T.2 Programación funcional (*Lisp)

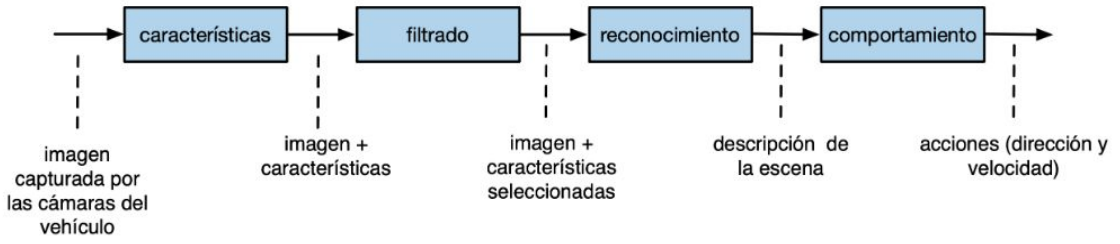
Es un tipo concreto de **programación declarativa:**

- **Declaración funciones matemáticas puras** (devuelven un único resultado, no modifican parámetros pasados, resultado solo depende de param. entrada), **sin estado interno ni efectos laterales** -> suma, resta, etc.
- Uso de la **recursión**
- **Listas** son estructuras de datos fundamentales
- **Funciones** como tipos de **datos primitivos**: expresiones lambda y funciones de orden superior
- **Variable == nombre a un valor** (**declaración**)
- ¡NO! Mutación, sino **Transparencia Referencial (en mismo ámbito, todas ocurrencias de variables y llamadas a funciones devuelven el mismo valor)**
- ¡NO! Pasos de ejecución, sino **composición de funciones**

vs a **programación imperativa:**

- **Variable == zona de memoria**
- **Referencias**
- **Pasos de ejecución**
- **Asignación** (mutación)

Composición de funciones -> transformar datos de entrada en otros datos de salida




```
(define (conduce-vehiculo imagenes)
    (obten-acciones
        (reconoce
            (filtra
                (obten-caracteristicas imagenes)))))
```

Modelo de computación de sustitución

Define cuatro reglas sencillas para evaluar una expresión.

1. Si *e* es un valor primitivo (ejemplo, un número), devolvemos valor.
2. Si *e* es un identificador, devolvemos su valor asociado con un `define` (error si no existe ese valor).
3. Si *e* es una expresión *(f arg1 ... argn)*, donde *f* es el nombre de una función primitiva (`+`, `-`, ...), evaluamos uno a uno los argumentos *arg1 ... argn* (con estas mismas reglas) y evaluamos la función primitiva con los resultados.

dependiendo del orden de evaluación que utilizamos:

Orden aplicativo

4. Si *e* es una expresión *(f arg1 ... argn)*, donde *f* es el nombre de una función definida con un `define`, **evaluar primero los argumentos** *arg1 ... argn* y después **sustituir *f* por su cuerpo**, **reemplazando cada parámetro** formal de la función por el correspondiente **argumento evaluado**. Después evaluaremos la expresión resultante usando estas mismas reglas.

Orden normal

4 Si *e* es una expresión *(f arg1 ... argn)*, donde *f* es el nombre de una función definida con un `define`, **sustituir *f* por su cuerpo**, **reemplazando cada parámetro** formal de la función por el correspondiente **argumento sin evaluar**. Después evaluar la expresión resultante usando estas mismas reglas.

Ej: Siendo f (x) = (+ (cuadrado (doble x)) 1)

Resultado de la evaluación usando el **modelo de sustitución aplicativo**:

```
(f (+ a 1)) ⇒                ; Para evaluar f, evaluamos primero su argumento
                            ; y sustituimos a por 2 (R2)
(f (+ 2 1)) ⇒                ; Evaluamos (+ 2 1) (R3)
(f 3) ⇒                      ; (R4)
(+ (cuadrado (doble 3)) 1) ⇒ ; Sustituimos (doble 3) (R4)
(+ (cuadrado (+ 3 3)) 1) ⇒   ; Evaluamos (+ 3 3) (R3)
(+ (cuadrado 6) 1) ⇒         ; Sustitumos (cuadrado 6) (R4)
(+ (* 6 6) 1) ⇒              ; Evaluamos (* 6 6) (R3)
(+ 36 1) ⇒                   ; Evaluamos (+ 36 1) (R3)
37
```

Y veamos el resultado de usar el **modelo de sustitución normal**:

```
(f (+ a 1)) ⇒                     ; Sustituimos (f (+ a 1))
                                  ; por su definición, con z = (+ a 1) (R4)
(+ (cuadrado (doble (+ a 1))) 1) ⇒ ; Sustituimos (cuadrado ...) (R4)
(+ (* (doble (+ a 1))
      (doble (+ a 1))) 1)         ; Sustituimos (doble  ...) (R4)
(+ (* (+ (+ a 1) (+ a 1))
      (+ (+ a 1) (+ a 1))) 1) ⇒    ; Evaluamos a (R2)
(+ (* (+ (+ 2 1) (+ 2 1))
      (+ (+ 2 1) (+ 2 1))) 1) ⇒    ; Evaluamos (+ 2 1) (R3)
(+ (* (+ 3 3)
      (+ 3 3)) 1) ⇒                ; Evaluamos (+ 3 3) (R3)
(+ (* 6 6) 1) ⇒                   ; Evaluamos (* 6 6) (R3)
(+ 36 1) ⇒                        ; Evaluamos (+ 36 1) (R3)
37
```

+***En paradigma funcional ambos modelos darán el **mismo resultado*****

Ej2: (define (zero x) (- x x))
(zero (random 10)) //Random no pertenece al paradigma funcional

Scheme: lenguaje de prog. funcional

Funciones

- Se evalúan por **modelo de sustitución aplicativo**
- Las expresiones siempre se evalúan **desde los paréntesis interiores** a los **exteriores**.

Formas especiales

- Forma de evaluación propia

////////////////////////// define ////////////////////////////////

(define <identificador> <expresión>)

- Asocia un id. a un valor
- Ej:

  (define altura 12) ; -> Asociamos a 'altura' el valor 12

  (define area (/ (* base altura) 2)) ; -> Asociamos a 'area' el valor 60

(define (<nombre-funcion> <argumentos>) <cuerpo>)

- Define una función
- Ej:

  ```
  (define (factorial x)
     (if (= x 0)
        1
        (* x (factorial (- x 1)))))
  ```
  -> Se crea la función con el cuerpo y le asociamos el nombre ‘factorial’

//

//////////////////////////// if ////////////////////////////

(if <condición> <expresión-true> <expresión-false>)

- Evaluar condición, y si el resultado es #t, evaluar la expresión-true, en otro caso, evaluar la expresión-false.

- Ej: (if (> 10 5) (substring "Hola qué tal" (+ 1 1) 4) (/ 12 0))

-> Evaluamos (> 10 5). Como el resultado es #t, evaluamos (substring "Hola qué tal" (+ 1 1) 4), que devuelve "la"

//

//////////////////////////// cond ////////////////////

(cond

(<exp-cond-1> <exp-consec-1>)

(<exp-cond-2> <exp-consec-2>)

...

(else <exp-consec-else>))

- Se evalúan de forma ordenada todas las exp-cond-i hasta que una de ellas devuelva #t
- Si alguna exp-cond-i devuelve #t, se devuelve el valor de la exp-consec-i.
- Si ninguna exp-cond-i es cierta, se devuelve el valor resultante de evaluar exp-consec-else.
- Ej:

```
(cond
  ((> 3 4) "3 es mayor que 4")
  ((< 2 1) "2 es menor que 1")
  ((= 3 1) "3 es igual que 1")
  ((> 3 5) "3 es mayor que 2")
  (else "ninguna condición es cierta"))
```

//

//////////////// and y or ////////////////////

(and exp1 ... expn)

- Ej:
  (and #f (/ 3 0)) ; ⇒ #f
  (and #t (> 2 1) (< 5 10)) ; ⇒ #t

(or exp1 ... expn)

- Ej:
  (or #t (/ 3 0)) ; ⇒ #t
  (or #f (< 2 10) (> 5 10)) ; ⇒ #t

//

////////////////////// quote //////////////////////////

(quote <identificador>) == ‘<identificador>

- Se devuelve el identificador **sin evaluar** (un símbolo).
- Ej:
  (quote x) ; -> el símbolo x
  'hola ; -> el símbolo hola

Nota:

Identificador** es un dato del lenguaje de **tipo symbol**
*** tipo Symbol (tipo atómico) != tipos Cadena (tipo compuesto)***

//

////////// quote con expresiones ////////////

(quote <expresiónr>) == ‘

- Si quote recibe una expresión correcta de Scheme (una expresión entre paréntesis) se devuelve la **lista** o **pareja** definida por la expresión (**sin evaluar sus elementos**).
- Ej:
  (quote (1 2 3 4)) ; -> La lista formada por los números 1 2 3 4
  '(a b c) ; -> La lista con los símbolos a, b, y c
  '(* (+ 1 (+ 2 3)) 5) ; -> Una lista con 3 elementos, el segundo de ellos otra lista
  '(1 . 2) ; -> La pareja (1 . 2)
  '((1 . 2) (2 . 3)) ; -> Una lista con las parejas (1 . 2) y (2 . 3)

//

Listas

- Una lista en Scheme puede contener cualquier valor, incluyendo otras listas, parejas, etc.
- Construye una lista llamando a la función **list** y pasándole un número variable de parámetros que son los **elementos** que posteriormente **evaluados, se incluirán** en la list.

  (list 1 (+ 1 1) (* 2 (+ 1 2))) ; ⇒ (1 2 6)

  *** **quote** también la crea, pero **sin evaluar los elementos**. ***

  '(1 (+ 1 1) (* 2 (+ 1 2))) ; ⇒ (1 (+ 1 1) (* 2 (+ 1 2)))
- Función list sin elementos o utilizando el símbolo `**()** -> **lista vacía**

Selección primer elemento: función car

(define lista2 '((1 2) 3 4))
(car lista2) ⇒ (1 2)

Selección resto de elementos: función cdr (los devuelve en forma de lista)

El **cdr de una lista con un elemento** devuelve una **lista vacia '()**.
(cdr lista2) ⇒ (3 4)
(cdr '(1)) => '()

Composición de listas: cons

Crea una lista nueva resultante de **añadir un elemento al comienzo** de la lista.

(cons '(1 2) '(1 2 3 4)) ; ⇒ ((1 2) 1 2 3 4)

Composición de listas: append

Crear una lista nueva resultado de **concatenar dos o más** listas.
(define list1 '(1 2 3 4))
(define list2 '(hola como estás))
(append list1 list2) ; ⇒ (1 2 3 4 hola como estás)

Recursión

- El **caso base** define el valor que devuelve la función en el caso elemental en el que no hay que hacer ningún cálculo.
- El **caso general** define una expresión que contiene una llamada a la propia función que estamos definiendo.

*Pasos: **1º** Boceto para encontrar caso **General** (Confiar en que la llamada recursiva siguiente devuelve la solución cuasi completa, a falta de realizar una operación adicional para hallar la solución final. La llamada recursiva siempre trabaja sobre un problema más sencillo.)

2º Encontrar casos **Base**

Ej: Función que sume los números hasta x.

```
(define (suma-hasta x)
      (if (= 0 x)
      0
      (+ (suma-hasta (- x 1)) x)))
```

Recursión y listas

- Un problema que recorra una **lista hasta el final**, finalizará cuando no hayan elementos en la lista, esto es, cuando la lista sea vacia (‘()), o lo que es lo mismo:
  (null? lista) -> #t
- Un problema que recorra una **lista hasta que quede un elemento**, finalizará cuando:
  (null? (cdr lista)) -> #t
- Se evitará anidar múltiples expresiones “if”, en cuyo caso se utilizará la expresión “cond”.
- Cuando un problema recursivo pueda solucionarse usando la expresión “and” u “or”, debemos evitar resolverlo mediante la expresión “if”.

Ej: Función que realice la suma de los elementos de una lista

```
(define (suma-lista lista)
      (if (null? lista)
            0
            (+ (car lista) (suma-lista (cdr lista)))))
```

Tipos de datos compuestos en Scheme

El tipo de dato pareja

Construcción de parejas cons

- Construye un dato compuesto a partir de otros dos datos (que llamaremos **izquierdo y derecho**).
- Los elementos izq. y der. **son evaluados** en la construcción de la pareja.

```
(cons 1 2) ; ⇒ (1 . 2)
(define c (cons 1 2))
```

Construcción de parejas quote

- Los elementos izq. y der. **NO son evaluados** en la construcción de la pareja.
- Definiendo la pareja entre **paréntesis y separando su parte izquierda y derecha con un punto**:

```
'(1 . 2) ; ⇒ (1 . 2)
```

Funciones de acceso car y cdr

- Car devuelve el elemento izquierdo.
- Cdr devuelve el elemento derecho.

```
define c (cons 1 2))
(car c) ; ⇒ 1
(cdr c) ; ⇒ 2
```

Función pair?

- Nos dice si un objeto es atómico o es una pareja:

```
(pair? 3) ; ⇒ #f
(pair? (cons 3 4)) ; ⇒ #t
```

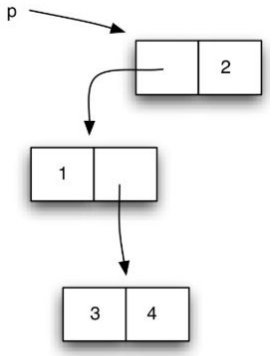
Características

- Pueden contener cualquier tipo de dato (**Scheme es débilmente tipado**)
- **Inmutables**
- **Objetos de primera clase** (asignar, devolver, pasar como arg., guardarse estruc. mayor).
- Tienen **propiedad de clausura de la función cons** (pueden formar parte de otras parejas).

Diagramas caja y puntero (Box & pointer)

- Nos permiten **representar estructuras** de parejas anidadas.

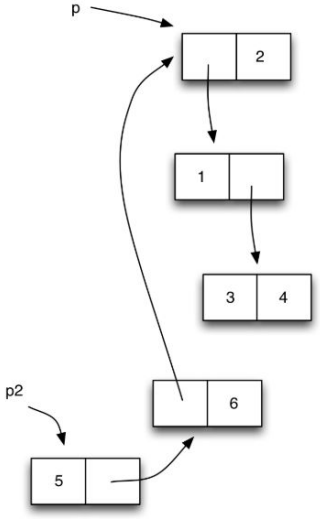
```
(define p (cons (cons 1
                      (cons 3 4))
                2))
```



Si después de haber evaluado la sentencia anterior evaluamos la siguiente:

```
(define p2 (cons 5 (cons p 6)))
```

*** En la pareja que se crea con (cons p 6) se guarda en la parte izquierda **la misma pareja** que hay en p. ***



*** Importante saber **recuperar un** determinado **dato** (pareja o dato atómico) una vez creada la estructura. ***

Escribe las expresiones que devuelven 1 y 4 a partir de p2.

(car (car (car (cdr p2)))) == (caaadr p2) -> 1
(cdr (cdr (car (car (cdr p2))))) == (cdr (cdaadr p2)) -> 4

La función **c????r** es equivalente a realizar los car y cdr por separado (límite de 4 “?”)

Relación entre listas y parejas en Scheme

- Una **lista es**:
  - **Pareja** contiene en parte **izquierda el primer elemento** de la lista y en parte **derecha el resto de la lista**.
  - ' ( ) que denota la **lista vacía**

  Ej:

```
(cons 1 '())
(cons 1
      (cons 2
            (cons 3
                  (cons 4 '()))))
```

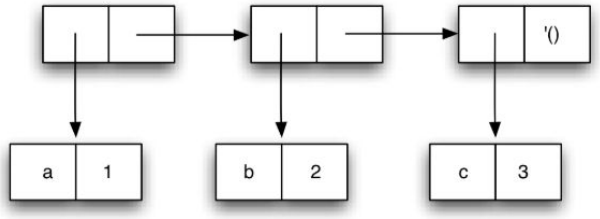
- Una **pareja No** es una **lista vacía**.
- Usamos **list? / symbol? y pair?** para comprobar si un objeto es respectivamente una lista / símbolo o pareja.
- Para saber si un objeto es lista vacía usamos **null?**

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

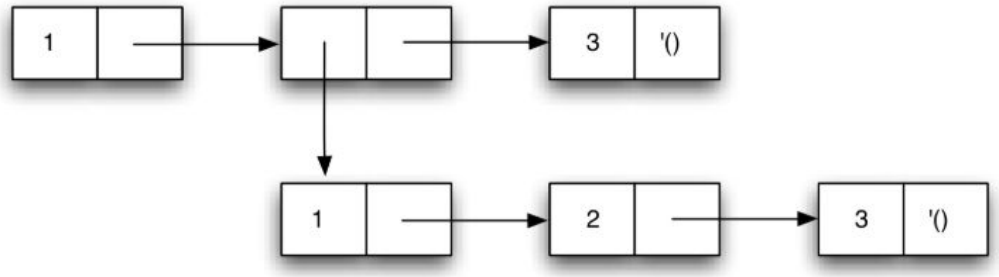
Listas con elementos compuestos

Box & pointer

```
(list (cons 'a 1)
      (cons 'b 2)
      (cons 'c 3)) ; ⇒ ((a . 1) (b . 2) (c . 2))
```



(define lista '(1 (1 2 3) 3)) (define (list 1 (list 1 2 3) 3))



*** El intérprete construye la salida conforme recorre las parejas. ***

```
(define p (cons 1 (cons 2 3)))->    (1 2 . 3)
```

Funciones de alto nivel sobre listas

```
(append '(a (b) c) '((d) e f)) ; ⇒ (a (b) c (d) e f)
(list-ref '(a (b) c d) 2) ; ⇒ c
(length '(a (b (c))) ; ⇒ 2
(reverse '(a b c))  ; ⇒ (c b a)
(list-tail '(a b c d) 2) ; ⇒ (c d)
```

Funciones recursivas que construyen listas

mi-list-ref

```
(define (mi-list-ref lista n)
   (if (= n 0)
      (car lista)
      (mi-list-ref (cdr lista) (- n 1))))
```

mi-list-tail

```
(define (mi-list-tail lista n)
   (if (= n 0)
       lista
       (mi-list-tail (cdr lista) (- n 1))))
```

mi-append

```
(define (mi-append l1 l2)
    (if (null? l1)
        l2
        (cons (car l1)
              (mi-append (cdr l1) l2))))
```

mi-reverse

```
(define (añade-al-final dato lista)
    (append lista (list dato)))
```

```
(define (mi-reverse lista)
    (if (null? lista) '()
    (añade-al-final (car lista) (mi-reverse (cdr lista)))))
```

cuadrados-hasta

```
(define (cuadrados-hasta x)
   (if (= x 1)
      '(1)
      (cons (cuadrado x)
            (cuadrados-hasta (- x 1)))))
```

filtra-pares

```
(define (filtra-pares lista)
   (cond
      ((null? lista) '())
      ((even? (car lista)) (cons (car lista)
       (filtra-pares (cdr lista))))
      (else (filtra-pares (cdr lista)))))
```

Funciones con número variable de argumentos

Funciones primitivas de Scheme, como + o max admiten un número variable de argumentos. ¿Cómo hacerlo en nuestras funciones?

- Un **punto** antes del último parámetro. Los parámetros antes del punto (si existen) tendrán como valores los argumentos usados en la llamada y el **resto de argumentos se pasarán en forma de lista** en el último parámetro.

  (define (funcion-dos-o-mas-args x y . lista-args) <cuerpo>) O
  (define (funcion-dos-o-mas-args . lista-args) <cuerpo>)

  (funcion-dos-o-mas-args 1 2 3 4 5 6)

Función apply: (apply funcion lista)

- Podemos aplicar una función de aridad n a una lista de datos de n datos, haciendo que cada uno de los datos se pasen a la función en orden como parámetros.

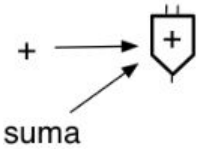
```
Ej1: función aridad 2

(define (suma-cuadrados x y)
    (+ (* x x) (* y y)))

(apply suma-cuadrados '(10 5))
```

```
Ej2: función recursiva aridad n

(define (suma-pareja p1 p2)
  (cons (+ (car p1) (car p2))
        (+ (cdr p1) (cdr p2))))

(define (suma-parejas . parejas)
  (if (null? parejas)
      '(0 . 0)
      (suma-pareja (car parejas) (apply suma-parejas (cdr
parejas)))))

(suma-parejas '(1 . 2) '(3 . 4) '(5 . 6)) ; ⇒ '(9 . 12)
```

Forma especial lambda

- Permite crear **funciones anónimas** en **tiempo de ejecución**.
- Sintaxis: (lambda (<arg1> ... <argn>) <cuerpo>)
- Ejecutar una expresión lambda en el intérprete devuelve un procedimiento.
  (lambda (x) (* x x)) ; ⇒ #<procedure>
- Puede asignarse a un identificador: `(define f (lambda (x) (* x x)))`
  - Y entonces usarlo: (f 3) -> 9
- No es necesario un id. para invocar la función
  - ((lambda (x) (* x x)) 3) => 9

```
*** Usamos procedure? para comprobar si un objeto es una función ***
*** Pueden asignarse funciones ya existentes a nuevos id usando define ***

+ ;                ⇒ <procedure:+>
(define suma +)
(suma 1 2 3 4) ; ⇒ 10
```



Azúcar sintáctico

La forma especial define para definir una función no es más que azúcar sintáctico.

```
(define (<nombre> <args>)
   <cuerpo>)
```

Siempre se convierte en:

```
(define <nombre>
   (lambda (<args>)
      <cuerpo>))
```

Funciones argumentos de otras funciones (*******)

```
(define (aplica f x y)
  (f x y))

(aplica + 2 3) ; ⇒ 5
(aplica * 4 5) ; ⇒ 10
(aplica (lambda (x y) (sqrt (+ (* x x) (* y y)))) 3 4) ; ⇒ 5
```

Generalización

- Pasar funciones como parámetros de otras nos va a permitir diseñar funciones más genéricas.
  Ej: Crea una función que calcule el sumatorio desde a hasta b de f(x):

```
(define (sum-f-x f a b)
   (if (> a b)
      0
      (+ (f a) (sum-f-x f (+ a 1) b))))
```

Siendo f:

```
(define (cubo x)
   (* x x x))

(sum-f-x cubo 1 10) ; ⇒ 3025
```

Funciones que devuelven funciones

- Debemos usar lambda en el cuerpo de una función. Así, cuando se invoca a esta función se evalúa lambda y se devuelve la función resultante.
- La **función que se devuelve** se denomina **clausura** (la fón creada en el ámbito local captura éste ámbito). Y decimos que la **función que ha construido** la clausura es una **función constructora**.
  Ej:

```
(define (construye-sumador k)
      (lambda (x)
            (+ x k)))

(construye-sumador 10) ; ⇒ #<procedure>
(define f (construye-sumador 10))
(f 3) ; ⇒ 13
((construye-sumador 10) 3) ; ⇒ 13
```

** Lambda captura ámbito local, con sus variables y valores (k = 10) **

Funciones en estructuras de datos

```
(define (cuadrado x) (* x x))
(define (suma-1 x) (+ x 1))
(define (doble x) (* x 2))
(define lista (list cuadrado suma-1 doble))
lista ⇒ (#<procedure:cuadrado>  #<procedure:suma-1>  #<procedure:doble>)
((car lista) 10) ; ⇒ 100
```

Funciones de orden superior

- Funciones que toman otras como parámetro o devuelven otra función. Permiten generalizar soluciones con un alto grado de abstracción.
- Permiten hacer código muy conciso y expresivo.

map: (map transforma lista) => lista

- Transforma una lista aplicando a todos sus elementos una función de transformación que se pasa como parámetro.
- La función transforma:
  (transforma elemento) => elemento
  Ej:

```
(define (suma-pareja pareja)
      (+ (car pareja) (cdr pareja)))

(map suma-pareja '((2 . 4) (3 . 6) (5 . 3))) ; ⇒ (6 9 8)
(map (lambda (pareja)
         (+ (car pareja) (cdr pareja)))
            '((2 . 4) (3 . 6) (5 . 3)))   ; ⇒  (6 9 8)
```

- Puede recibir n listas, todas ellas de la misma longitud. En este caso, la función transforma debe recibir n argumentos. map aplica transforma a los elementos cogidos de las n listas y construye así la lista resultante.

  Ej:

```
(map + '(1 2 3) '(10 20 30)) ; ⇒ (11 22 33)
```

filter: (filter predicado lista) => lista

- Toma como parámetro un predicado y una lista y devuelve como resultado los elementos de la lista que cumplen el predicado.
- La función predicado: (predicado elem) => boolean
  Ej:

```
(filter even? '(1 2 3 4 5 6 7 8)) ; ⇒ (2 4 6 8)
```

exists?: (exists? predicado lista) => boolean

- Recibe un predicado y una lista y comprueba si algún elemento de la lista cumple ese predicado.
- La función predicado: (predicado elem) => boolean

Ej:

```
(ormap even? '(1 2 3 4 5 6)) ; ⇒ #t
(ormap (lambda (x)
       (> x 10)) '(1 3 5 8)) ; ⇒ #f
```

for-all?: (for-all? predicado lista) => boolean

- Recibe un predicado y una lista y comprueba que todos los elementos de la lista cumplen ese predicado.
  Ej:

```
(andmap even? '(2 4 6)) ; ⇒ #t
(andmap (lambda (x)
       (> x 10)) '(12 30 50 80)) ; ⇒ #t
```

foldr: (foldr combina base lista) => valor

- Recorre la lista (de **derecha a izquierda**) aplicando la función combina de forma acumulativa a sus elementos y devolviendo un valor como resultado.
- La **función de plegado** (**combina** dato resultado), que recibe un dato de la lista y lo acumula con el otro parámetro resultado (al que debemos dar un valor inicial y es el parámetro base de la función foldr).
  Ej:

```
(define (suma dato resultado) (+ dato resultado))
(foldr suma 0 '(1 2 3)) ; ⇒ 6
```

foldl

- Similar a foldr con la dif. de que el plegado se realiza de **izquierda a derecha**
  Ej:

```
(foldl cons '() '(1 2 3 4)) ; ⇒ (4 3 2 1)
```

Funciones recursivas vs FOS y lambda

Función (suma-n n lista) Recursiva

```
(define (suma-n n lista)
    (if (null? lista)
        '()
        (cons (+ (car lista) n)
              (suma-n n (cdr lista)))))

(suma-n 10 '(1 2 3 4)) ; ⇒ (11 12 13 14)
```

Función (suma-n n lista) FOS

```
(define (suma-n n lista)
    (map (lambda (x) (+ x n)) lista))

(suma-n 10 '(1 2 3 4)) ; ⇒ (11 12 13 14)
```

Composición de FOS

- Las anteriores funciones de orden superior devuelven listas, es muy común componer las llamadas, de forma que la salida de función se utilice como entrada de otra.
  Ej: Implementar una función que sume un número n a todos los elementos de una lista (igual que la anterior) y después que sume todos los elementos resultantes.

  ```
  (define (suma-n-total n lista)
        (foldr + 0
              (map (lambda (x) (+ x n)) lista)))

  (suma-n-total 100 '(1 2 3 4)) ; ⇒ 410
  ```

Bibliografía

Los enunciados de los ejercicios resueletos, y los resúmenes, se han elaborado a partir del material publicado en https://domingogallardo.github.io/ , material del que es propietario el Departamento de Ciencia de la Computación e Inteligencia Artificial de la Universidad de Alicante, Domingo Gallardo, Cristina Pomares, Antonio Botía y Francisco Martínez.