)))
;--> 15
```

Ligados locales de funciones recursivas

- **Solución 2:** `letrec` permite usar los nombres a ligar como parte de la expresión de los valores a asignar.
  `(letrec ((var val) ...) exp1 exp2 ...)`

```
(letrec ((sum (lambda (ls)
    (if (null? ls)
        0
        (+ (car ls) (sum (cdr ls)))
    ))))
  (sum '(1 2 3 4 5)))
;--> 15
```

Recursión mutua

- Con `letrec` podemos definir funciones mutuamente recursivas.

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

Restricción de `letrec`

- Hay una restricción para el uso de `letrec`: Cada valor a asignar (*val*) debe poder ser evaluado sin tener que evaluar alguna variable (*var*).
- Esto es una restricción sencilla pensando que muchas veces los valores a asignar son expresiones *lambda* en que pueden aparecer nombres aún sin ligar.

```
(letrec ((y (+ x 2))
         (x 1))
  y)
;error: depende de la implementacion

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

`let` con nombre

- La expresión es conocida como *let con nombre* `(let name ((var val) ...) exp1 exp2 ...)` y es equivalente a

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

```
(define cuenta (lambda (x l)
    (let aux ((x x) (l l) (r 0))
        (cond
            ((null? l) r)
            ((eqv? x (car l))
                (aux x (cdr l) (+ 1 r)))
            (else (aux x (cdr l) r))
        )
    ))
)

(cuenta 'a '(a b a c a d))
;--> 3
```

Llamadas de cola

- Si en una expresión lambda hay algún llamado cuyo valor será retornado inmediatamente por la expresión lambda (o sea, después de la evaluación del llamado no queda nada más que hacer que retornar su valor), se dice que este es un *llamado en posición de cola* o *llamado de cola*.
- Usar llamados de cola tiene la ventaja de poder ser optimizado por el intérprete (o compilador) como un simple *"goto"* (no requiere guardar estado en el stack de llamadas).
- En los siguientes ejemplos, todas las llamadas a f están en posición de cola, pero las llamadas a g no.

```
(lambda () (f (g)))
(lambda () (if (g) (f) (f)))
(lambda () (let ((x 4)) (f)))
(lambda () (or (g) (f)))
```

Recursión de cola

- Cuando una función recursiva sólo usa llamados de cola para su paso recursivo se dice que es una *función recursiva de cola*.
- Las llamadas de cola toman mayor importancia en funciones recursivas, pues estas suelen requerir un gran número de pasos recursivos para llegar a un resultado, lo que se traduce en un alto número de recursos consumidos para ello.
- Una función recursiva se puede decir que básicamente es una función *iterativa*.
- Toda función recursiva puede escribirse como una función recursiva de cola.

Recursión de cola

$n! = n \times (n-1)!$

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

$n! = n \times (n-1) \times \cdots \times 2 \times 1$

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

Recursión de cola

No recursivo de cola...

```
(factorial 5)
|(factorial 5)
| (fact 4)
| |(fact 3)
| | (fact 2)
| | |(fact 1)
| | | (fact 0)
| | | 1
| | |1
| | 2
| |6
| 24
|120
120
```

Recursión de cola

Recursivo de cola...

```
(factorial 5)
|(factorial 5)
|(fact 4 5)
|(fact 3 20)
|(fact 2 60)
|(fact 1 120)
|(fact 0 120)
|120
120
```

8. Temas Avanzados

Secuenciaron

```
(begin exp1 exp2 ...)
; retorno: resultado de ultima expresion
```

- 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 si tuvieran implícitamente un `begin`.

Re-ligado o 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.
- Cuidado, `set!` no cumple con la filosofía funcional.

Ejemplo de Re-ligado

```
(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 de Contador

```
(define contador 0)
; => contador
(define cuenta
        (lambda ()
                (set! contador (+ contador 1))
                contador))
; => cuenta
(cuenta)                ; => 1
(cuenta)                ; => 2
```

Ejemplo de Generador de Contadores

```
(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
```

Evaluación Perezosa

- Es sólo evaluada la primera vez que se invoca
- Útil para optimizar evaluaciones complejas y estáticas
- En Scheme se utiliza el funcional `lazy`

```
(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

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

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

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 Evaluación Retardada

```
;;; 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
```

Continuaciones

- Evaluación de expresiones Scheme implica dos decisiones:
  1. ¿Qué evaluar?
  2. ¿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.
- Para hacer esto se utiliza `call-with-current-continuation`
- Desagradablemente largo: Versión de bolsillo `call/cc`

`call/cc`

- Se pasa como argumento formal a un procedimiento `p`
- Obtiene la continuación actual y se lo pasa como argumento actual a `p`
- 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 `call/cc`

```
(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 `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 `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: ()
```

Apply and Eval

```
(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.

```
(eval obj)
; retorno: evaluacion de obj como un programa Scheme
```

`eval` evalúa un programa Scheme `obj`, y el ámbito es distinto

Mapeado de listas

```
(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)
```

Cuasi-citaciones

```
(quasiquote obj) (o `obj)
(unquote obj)    (o ,obj)
(unquote-splicing obj) (o ,@obj)
```

- `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`.

Ejemplo Cuasi-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)
```

9. Temas Misceláneos

Do

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

- Lar 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.
- Uselo solo en caso de emergencia!

Ejemplo 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 lista1 lista2 ...)
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.
- Uselo solo en caso de emergencia!

Ejemplo 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
```

Entrada y Salida

- Toda E/S se realiza por puertos.
- Un puertos es un puntero a un flujo (posiblemente infinito) de caracteres (e.g. archivo)
- Un puerto es un objeto de primera clase
- El sistema tiene dos puertos por defecto: entrada y salida estándar (normalmente es el terminal en un ambiente interactivo)
- Cuando se agotan los datos de entrada se retorna el objeto especial `eof`

Operaciones de Entrada

```
(input-port? obj)
;retorno: #t si obj es puerto de entrada
(current-input-port)
;retorno: puerto de entrada actual
(open-input-file filename)
;retorno: nuevo puerto de entrada
(call-with-input-file file proc)
;accion: abre file, llama a proc y cierra puerto
;retorno: valor de evaluacion de proc
(close-input-port input-port)
;accion: cierra puerto de entrada
(eof-object? obj)
;retorno: #t si obj es fin de archivo
(read)
(read input-port)
;retorno: proximo objeto de input-port
```

Ejemplo de Entrada

```
(input-port? '(a b c))                  => unspecified
(input-port? (current-input-port))      => #t
(input-port? (open-input-file "f.dat")) => #t

(let ((p (open-input-file "mi_archivo")))
  (let leer ((x (read p)))
    (if (eof-object? x)
      (begin
        (close-input-port p)
        '())
      (cons x (leer (read x))))))
```

Operaciones de Salida

```
(output-port? obj)
;retorno: #t si obj es puerto de salida
(current-output-port)
;retorno: puerto de salida actual
(open-output-file file)
;retorno: nuevp puerto de salida
(call-with-output-file file proc)
;accion: abre file, llama a proc y cierra puerto
;retorno: valor de evaluacion de proc
(close-output-port output-port)
;accion: cierra puerto de salida
(write obj)
(write obj output-port)
;accion: escribe obj a output-port
(newline)
(newline output-port)
;accion: escribe nueva linea a output-port
```

Ejemplo de Salida

```
(define copy-file
  (lambda (infile outfile)
    (call-with-input-file infile
     (lambda (iport)
       (call-with-output-file outfile
        (lambda (oport)
         (do ((c (read-char iport)
           (read-char iport)))
           ((eof-object? c))
         (write-char c oport))))))))
;=> copy-file
(copy-file "infile" "outfile")
;=> #t
```

EOC

FIN