

# PRACTICA 1

---

## Ejercicio 1

- a) Implementa la función (`binario-a-decimal b3 b2 b1 b0`) que reciba 4 bits que representan un número en binario y devuelva el número decimal equivalente.

```
(binario-a-decimal 1 1 1 1) ; => 15
(binario-a-decimal 0 1 1 0) ; => 6
(binario-a-decimal 0 0 1 0) ; => 2
```

**Nota:** recuerda que para realizar esta conversión, se utiliza la siguiente fórmula:

$$n = b_3 * 2^3 + b_2 * 2^2 + b_1 * 2^1 + b_0 * 2^0$$

Para la implementación de la expresión debes utilizar la función `expt`.

- b) Implementa la función (`binario-a-hexadecimal b3 b2 b1 b0`) que reciba 4 bits de un número representado en binario y devuelva el carácter correspondiente a su representación en hexadecimal.

```
(binario-a-hexadecimal 1 1 1 1) ; => #\F
(binario-a-hexadecimal 0 1 1 0) ; => #\6
(binario-a-hexadecimal 1 0 1 0) ; => #\A
```

**Nota:** para realizar esta conversión, como paso intermedio debes pasar primero el número binario a su representación decimal (utilizando la función definida en el apartado anterior) y después a su correspondiente hexadecimal.

Recuerda que la representación hexadecimal de los números decimales del 0 al 9 es el carácter correspondiente a ese número, y que el número decimal 10 se representa con el carácter A, el 11 con el B, y así sucesivamente hasta el 15 que es el F en hexadecimal.

Para la implementación de esta función auxiliar que pasa de decimal a hexadecimal debes usar las funciones `integer->char` y `char->integer`. En la función `char->integer` los caracteres consecutivos están asociados con números consecutivos. Por ejemplo, el entero correspondiente al carácter `#\A` es uno menos que el correspondiente al carácter `#\B`. Los caracteres de números y los de letras no son consecutivos.

```
;-----
; Ejercicio 1: ;
;-----

(define (binario-a-decimal b3 b2 b1 b0)
  (+ (* b3 (expt 2 3))
```

```

(* b2 (expt 2 2))
(* b1 2)
b0))

(check-equal? (binario-a-decimal 1 1 1 1) 15)
(check-equal? (binario-a-decimal 0 1 1 0) 6)
(check-equal? (binario-a-decimal 0 0 1 0) 2)

(define (decimal-a-hexadecimal valor)
(if (< valor 10)
(integer->char (+ (char->integer #\0) valor))
(integer->char (+ (char->integer #\A) (- valor 10)))))

(define (binario-a-hexadecimal b3 b2 b1 b0)
(decimal-a-hexadecimal (binario-a-decimal b3 b2 b1 b0)))

(check-equal? (binario-a-hexadecimal 1 1 1 1) #\F)
(check-equal? (binario-a-hexadecimal 0 1 1 0) #\6)
(check-equal? (binario-a-hexadecimal 1 0 1 0) #\A)

```

## Ejercicio 2

Implementa la función (`menor-de-tres n1 n2 n3`) que reciba tres números como argumento y devuelva el menor de los tres, intentando que el número de condiciones sea mínima.

No debes utilizar la función `min`.

Implementa dos versiones de la función:

- versión 1: usando la forma especial `if`
- versión 2 (llámala `menor-de-tres-v2`): definiendo una función auxiliar (`menor x y`) que devuelva el menor de dos números (deberás usar también la forma especial `if` para implementarla) y construyendo la función `menor-de-tres-v2` como una composición de llamadas a esta función auxiliar.

```

(menor-de-tres 2 8 1) ;; => 1
(menor-de-tres-v2 3 0 3) ;; => 0

```

```

;-----;
; Ejercicio 2: ;
;-----;

(define (menor-de-tres a b c)
(if (< a b)
(if (< a c) a c)
(if (< b c) b c)))

(check-equal? (menor-de-tres 2 8 1) 1)
(check-equal? (menor-de-tres 8 2 1) 1)
(check-equal? (menor-de-tres 2 1 8) 1)

```

```
(check-equal? (menor-de-tres 8 1 2) 1)
(check-equal? (menor-de-tres 1 8 2) 1)
(check-equal? (menor-de-tres 1 2 8) 1)

(define (menor a b)
  (if (< a b) a b))

(define (menor-de-tres-v2 a b c)
  (menor (menor a b) c))

(check-equal? (menor-de-tres-v2 3 0 3) 0)
(check-equal? (menor-de-tres-v2 0 3 3) 0)
(check-equal? (menor-de-tres-v2 3 3 0) 0)
(check-equal? (menor-de-tres-v2 3 5 3) 3)
(check-equal? (menor-de-tres-v2 5 3 3) 3)
(check-equal? (menor-de-tres-v2 3 3 5) 3)
```

### Ejercicio 3

a) Supongamos las definiciones

```
(define (f x y)
  (cons x y))

(define (g x)
  (cons 2 x))
```

Realiza la evaluación paso a paso de la siguiente expresión

```
(f (g (+ 2 1)) (+ 1 1))
```

mediante el **modelo de sustitución**, utilizando tanto el **orden aplicativo** y como el **orden normal**.

Escribe la solución entre comentarios en el propio fichero **.rkt** de la práctica.

b) Supongamos las definiciones

```
(define (f x)
  (/ x 0))

(define (g x y)
  (if (= x 0)
      0
      y))
```

Igual que en el apartado anterior, realiza la evaluación paso a paso de la siguiente expresión

```
(g 0 (f 10))
```

mediante el **modelo de sustitución**, utilizando tanto el **orden aplicativo** y como el **orden normal**. Y escribe la solución entre comentarios en el propio fichero **.rkt** de la práctica.

```
;-----;
; Ejercicio 3: ;
;-----;

;; Orden aplicativo

; (f (g (+ 2 1)) (+ 1 1)) -> R3
; (f (g 3) (+ 1 1)) -> R4
; (f (cons 2 3) (+ 1 1)) -> R3
; (f (2 . 3) (+ 1 1)) -> R3
; (f (2 . 3) 2) -> R4
; (cons (2 . 3) 2) -> R3
; ((2 . 3) . 2)

;; Orden normal

; (f (g (+ 2 1)) (+ 1 1)) -> R4
; (cons (g (+ 2 1)) (+ 1 1)) -> R4
; (cons (cons 2 (+ 2 1)) (+ 1 1)) -> R3
; (cons (cons 2 3) (+ 1 1)) -> R3
; (cons (2 . 3) (+ 1 1)) -> R3
; (cons (2 . 3) 2) -> R3
; ((2 . 3) . 2)

;; Orden aplicativo

; (g 0 (f 10)) -> R4 f
; (g 0 (/ 10 0)) -> R3
; #!error _2ºarg

;; Orden normal

; (g 0 (f 10)) -> R4 g
; (if (= 0 0) 0 (f 10)) -> R3
; (if #t 0 (f 10)) -> Evaluación If
; 0
```

## Ejercicio 4

Implementa la función (**tirada-ganadora t1 t2**) que reciba 2 parejas como argumento, donde cada pareja representa una tirada con 2 dados (contiene dos números). La función debe determinar qué tirada es la

ganadora, teniendo en cuenta que será aquella cuya suma de sus 2 dados esté más próxima al número 7. La función devolverá 1 si **t1** es la ganadora, 2 si **t2** es la ganadora o bien 0 si hay un empate. Este último caso se producirá cuando la diferencia con 7 de ambas tiradas es la misma.

```
(tirada-ganadora (cons 1 3) (cons 1 6)) ; => 2
(tirada-ganadora (cons 1 5) (cons 2 2)) ; => 1
(tirada-ganadora (cons 6 2) (cons 3 3)) ; => 0
```

```
;-----
; Ejercicio 4: ;
;-----

(define (distancia-a-7 t)
  (abs (- 7 (+ (car t) (cdr t)))))

(define (mejor-distancia d1 d2)
  (cond
    ((< d1 d2) 1)
    ((> d1 d2) 2)
    (else 0)))

(define (tirada-ganadora t1 t2)
  (mejor-distancia (distancia-a-7 t1)
    (distancia-a-7 t2)))

(check-equal? (tirada-ganadora (cons 1 3) (cons 1 6)) 2)
(check-equal? (tirada-ganadora (cons 1 5) (cons 2 2)) 1)
(check-equal? (tirada-ganadora (cons 6 2) (cons 3 3)) 0)
```

## Ejercicio 5

Supongamos que queremos programar un juego de cartas. Lo primero que debemos hacer es definir una forma de representar las cartas y funciones que trabajen con esa representación. En este ejercicio vamos a implementar esas funciones.

Representaremos una carta por un símbolo con dos letras: la primera indicará su número o figura y la segunda el palo de la carta.

Por ejemplo:

```
(define tres-de-oro '3O)
(define as-de-copas 'AC)
(define caballo-de-espadas 'CE)
```

Debemos definir la función **carta** que devuelve una pareja con el valor correspondiente a su orden en la baraja española (un número) y el palo de la carta (un símbolo).

```
(carta tres-de-oros) ; => (3 . Oros)
(cartas as-de-copas) ; => (1 . Copas)
(cartas 'RB) ; => (12 . Bastos)
```

Los valores de las cartas de la baraja española son:

```
A (As) => 1
S (Sota) => 10
C (Caballo) => 11
R (Rey) => 12
```

Para realizar el ejercicio debes definir en primer lugar las funciones (`obten-palo char`) y (`obten-valor char`) que devuelven el palo y el valor, dado un carácter. Y debes implementar la función `carta` usando estas dos funciones.

```
(obten-palo #\O) ; => Oros
(obten-palo #\E) ; => Espadas
(obten-valor #\3) ; => 3
(obten-valor #\S) ; => 10
```

!!! Note "Pista" Puedes utilizar las funciones (`symbol->string simbolo`) que convierte un símbolo en una cadena y (`string-ref cadena pos`) que devuelve el carácter de una cadena situado en una determinada posición.

```
;-----
; Ejercicio 5: ;
;-----

(define (obten-palo caracter)
  (cond
    ((equal? #\O caracter) 'Oros)
    ((equal? #\C caracter) 'Copas)
    ((equal? #\E caracter) 'Espadas)
    (else 'Bastos)))

(define (obten-valor caracter)
  (cond
    ((equal? #\A caracter) 1) ; As
    ((equal? #\S caracter) 10) ; Sota
    ((equal? #\C caracter) 11) ; Caballo
    ((equal? #\R caracter) 12) ; Rey
    (else (- (char->integer caracter) (char->integer #\0)))))

(check-equal? (obten-palo #\O) 'Oros)
(check-equal? (obten-palo #\E) 'Espadas)
```

```

(check-equal? (obten-valor #\3) 3)
(check-equal? (obten-valor #\$) 10)

(define (par-caracteres-a-carta pareja)
  (cons (obten-valor (car pareja))
        (obten-palo (cdr pareja)))))

(define (simbolo-a-par-caracteres simbolo)
  (cons (string-ref (symbol->string simbolo) 0)
        (string-ref (symbol->string simbolo) 1)))

(define (carta simbolo)
  (par-caracteres-a-carta
    (simbolo-a-par-caracteres simbolo)))

(check-equal? (carta 'AC) '(1 . Copas))
(check-equal? (carta '20) '(2 . Oros))
(check-equal? (carta 'RB) '(12 . Bastos))

(define tres-de-oros '30)
(define as-de-copas 'AC)
(define caballo-de-espadas 'CE)
(define rey-de-bastos 'RB)

(check-equal? (carta tres-de-oros) '(3 . Oros))
(check-equal? (carta as-de-copas) '(1 . Copas))
(check-equal? (carta caballo-de-espadas) '(11 . Espadas))
(check-equal? (carta rey-de-bastos) '(12 . Bastos))

```

## Ejercicio 6

Define la función **tipo-triangulo** que recibe como parámetro las coordenadas en el plano de los vértices de un triángulo representados con parejas. La función devuelve una cadena con el tipo de triángulo correspondiente: equilátero, isósceles o escaleno.

Recuerda que un triángulo equilátero es aquel cuyos tres lados tienen la misma longitud, el isósceles el que tiene dos lados iguales y el escaleno el que todos sus lados son diferentes.

Ejemplos:

```

(tipo-triangulo (cons 1 1) (cons 1 6) (cons 6 1)) ; => "isósceles"
(tipo-triangulo (cons -2 3) (cons 2 6) (cons 5 3)) ; => "escaleno"
(tipo-triangulo (cons -3 0) (cons 3 0) (cons 0 5.1961)) ; => "equilatero"

```

!!! Note "Nota" Para comparar dos números reales debemos comprobar si la resta entre ambos es menor que una constante **epsilon** que hemos definido. Por ejemplo, **epsilon** puede valer 0.0001.

Puedes usar la siguiente función auxiliar:

```
```racket
(define epsilon 0.0001)

(define (iguales-reales? x y)
  (< (abs (- x y)) epsilon))
```

```

```
;-----;
; Ejercicio 6: ;
;-----;

(define epsilon 0.0001)

(define (iguales-reales? x y)
  (< (abs (- x y)) epsilon))

(define (distancia coor1 coor2)
  (sqrt (+ (expt (- (car coor1) (car coor2)) 2)
              (expt (- (cdr coor1) (cdr coor2)) 2)))))

(define (tipo-triangulo-por-distancias d1 d2 d3)
  (cond
    ((and (iguales-reales? d1 d2)
          (iguales-reales? d1 d3)) "equilatero")
    ((or (iguales-reales? d1 d2)
         (iguales-reales? d2 d3)
         (iguales-reales? d3 d1)) "isósceles")
    (else "escaleno")))

(define (tipo-triangulo coor1 coor2 coor3)
  (tipo-triangulo-por-distancias (distancia coor1 coor2)
                                 (distancia coor2 coor3)
                                 (distancia coor3 coor1)))

  (check-equal? (tipo-triangulo (cons 1 1) (cons 1 6) (cons 6 1)) "isósceles")
  (check-equal? (tipo-triangulo (cons -2 3) (cons 2 6) (cons 5 3)) "escaleno")
  (check-equal? (tipo-triangulo (cons -3 0) (cons 3 0) (cons 0 5.1961)))
  "equilatero")
```

## PRACTICA 2

---

### Ejercicio 1

a.1) Implementa la función recursiva (**minimo lista**) que recibe una lista con números como argumento y devuelve el menor número de la lista. Suponemos listas de 1 o más elementos.

Para la implementación debes usar la función **menor** definida en la práctica anterior.

### !!! Tip "Pista"

Podemos expresar el caso general de la recursión de la siguiente forma:

- > El mínimo de los elementos de una lista es el menor entre el primer elemento de la lista y el mínimo del resto de la lista.

Y el caso base:

- > El mínimo de una lista con un único número es ese número.

### Ejemplos:

```
(minimo '(1 8 6 4 3)) ; => 1
(minimo '(1 -1 3 -6 4)) ; => -6
```

a.2) Vamos a investigar el funcionamiento de la recursión en la función `minimo`. Supongamos la siguiente llamada:

```
(minimo '(1 8 6 4 3)) ; => 1
```

- ¿Qué lista se pasa como parámetro a la primera llamada recursiva a la función?
- ¿Qué devuelve esa llamada recursiva?
- ¿Con qué argumentos se llama a la función `menor` que devuelve el resultado final?

b) Implementa la función recursiva (`concatena lista-chars`) que recibe una lista de caracteres y devuelve la cadena resultante de concatenarlos.

### Ejemplos:

```
(concatena '(\#\H #\o #\l #\a)) ; => "Hola"
(concatena '(\#\S #\c #\h #\e #\m #\e #\space #\m #\o #\l #\a))
; => "Scheme mola"
```

c) Implementa la función (`contiene? cadena char`) que comprueba si una cadena contiene un carácter determinado. Debes usar la función `string->list` e implementar la función auxiliar recursiva (`contiene-lista? lista dato`).

### Ejemplos:

```
(contiene? "Hola" #\o) ; => #t
(contiene? "Esto es una frase" #\space) ; => #t
```

```
(contiene? "Hola" #\h) ; => #f
```

```
;-----;
; Ejercicio 1: ;
;-----;

;-----;
; a.1) ;
;-----;

(define (menor a b)
(if (< a b) a b))

(define (minimo lista)
(if (null? (cdr lista))
(car lista)
(menor (car lista) (minimo (cdr lista)))))

(check-equal? (minimo '(1 8 6 4 3)) 1)
(check-equal? (minimo '(1 -1 3 -6 4)) -6)

;-----;
; a.2) ;
;-----;
; ¿Qué lista se pasa como parámetro a la primera llamada recursiva a la
función?
; (8 6 4 3)
; ¿Qué devuelve esa llamada recursiva?
; 3
; ¿Con qué argumentos se llama a la función menor que devuelve el resultado
final?
; (menor 1 3)

;-----;
; b) ;
;-----;

(define (concatena lista-chars)
(if (null? lista-chars)
""
(string-append (string (car lista-chars))
(concatena (cdr lista-chars)))))

(check-equal? (concatena '#\H #\o #\l #\a) "Hola")
(check-equal? (concatena '#\S #\c #\h #\e #\m #\e #\space #\m #\o #\l #\a))
"Scheme mola"

;-----;
; c) ;
;-----;
```

```
(define (contiene-lista? lista dato)
  (if (null? lista)
      #f
      (or (equal? (car lista) dato)
          (contiene-lista? (cdr lista) dato)))))

(check-false (contiene-lista? '() 'algo))
(check-true (contiene-lista? '("algo" 3 #\A) 3) )
(check-false (contiene-lista? '("algo" 3 #\A) 'algo))

(define (contiene? cadena char)
  (contiene-lista? (string->list cadena) char))

(check-true (contiene? "Hola" #\o))
(check-true (contiene? "Esto es una frase" #\space))
(check-false (contiene? "Hola" #\h))
```

## Ejercicio 2

- a) Implementa la función recursiva (**binario-a-decimal lista-bits**) que reciba una lista de bits que representan un número en binario (el primer elemento será el bit más significativo) y devuelva el número decimal equivalente.

!!! Hint "Pista" Puedes utilizar la función **length**.

```
(binario-a-decimal '(1 1 1 1)) ; => 15
(binario-a-decimal '(1 1 0)) ; => 6
(binario-a-decimal '(1 0)) ; => 2
```

- b) Implementa la función recursiva (**ordenada-creciente? lista-nums**) que recibe como argumento una lista de números y devuelve **#t** si los números de la lista están ordenados de forma creciente o **#f** en caso contrario. Suponemos listas de 1 o más elementos.

```
(ordenada-creciente? '(-1 23 45 59 99)) ; => #t
(ordenada-creciente? '(12 50 -1 293 1000)) ; => #f
(ordenada-creciente? '(3)) ; => #t
```

```
;-----;
; Ejercicio 2: ;
;-----;

;----;
;a) ;
;----;
```

```

(define (binario-a-decimal lista-bits)
  (if (null? lista-bits)
      0
      (+ (* (car lista-bits)
             (expt 2 (- (length lista-bits) 1)))
          (binario-a-decimal (cdr lista-bits)))))

(check-equal? (binario-a-decimal '(1 1 1 1)) 15)
(check-equal? (binario-a-decimal '(1 1 0)) 6)
(check-equal? (binario-a-decimal '(1 0)) 2)

;----;
; b) ;
;----;

(define (ordenada-creciente? lista-nums)
  (or (null? (cdr lista-nums))
      (and (< (car lista-nums)
                (cadr lista-nums))
           (ordenada-creciente? (cdr lista-nums)))))

(check-true (ordenada-creciente? '(-1 23 45 59 99)))
(check-false (ordenada-creciente? '(12 50 -1 293 1000)))
(check-true (ordenada-creciente? '(3)))

;-----;
; Ejercicio 2: ;
;-----;

;----;
; a.1) ;
;----;

(define p1 (list (cons 1 2) (list 3 4)))

(check-equal? p1 '((1 . 2) . ((3 . (4 . ())) . ())))

;----;
; a.2) ;
;----;

(check-equal? (cdar p1) 2)

(check-equal? (cadadr p1) 4)

;----;
; b.1) ;
;----;

(define p2 (cons (list (cons 7 (cons 8 9)) (list 1 2) 3) (cons 10 11)))

(check-equal? p2 '(((7 . (8 . 9)) . ((1 . (2 . ()))) . (3 . ()))) . (10 . 11)))

```

```

;-----;
; b.2) ;
;-----;

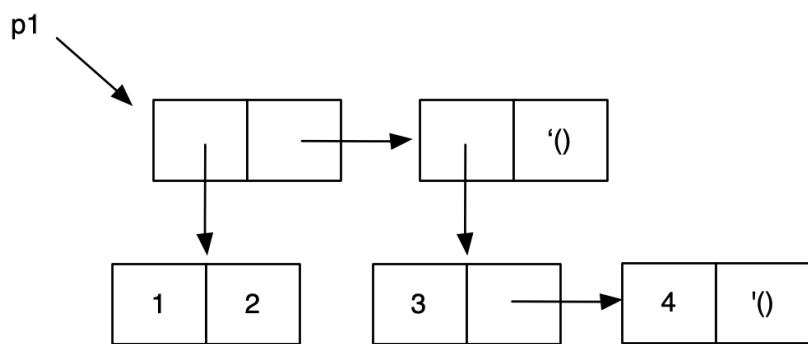
(check-equal? (cddar p2) 9)

(check-equal? (cadadr (car p2)) 2)

```

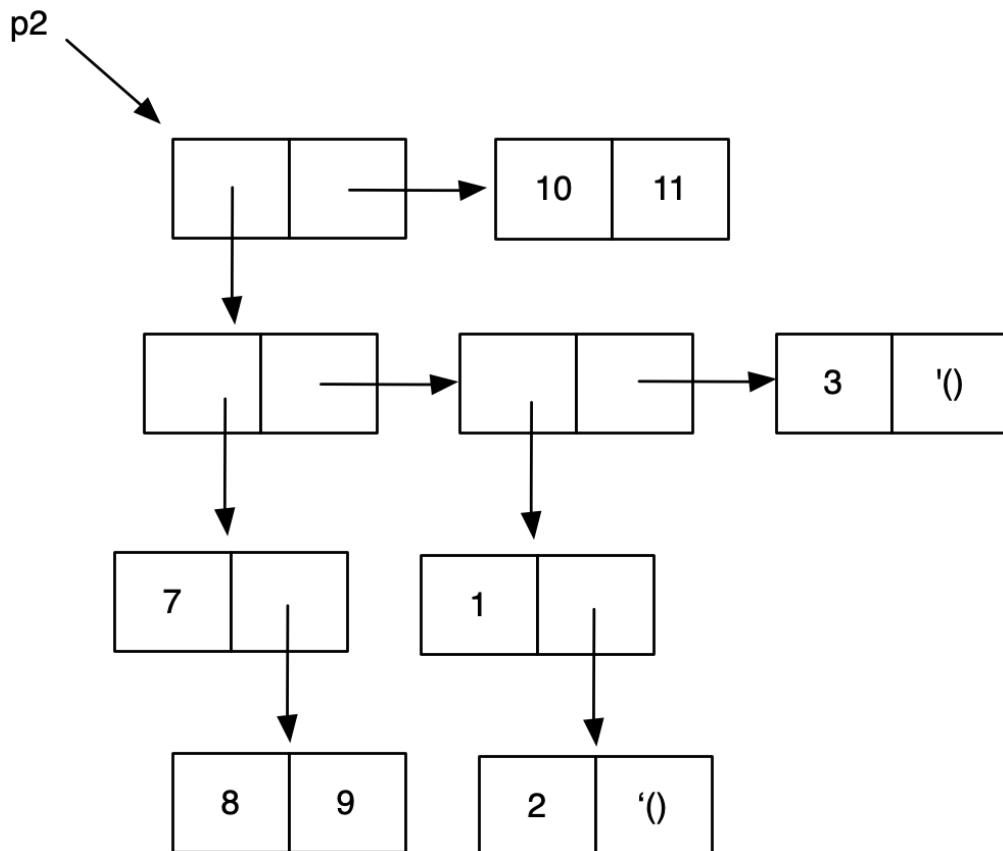
### Ejercicio 3

a.1) Dado el siguiente *box & pointer*, escribe la expresión en Scheme que define **p1** usando el mínimo número de llamadas a **list** y **cons**. No debes utilizar expresiones con **quote**.



a.2) Escribe las expresiones que devuelven 2 y 4 a partir de **p1**.

b.1) Dado el siguiente diagrama caja y puntero, escribe la expresión en Scheme que define **p2** usando el mínimo número de llamadas a **list** y **cons**.



b.2) Escribe las expresiones que devuelven 9 y 2 a partir de p2.

## Ejercicio 4

Vamos a programar una versión simplificada del *blackjack* o 21. Dos jugadores juegan un número de cartas y suman todos sus valores. Gana el que se acerque más al 21 sin pasarse.

Representaremos las cartas como en la práctica 1, y podemos usar las funciones allí definidas para obtener su valor. Suponemos que el valor es el propio de la carta (no seguiremos la regla del juego original en el que las figuras valen 10).

Cada jugador tendrá una lista de cartas y deberá indicar un número n que representa el número de cartas de esa lista con las que se queda.

Tendremos que implementar la función (*blackjack cartas1 n1 cartas2 n2*) que recibe la lista de cartas del jugador 1, el número de cartas que se queda el jugador 1, la lista de cartas del jugador 2 y el número de cartas del jugador 2.

Por ejemplo, supongamos las siguientes cartas del jugador 1 y del jugador 2:

```
(define cartas1 '(30 5E AC 2B 50 5C 4B))  
(define cartas2 '(CE A0 3B AC 2E SC 4C))
```

Supongamos que el jugador 1 se queda con 5 cartas de su lista y el jugador 2 con 3. Las primeras 5 cartas del jugador 1 suman 16 y las 3 primeras cartas del jugador 2 suman 15. Ganaría el jugador 1.

```
(blackjack cartas1 5 cartas2 3) ; ⇒ 1
```

La función *blackjack* devolverá 1 si gana el jugador 1, 2 si gana el jugador 2, 0 si empatan y -1 si los dos jugadores se pasan de 21.

Debes implementar la función *blackjack* y una función auxiliar recursiva que sea la que sume los valores de las n primeras cartas de una lista. Si n es mayor que el número de cartas, devolverá la suma de todas las cartas de la lista.

Puedes definir cualquier otra función auxiliar que necesites.

Otros ejemplos:

```
(blackjack cartas1 5 cartas2 4) ; ⇒ 0  
(blackjack cartas1 5 cartas2 3) ; ⇒ 1  
(blackjack cartas1 3 cartas2 4) ; ⇒ 2  
(blackjack cartas1 7 cartas2 6) ; ⇒ -1
```

```

;-----;
; Ejercicio 4: ;
;-----;

(define (obten-palo caracter)
(cond
  ((equal? #\O caracter) 'Oros)
  ((equal? #\C caracter) 'Copas)
  ((equal? #\E caracter) 'Espadas)
  (else 'Bastos)))

(define (obten-valor caracter)
(cond
  ((equal? #\A caracter) 1) ; As
  ((equal? #\S caracter) 10) ; Sota
  ((equal? #\C caracter) 11) ; Caballo
  ((equal? #\R caracter) 12) ; Rey
  (else (- (char->integer caracter) (char->integer #\0)))))

(define (cadena-a-par-caracteres cadena)
(cons (string-ref cadena 0)
      (string-ref cadena 1)))

(define (par-caracteres-a-carta pareja)
(cons (obten-valor (car pareja))
      (obten-palo (cdr pareja)))))

(define (carta simbolo)
(par-caracteres-a-carta
(cadena-a-par-caracteres
  (symbol->string simbolo)))))

(define cartas1 '(30 5E AC 2B 50 5C 4B))
(define cartas2 '(CE AO 3B AC 2E SC 4C))

(define (suma-valores cartas n)
(if (or (null? cartas) (= n 0))
  0
  (+ (car (carta (car cartas)))
    (suma-valores (cdr cartas)
      (- n 1)))))

(check-equal? (suma-valores cartas1 5) 16)
(check-equal? (suma-valores cartas1 2) 8)
(check-equal? (suma-valores cartas2 3) 15)
(check-equal? (suma-valores cartas2 8) 32)

(define (compara-sumas suma1 suma2)
(cond
  ((and (> suma1 21) (> suma2 21)) -1)
  ((> suma2 21) 1)
  ((> suma1 21) 2)
  ((> suma1 suma2) 1)
  (else 0)))

```

```
((> suma2 suma1) 2)
(else 0))

(check-equal? (compara-sumas 15 10) 1)
(check-equal? (compara-sumas 12 19) 2)
(check-equal? (compara-sumas 23 13) 2)
(check-equal? (compara-sumas 18 22) 1)
(check-equal? (compara-sumas 24 24) -1)
(check-equal? (compara-sumas 16 16) 0)

(define (blackjack cartas1 n1 cartas2 n2)
  (compara-sumas (suma-valores cartas1 n1)
                 (suma-valores cartas2 n2)))

(check-equal? (blackjack cartas1 5 cartas2 4) 0)
(check-equal? (blackjack cartas1 5 cartas2 3) 1)
(check-equal? (blackjack cartas1 3 cartas2 4) 2)
(check-equal? (blackjack cartas1 7 cartas2 6) -1)
```

## Ejercicio 5

a) Implementa las funciones (`suma-izq pareja n`) y (`suma-der pareja n`) definidas de la siguiente forma:

- (`suma-izq pareja n`): devuelve una nueva pareja con la parte izquierda incrementada en `n`.
- (`suma-der pareja n`): devuelve una nueva pareja con la parte derecha incrementada en `n`.

Ejemplos:

```
(suma-izq (cons 10 20) 3) ; => (13 . 20)
(suma-der (cons 10 20) 5) ; => (10 . 25)
```

b) Implementa la función recursiva (`suma-impares-pares lista-num`) que devuelva una pareja cuya parte izquierda sea la suma de los números impares de la lista y la parte derecha la suma de los números pares. Debes utilizar las funciones auxiliares definidas en el apartado anterior. También puedes utilizar las funciones predefinidas `even?` y `odd?`.

Ejemplos:

```
(suma-impares-pares '(3 2 1 4 8 7 6 5)) ; => (16 . 20)
(suma-impares-pares '(3 1 5)) ; => (9 . 0)
```

```
;-----;
; Ejercicio 5: ;
;-----;
```

```

;----;
; a) ;
;----;

(define (suma-izq pareja n)
  (cons (+ (car pareja) n)
        (cdr pareja)))

(define (suma-der pareja n)
  (cons (car pareja)
        (+ (cdr pareja) n)))

(check-equal? (suma-izq (cons 10 20) 3) '(13 . 20))
(check-equal? (suma-der (cons 10 20) 5) '(10 . 25))

;----;
; b) ;
;----;

(define (suma-impares-pares lista-num)
  (cond
    ((null? lista-num)
     (cons 0 0))
    ((even? (car lista-num))
     (suma-der (suma-impares-pares (cdr lista-num)) (car lista-num)))
    (else
     (suma-izq (suma-impares-pares (cdr lista-num)) (car lista-num)))))

(check-equal? (suma-impares-pares '(3 2 1 4 8 7 6 5)) '(16 . 20))
(check-equal? (suma-impares-pares '(3 1 5)) '(9 . 0))

```

## Ejercicio 6

Implementa la función recursiva (`cadena-mayor lista`) que recibe un lista de cadenas y devuelve una pareja con la cadena de mayor longitud y dicha longitud. En el caso de que haya más de una cadena con la máxima longitud, se devolverá la última de ellas que aparezca en la lista.

En el caso en que la lista sea vacía se devolverá la pareja con la cadena vacía y un 0 (la longitud de la cadena vacía).

**Pista:** puedes utilizar la función `string-length`

```

(cadena-mayor '("vamos" "a" "obtener" "la" "cadena" "mayor")) ; => ("obtener" .
7)
(cadena-mayor '("prueba" "con" "maximo" "igual")) ; => ("maximo" . 6)
(cadena-mayor '()) ; => ("" . 0)

```

```

;;-----
;; Ejercicio 6: ;

```

```

;-----;

(define (mejor-cadena pareja cadena)
  (if (> (string-length cadena)
            (cdr pareja))
      (cons cadena (string-length cadena))
      pareja))

(define (cadena-mayor lista)
  (if (null? lista)
      (cons "" 0)
      (mejor-cadena (cadena-mayor (cdr lista)) (car lista)))))

(check-equal? (cadena-mayor '("vamos" "a" "obtener" "la" "cadena" "mayor"))
'("obtener" . 7))
  (check-equal? (cadena-mayor '("prueba" "con" "maximo" "igual")) '("maximo" .
6))
  (check-equal? (cadena-mayor '()) '("") . 0))

```

## PRACTICA 3

---

### Ejercicio 1

a) Implementa la función recursiva (`contiene-prefijo prefijo lista-pal`) que recibe una cadena y una lista de palabras. Devuelve una lista con los booleanos resultantes de comprobar si la cadena es prefijo de cada una de las palabras de la lista.

Debes definir una función auxiliar (`es-prefijo? pal1 pal2`) que compruebe si la palabra 1 es prefijo de la palabra 2.

!!! Hint "Pista" Puedes usar la función (`substring palabra inicio final`) que devuelve la subcadena de la `palabra` que va desde la posición `inicio` hasta la posición `final` (sin incluir).

Ejemplos:

```

(es-prefijo? "ante" "anterior") ; => #t
(contiene-prefijo "ante" '("anterior" "antígona" "antena" "anatema"))
; => (#t #f #t #f)

```

b) Implementa la función recursiva (`inserta-pos dato pos lista`) que recibe un dato, una posición y una lista e inserta el dato en la posición indicada de la lista. Si la posición es 0, el dato se inserta en cabeza. Suponemos que la posición siempre será positiva y menor o igual que la longitud de la lista.

Ejemplos:

```

(inserta-pos 'b 2 '(a a a a)) ; => '(a a b a a)
(inserta-pos 'b 0 '(a a a a)) ; => '(b a a a a)

```

c) Implementa la función recursiva (`inserta-ordenada n lista-ordenada`) que recibe un número y una lista de números ordenados de menor a mayor y devuelve la lista resultante de insertar el número `n` en la posición correcta para que la lista siga estando ordenada.

Ejemplo:

```
(inserta-ordenada 10 '(-8 2 3 11 20)) ; => (-8 2 3 10 11 20)
```

Usando la función anterior `inserta-ordenada` implementa la función recursiva (`ordena lista`) que recibe una lista de números y devuelve una lista ordenada.

Ejemplo:

```
(ordena '(2 -1 100 4 -6)) ; => (-6 -1 2 4 100)
```

## Ejercicio 2

a) Escribe la función (`expande-parejas pareja1 pareja2 ... pareja_n`) que recibe un número variable de argumentos y devuelve una lista donde se han "expandido" las parejas, creando una lista con tantos elementos como el número que indique cada pareja.

La función `expande-parejas` deberá llamar a una función recursiva (`expande-lista lista-parejas`) que trabaje sobre una lista de parejas.

Ejemplo:

```
(expande-parejas '(#t . 3) '("LPP" . 2) '(b . 4))
; => (#t #t #t "LPP" "LPP" b b b b)
```

!!! Hint "Pista" Puedes definir una función auxiliar (`expande-pareja pareja`) que recibe una pareja y devuelve la lista expandida resultante de expandir sólo esa pareja.

b) Implementa la función recursiva (`expande2 lista`). Recibe una lista en la que hay intercalados algunos números enteros positivos. Devuelve la lista original en la que se han expandido los elementos siguientes a los números, tantas veces como indica el número. La lista nunca va a contener dos números consecutivos y siempre va a haber un elemento después de un número.

Ejemplo:

```
(expande2 '(4 clase ua 3 lpp aulario))
; => (clase clase clase clase ua lpp lpp lpp aulario))
```

En el ejemplo, el 4 indica que el siguiente elemento (**clase**) se debe repetir 4 veces en la lista expandida y el 3 indica que el siguiente elemento (**lpp**) se va a repetir 3 veces.

Como en los anteriores ejercicios, te recomendamos implementar alguna función auxiliar.

### Ejercicio 3

a) En matemáticas, el conjunto potencia de un conjunto es el conjunto formado por todos los subconjuntos del conjunto original. Vamos a representar este concepto con listas y diseñar una función recursiva que lo implemente.

Dada una lista de elementos, podemos representar su conjunto potencia como la lista de todas las sublistas posibles formadas con elementos de la lista original.

Ejemplo:

```
(conjunto-potencia '(1 2 3))
; => ((1 2 3) (1 2) (1 3) (1) (2 3) (2) (3) ())
```

Implementa la función recursiva (**conjunto-potencia lista**) que devuelva el conjunto potencia de la lista original.

Algunas pistas:

- El conjunto potencia de una lista vacía es una lista con la lista vacía como único elemento.
- El conjunto potencia de una lista se puede construir uniendo dos listas: la lista resultante de añadir el primer elemento a todas las listas del conjunto potencia sin el primer elemento y el conjunto potencia de la lista sin el primer elemento.

```
(conjunto-potencia '(1 2 3)) = ((1 2 3) (1 2) (1 3) (1)) +
                                ((2 3) (2) (3) ())
```

b) Implementa la función recursiva (**producto-cartesiano lista1 lista2**) que devuelva una lista con todas las parejas resultantes de combinar todos los elementos de la lista 1 con todos los elementos de la lista 2.

Ejemplo:

```
(producto-cartesiano '(1 2) '(1 2 3))
; => ((1 . 1) (1 . 2) (1 . 3) (2 . 1) (2 . 2) (2 . 3))
```

### Ejercicio 4

a) Indica qué devuelven las siguientes expresiones en Scheme. Puede ser que alguna expresión contenga algún error. Indícalo también en ese caso y explica qué tipo de error. Hazlo sin el intérprete, y después

comprueba con el intérprete si tu respuesta era correcta.

```
((lambda (x) (* x x)) 3) ; => ?
((lambda () (+ 6 4))) ; => ?
((lambda (x y) (* x (+ 2 y))) (+ 2 3) 4) ; => ?
((lambda (x y) (* x (+ 2 x))) 5) ; => ?

(define f (lambda (a b) (string-append "****" a b "****")))
(define g f)
(procedure? g) ; => ?
(g "Hola" "Adios") ; => ?
```

b) Hemos visto en teoría que la forma especial **define** para construir funciones es *azucar sintáctico* y que el intérprete de Scheme la convierte en una expresión equivalente usando la forma especial **lambda**.

Escribe cuál sería las expresiones equivalentes, usando la forma especial **lambda** a las siguientes definiciones de funciones:

```
(define (suma-3 x)
  (+ x 3))

(define (factorial x)
  (if (= x 0)
    1
    (* x (factorial (- x 1)))))
```

c) Suponiendo las siguientes definiciones de funciones indica qué devolverían las invocaciones. Puede ser que alguna expresión contenga algún error. Indícalo también en ese caso y explica qué tipo de error.

Hazlo sin el intérprete, y después comprueba con el intérprete si tu respuesta era correcta.

```
(define (doble x)
  (* 2 x))

(define (foo f g x y)
  (f (g x) y))

(define (bar f p x y)
  (if (and (p x) (p y))
    (f x y)
    'error))

(foo + 10 doble 15) ; => ?
(foo doble + 10 15) ; => ?
(foo + doble 10 15) ; => ?
(foo string-append (lambda (x) (string-append "****" x)) "Hola" "Adios") ; => ?
```

```
(bar doble number? 10 15) ; => ?
(bar string-append string? "Hola" "Adios") ; => ?
(bar + number? "Hola" 5) ; => ?
```

## Ejercicio 5

a) Implementa la función recursiva ([crea-baraja lista-parejas](#)) que recibe una lista de parejas con un palo y un valor (ambos caracteres) y devuelve una lista de cartas tal y como se representaban en la práctica anterior.

Por ejemplo:

```
(crea-baraja '((#\u2660 . #\A) (#\u2663 . #\2)
              (#\u2665 . #\3) (#\u2666 . #\R)))
; => (A♠ 2♣ 3♥ R♦)
```

b) Vamos a trabajar con cartas de póker. Los valores posibles son A, 2, 3, ..., 8, 9, 0, J, Q, K (representamos el número 10 con el carácter y el símbolo 0). Y los palos son los símbolos correspondientes a los caracteres UTF #\u2660, #\u2663, #\u2665 y #\u2666: ♠, ♣, ♥ y ♦.

```
(define palos '(\u2660 \u2663 \u2665 \u2666))
(define valores '(#\A #\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 #\J #\Q #\K))
```

Implementa la función ([baraja-poker](#)) que devuelve una lista con todas las cartas de una baraja de póker. No es una función recursiva. Debes usar la función anterior [crea-baraja](#) y la función [producto-cartesiano](#) definida en el ejercicio 3.

```
(baraja-poker)
; => (A♠ 2♠ 3♠ ... K♠ A♣ 2♣ ... K♣ A♥ 2♥ ... K♥ A♦ 2♦ ... Q♦ K♦)
```

No hace falta que hagas ninguna prueba de la función. Basta con que dejes en el código la llamada a la función y al ejecutar el programa verás por pantalla que la baraja es correcta.

c) Implementa la función recursiva ([mezcla lista](#)) que recibe una lista y la devuelve mezclada (sus elementos se han intercambiando en posiciones aleatorias).

```
(mezcla '(1 2 3 4 5 6)) ; => (2 1 6 4 5 3)
```

Llama a la función con la baraja de póker para devuelva la baraja mezclada y la muestre por pantalla.

!!! Hint "Pista" Puedes usar la función [inserta-pos](#) definida anteriormente y la función ([random inicial final](#)) que devuelve un número aleatorio entre el valor inicial (incluido) y el valor final (sin incluir).

## Ejercicio 6

Implementa una función recursiva (`filtra-simbolos lista-simbolos lista-num`) que recibe una lista de símbolos y una lista de números enteros (ambas de la misma longitud) y devuelve una lista de parejas. Cada pareja está formada por el símbolo de la i-ésima posición de `lista-simbolos` y el número entero situado esa posición de `lista-num`, siempre y cuando dicho número se corresponda con la longitud de la cadena correspondiente al símbolo. Puedes utilizar las funciones predefinidas `string-length` y `symbol->string`.

Ejemplo:

```
(filtra-simbolos '(este es un ejercicio de examen) '(2 1 2 9 1 6))
; => ((un . 2) (ejercicio . 9) (examen . 6))
```

## SOLUCION PRACTICA 3

```
;-----
; Ejercicio 1: ;
;-----

;---;
; a) ;
;---;

(define (es-prefijo? pal1 pal2)
  (and (<= (string-length pal1) (string-length pal2))
       (equal? pal1 (substring pal2 0 (string-length pal1)))))

(check-true (es-prefijo? "" "algo"))
(check-true (es-prefijo? "ante" "anterior"))
(check-false (es-prefijo? "antena" "anterior"))
(check-false (es-prefijo? "antena" "ante"))

(define (contiene-prefijo prefijo lista-pal)
  (if (null? lista-pal)
      '()
      (cons (es-prefijo? prefijo (car lista-pal))
            (contiene-prefijo prefijo (cdr lista-pal)))))

(check-equal? (contiene-prefijo "ante" '("anterior" "antígona" "antena"
                                         "anatema"))
              '#t)

;---;
; b) ;
;---;

(define (inserta-pos dato pos lista)
  (if (= pos 0)
      (cons dato lista)
```

```

  (cons (car lista)
        (inserta-pos dato (- pos 1) (cdr lista)))))

(check-equal? (inserta-pos 'b 2 '(a a a a)) '(a a b a a))
(check-equal? (inserta-pos 'b 0 '(a a a a)) '(b a a a a))
(check-equal? (inserta-pos 'b 4 '(a a a a)) '(a a a a b))

;----;
; c) ;
;----;

(define (inserta-ordenada n lista-ordenada)
  (cond
    ((null? lista-ordenada) (list n))
    ((<= n (car lista-ordenada)) (cons n lista-ordenada))
    (else (cons (car lista-ordenada)
                 (inserta-ordenada n (cdr lista-ordenada))))))

(check-equal? (inserta-ordenada 10 '(-8 2 3 11 20)) '(-8 2 3 10 11 20))

(define (ordena lista)
  (if (null? lista)
      '()
      (inserta-ordenada (car lista) (ordena (cdr lista)))))

(check-equal? (ordena '(2 -1 100 4 -6)) '(-6 -1 2 4 100))

;-----;
; Ejercicio 2: ;
;-----;

;----;
; a) ;
;----;

(define (expande-pareja pareja)
  (if (= (cdr pareja) 0)
      '()
      (cons (car pareja)
            (expande-pareja (cons (car pareja)
                                  (- (cdr pareja) 1))))))

(check-equal? (expande-pareja '#f . 5) '#f #f #f #f #f)

(define (expande-lista lista-parejas)
  (if (null? lista-parejas)
      '()
      (append (expande-pareja (car lista-parejas))
              (expande-lista (cdr lista-parejas)))))

(define (expande-parejas . lista-parejas)
  (expande-lista lista-parejas))

```

```
(check-equal? (expande-parejas '(#t . 3) '("LPP" . 2) '(b . 4))
  '(#t #t #t "LPP" "LPP" b b b b))

;----;
; b) ;
;----;

(define (expande2 lista)
  (cond
    ((null? lista) '())
    ((and (integer? (car lista))
          (positive? (car lista))))
     (append (expande-pareja (cons (cadr lista)
                                    (car lista)))
             (expande2 (cddr lista))))
    (else (cons (car lista) (expande2 (cdr lista)))))))

(check-equal? (expande2 '(4 clase ua 3 lpp aulario))
  '(clase clase clase ua lpp lpp lpp aulario))

;-----;
; Ejercicio 3: ;
;-----;

;----;
; a) ;
;----;

(define (combina-elemento elemento lista-conjuntos)
  (if (null? lista-conjuntos)
      '()
      (cons (cons elemento
                  (car lista-conjuntos))
            (combina-elemento elemento
                  (cdr lista-conjuntos)))))

(check-equal? (combina-elemento 'a '((b c) (b) (c) ()))
  '((a b c) (a b) (a c) (a)))

(define (amplia-potencia elemento lista-conjuntos)
  (append (combina-elemento elemento lista-conjuntos)
          lista-conjuntos))

(check-equal? (amplia-potencia 'a '((b c) (b) (c) ()))
  '((a b c) (a b) (a c) (a) (b c) (b) (c) ()))

(define (conjunto-potencia lista)
  (if (null? lista)
      (list '())
      (amplia-potencia (car lista)
                      (conjunto-potencia (cdr lista)))))

(check-equal? (conjunto-potencia '(a b c))
  '(a b c))
```

```

'((a b c) (a b) (a c) (a) (b c) (b) (c) ())

;----;
; b) ;
;----;

(define (crea-parejas elemento lista)
  (if (null? lista)
      '()
      (cons (cons elemento (car lista))
            (crea-parejas elemento (cdr lista)))))

(define (producto-cartesiano lista-1 lista-2)
  (if (null? lista-1)
      '()
      (append (crea-parejas (car lista-1) lista-2)
              (producto-cartesiano (cdr lista-1) lista-2)))))

(check-equal? (producto-cartesiano '(1 2) '(1 2 3))
               '((1 . 1) (1 . 2) (1 . 3) (2 . 1) (2 . 2) (2 . 3)))

;-----;
; Ejercicio 4: ;
;-----;

;----;
; a) ;
;----;

(check-equal? ((lambda (x) (* x x)) 3) 9)
(check-equal? ((lambda () (+ 6 4))) 10)
(check-equal? ((lambda (x y) (* x (+ 2 y))) (+ 2 3) 4) 30)
;((lambda (x y) (* x (+ 2 x))) 5) ; => Le falta un argumento:
(check-equal? ((lambda (x y) (* x (+ 2 x))) 5 0) 35)

(define f (lambda (a b) (string-append "***" a b "***")))
(define g f)
(check-true (procedure? g)) ; => Cierto que g es una función (igual que f):
(check-equal? (g "Hola" "Adios") "***HolaAdios***")

;----;
; b) ;
;----;

(define suma-3
  (lambda (x) (+ x 3)))

(check-equal? (suma-3 2) 5)

(define factorial
  (lambda (x)
    (if (= x 0)
        1

```

```
(* x (factorial (- x 1)))))

(check-equal? (factorial 3) 6)

;----;
; c) ;
;----;

(define (doble x)
  (* 2 x))

(define (foo f g x y)
  (f (g x) y))

(define (bar f p x y)
  (if (and (p x) (p y))
    (f x y)
    'error))

;(foo + 10 doble 15) ; ⇒ 10 debía ser una función de un solo argumento (como
;doble)
;(foo doble + 10 15) ; ⇒ doble es de un solo argumento y debería permitir dos
;(como +)
(check-equal? (foo + doble 10 15) 35)
(check-equal? (foo string-append (lambda (x) (string-append "***" x)) "Hola"
"Adios")
  "***HolaAdios")

;(bar doble number? 10 15)
; ⇒ Si 10 y 15 son números, doble los debería admilar a los dos como argumentos
(check-equal? (bar string-append string? "Hola" "Adios") "HolaAdios")
;Devuelve el símbolo error en lugar de intentar sumar "Hola" y 5:
(check-equal? (bar + number? "Hola" 5) 'error)

;-----;
; Ejercicio 5: ;
;-----;

;----;
; a) ;
;----;

(define (crea-carta pareja)
  (string->symbol (string (cdr pareja) (car pareja)))))

(check-equal? (crea-carta '(\u2665 . \Q)) 'Q♥)

(define (crea-baraja lista-parejas)
  (if (null? lista-parejas)
    '()
    (cons (crea-carta (car lista-parejas))
      (crea-baraja (cdr lista-parejas)))))
```

```

(check-equal? (crea-baraja '((#\u2660 . #\A)
                            (#\u2663 . #\2)
                            (#\u2665 . #\3)
                            (#\u2666 . #\R)))
               '(A♣ 2♣ 3♥ R♦))

;----;
; b) ;
;----;

(define palos '(\u2660 \u2663 \u2665 \u2666))
(define valores '(\A \2 \3 \4 \5 \6 \7 \8 \9 \0 \J \Q \K))

(define (baraja-poker)
  (crea-baraja
    (producto-cartesiano palos valores)))

(baraja-poker) ; ⇒ (A♣ 2♣ 3♣ ... K♣ A♣ 2♣ ... K♣ A♥ 2♥ ... K♥ A♦ 2♦ ... Q♦ K♦)

;----;
; c) ;
;----;

(define (mezcla lista)
  (if (null? lista)
      '()
      (inserta-pos (car lista)
                   (random 0 (length lista))
                   (mezcla (cdr lista)))))

(mezcla '(1 2 3 4 5 6))
(mezcla (baraja-poker))

;-----;
; Ejercicio 6: ;
;-----;

(define (filtra-simbolos lista-simbolos lista-num)
  (cond
    ((or (null? lista-simbolos) (null? lista-num)) '())
    ((= (string-length (symbol->string (car lista-simbolos)))
        (car lista-num))
     (cons (cons (car lista-simbolos)
                 (car lista-num))
           (filtra-simbolos (cdr lista-simbolos)
                           (cdr lista-num))))
    (else (filtra-simbolos (cdr lista-simbolos)
                           (cdr lista-num)))))

(check-equal? (filtra-simbolos '(este es un ejercicio de examen) '(2 1 2 9 1 6))
              '((un . 2) (ejercicio . 9) (examen . 6)))

```

# PRACTICA 4

---

## Ejercicio 1

a) Indica qué devuelven las siguientes expresiones, sin utilizar el intérprete. Comprueba después si has acertado.

```
(map (lambda (x)
  (cond
    ((symbol? x) (symbol->string x))
    ((number? x) (number->string x))
    ((boolean? x) (if x "#t" "#f"))
    (else "desconocido"))) '(1 #t hola #f (1 . 2))) ; => ?

(filter (lambda (x)
  (equal? (string-ref (symbol->string x) 1) #\a)) '(alicante barcelona
madrid almería)) ; => ?

(foldl (lambda (dato resultado)
  (string-append
    (symbol->string (car dato))
    (symbol->string (cdr dato))
    resultado)) "" '((a . b) (hola . adios) (una . pareja))) ; => ?

(foldr (lambda (dato resultado)
  (cons (+ (car resultado) dato)
    (+ (cdr resultado) 1))) '(0 . 0) '(1 1 2 2 3 3)) ; => ?
```

b) Sin utilizar el intérprete DrRacket, rellena los siguientes huecos para obtener el resultado esperado. Después usa el intérprete para comprobar si has acertado.

```
; Los siguientes ejercicios utilizan esta definición de lista

(define lista '((2 . 7) (3 . 5) (10 . 4) (5 . 5)))

; Queremos obtener una lista donde cada número es la suma de las
; parejas que son pares

(filter _____
  (_____ (lambda (x) (+ (car x)
    (cdr x)))
  lista))
; => (8 14 10)

; Queremos obtener una lista de parejas invertidas donde la "nueva"
; parte izquierda es mayor que la derecha.
```

```
(filter _____
         (map _____ lista))
; => ((7 . 2) (5 . 3))

; Queremos obtener una lista cuyos elementos son las partes izquierda
; de aquellas parejas cuya suma sea par.

(foldr _____ '()
         (_____ (lambda (x) (even? (+ (car x) (cdr x)))) lista))
; => (3 10 5)
```

c) Rellena los siguientes huecos **con una única expresión**. Comprueba con el intérprete si lo has hecho correctamente.

```
(define (f x) (lambda (y z) (string-append y z x)))
(define g (f "a"))
(check-equal? _____ "claselppa")

(define (f x) (lambda (y z) (list y x z)))

(check-equal? (g "hola" "clase") (list "hola" "lpp" "clase"))

(define (f g) (lambda(z x) (g z x)))
(check-equal? _____ '(3 . 4))
```

## Ejercicio 2

Implementa utilizando funciones de orden superior las funciones (**crea-baraja lista-parejas**) y (**expande-lista lista-parejas**) de la práctica 3. Para la implementación de **expande-lista** debes utilizar la función **expande-pareja** usada también en la práctica 3.

```
(crea-baraja '((#\u2660 . #\A) (#\u2663 . #\2)
               (#\u2665 . #\3) (#\u2666 . #\R)))
; => (A♠ 2♣ 3♥ R♦)

(expande-lista '((#t . 3) ("LPP" . 2) (b . 4)))
; => '(#t #t #t "LPP" "LPP" b b b b)
```

## Ejercicio 3

a) Implementa usando funciones de orden superior la función (**suma-n-izq n lista-parejas**) que recibe una lista de parejas y devuelve otra lista a la que hemos sumado **n** a todas las partes izquierdas.

Ejemplo

```
(suma-n-izq 10 '((1 . 3) (0 . 9) (5 . 8) (4 . 1)))
; => ((11 . 3) (10 . 9) (15 . 8) (14 . 1))
```

- b) Implementa usando funciones de orden superior la función (`aplica-2 func lista-parejas`) que recibe una función de dos argumentos y una lista de parejas y devuelve una lista con el resultado de aplicar esa función a los elementos izquierdo y derecho de cada pareja.

Ejemplo:

```
(aplica-2 + '((2 . 3) (1 . -1) (5 . 4)))
; => (5 0 9)
(aplica-2 (lambda (x y)
  (if (even? x)
      y
      (* y -1))) '((2 . 3) (1 . 3) (5 . 4) (8 . 10)))
; => (3 -3 -4 10)
```

- c) Implementa la función (`filtra-simbolos lista-simbolos lista-num`) de la práctica 3, usando una composición de funciones en las que se use `map`.

#### Ejercicio 4

- a) La función de Racket (`index-of lista dato`) devuelve la posición de un dato en una lista o `#f` si el dato no está en la lista. Si el dato está repetido en la lista devuelve la posición de su primera aparición.

Implementa la función `mi-index-of` que haga lo mismo, usando funciones de orden superior. Puedes usar también alguna función auxiliar.

!!! Hint "Pista" Puedes utilizar la función `foldl` para recorrer la lista de izquierda a derecha buscando el dato. Puedes usar como resultado del `foldl` una pareja en cuya parte derecha vayamos calculando la posición y en la parte izquierda haya un booleano que indique si hemos encontrado o no el dato.

Ejemplos:

```
(mi-index-of '(a b c d c) 'c) ; => 2
(mi-index-of '(1 2 3 4 5) 10) ; => #f
```

- b) Completa la definición de la siguiente función de orden superior (`busca-mayor mayor? lista`) que busca el mayor elemento de una lista. Recibe un predicado `mayor?` que compara dos elementos de la lista y devuelve `#t` o `#f` dependiendo de si el primero es mayor que el segundo.

```
(define (busca-mayor mayor? lista)
  (foldl _____ (car lista) (cdr lista)))
```

Escribe algunos `check-equal?` en los que compruebes el funcionamiento de `busca-mayor`, utilizando funciones `mayor?` distintas.

c) Define la función (`posicion-mayor mayor? lista`) que devuelva la posición del mayor elemento de la lista utilizando las dos funciones anteriores.

## Ejercicio 5

a) Supongamos que vamos a representar una mano de cartas como una lista de cartas. Podemos entonces representar un juego de  $n$  manos como una lista de  $n$  listas.

Por ejemplo, la siguiente lista representaría un juego con 3 manos de 5 cartas:

```
((K♦ J♥ 2♥ 2♠ 8♥) (A♥ 3♠ 5♦ 3♣ J♦) (3♦ Q♥ 0♠ 2♣ 9♦))
```

Para construir este juego vamos a necesitar una función auxiliar que vaya construyendo las manos, añadiendo una carta a la primera mano de la lista si ésta no tiene todas las cartas necesarias.

Tienes que definir la función (`añade-carta carta n-cartas manos`) que recibe una carta, un número que representa el número de cartas que deben tener las manos y una lista de manos, cuya primera mano puede estar incompleta. En ese caso, la función añadirá la carta a esa primera mano de la lista. Si la primera mano está completa, o la lista de manos está vacía, se deberá añadir una nueva mano a la lista, con la única carta que se pasa como parámetro.

Ejemplos:

```
(añade-carta 'K♦ 3 '()) ; ⇒ ((K♦))
(añade-carta 'J♥ 3 '((2♥ 2♠) (5♦ 3♣ J♦))) ; ⇒ ((J♥ 2♥ 2♠) (5♦ 3♣ J♦))
(añade-carta '3♠ 3 '((5♦ 3♣ J♦))) ; ⇒ ((3♠) (5♦ 3♣ J♦))
```

Una vez definida la función anterior, y utilizando funciones de orden superior, debes implementar la función (`reparte n-manos n-cartas baraja`) que devuelva un juego con  $n$  manos de  $n$  cartas sacadas la parte de arriba de una baraja inicial. Puedes utilizar las funciones `baraja-poker` y `mezcla` de la práctica anterior para crear la baraja inicial.

Ejemplo:

```
(define baraja (mezcla (baraja-poker)))
baraja ; ⇒ (Q♠ 9♥ 4♦ K♣ 7♥ J♦ 5♣ 4♦ 5♦ 6♦ 8♦ A♦ Q♦ 0♦ 7♦ 3♣ Q♥ ...)
(reparte 3 5 baraja)
; ⇒ ((7♦ 0♦ Q♦ A♦ 8♦) (6♦ 5♦ 4♦ 5♣ J♦) (7♥ K♣ 4♦ 9♥ Q♦))
```

b) La siguiente función devuelve el valor de una carta:

```
(define (valor-carta carta orden)
  (+ 1 (index-of orden (string-ref (symbol->string carta) 0))))
```

El parámetro **orden** es una lista de todos los caracteres que representan los posibles valores de una carta, ordenados de menor a mayor.

Por ejemplo:

```
(define orden '(#\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 #\J #\Q #\K #\A))
(valor-carta 'A♦ orden) ; => 13
(valor-carta 'J♥ orden) ; => 10
(valor-carta '2♦ orden) ; => 1
```

Implementa, utilizando funciones de orden superior y funciones definidas anteriormente en esta práctica, la función (**mano-ganadora lista-manos**) que recibe una lista de manos y devuelve la posición de la mano ganadora utilizando la valoración del póker. La mano ganadora es la que tiene una carta más alta. Si hay empate, deberás devolver la posición de la primera mano que participa en el empate.

!!! Note "Pista" Puedes definir una función de un único parámetro que devuelve el valor de una carta usando el orden definido por el póker usando la siguiente expresión lambda:

```
```racket
(lambda (carta)
  (valor-carta carta
    '(#\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 #\J #\Q #\K #\A)))
```

```

Puedes usar cualquier función definida anteriormente.

```
(define lista-manos (reparte 3 5 (mezcla (baraja-poker))))
; lista-manos => ((9♦ 2♦ K♥ 0♦ 7♥) (6♦ 4♦ 7♣ 5♥ 4♦) (0♣ 4♣ 5♣ 3♥ J♥))
(mano-ganadora lista-manos) ; => 0
```

## SOLUCION PRACTICA 4

```
;-----
; Ejercicio 1: ;
;-----

;---;
; a) ;
;---;
```

```

(check-equal? (map (lambda (x)
                           (cond
                               ((symbol? x) (symbol->string x))
                               ((number? x) (number->string x))
                               ((boolean? x) (if x "#t" "#f"))
                               (else "desconocido")))
                           '(1 #t hola #f (1 . 2)))
                           ("1" "#t" "hola" "#f" "desconocido"))

(check-equal? (filter (lambda (x)
                           (equal? (string-ref (symbol->string x) 1) #\a))
                           '(alicante barcelona madrid almería))
                           '(barcelona madrid))

(check-equal? (foldl (lambda (dato resultado)
                           (string-append
                               (symbol->string (car dato))
                               (symbol->string (cdr dato))
                               resultado)) "" '((a . b) (hola . adios) (una .
pareja)))
                           "unaparejaholaadiosab")

(check-equal? (foldr (lambda (dato resultado)
                           (cons (+ (car resultado) dato)
                                 (+ (cdr resultado) 1))) '(0 . 0) '(1 1 2 2 3 3))
                           '(12 . 6))

;----;
; b) ;
;----;

(define lista '((2 . 7) (3 . 5) (10 . 4) (5 . 5)))

(check-equal? (filter even?
                           (map (lambda (x) (+ (car x)
                                             (cdr x)))
                           lista))
                           '(8 14 10))

(check-equal? (filter (lambda (x) (> (car x) (cdr x)))
                           (map (lambda (x) (cons (cdr x) (car x))) lista))
                           '((7 . 2) (5 . 3)))

(check-equal? (foldr (lambda (d r) (cons (car d) r)) '()
                           (filter (lambda (x) (even? (+ (car x) (cdr x)))) lista))
                           '(3 10 5))

;----;
; b) ;
;----;

(define (f1 x) (lambda (y z) (string-append y z x)))
(define g1 (f1 "a"))
(check-equal? (g1 "clase" "lpp") "claselppa")

```

```
(define (f2 x) (lambda (y z) (list y x z)))
(define g2 (f2 "lpp"))
(check-equal? (g2 "hola""clase") (list "hola" "lpp" "clase"))

(define (f g) (lambda(z x) (g z x)))
(check-equal? ((f cons) 3 4) '(3 . 4))

;-----
; Ejercicio 2: ;
;-----;

(define (crea-carta pareja)
(string->symbol (string (cdr pareja) (car pareja)))))

(define (crea-baraja lista-parejas)
(map crea-carta lista-parejas))

(check-equal? (crea-baraja '((#\u2660 . #\A) (#\u2663 . #\2)
                             (#\u2665 . #\3) (#\u2666 . #\R)))
               '(A♣ 2♣ 3♥ R♦))

(define (expande-pareja pareja)
(if (= (cdr pareja) 0)
  '()
  (cons (car pareja)
        (expande-pareja (cons (car pareja)
                               (- (cdr pareja) 1))))))

(define (expande-lista lista-parejas)
(foldr append '() (map expande-pareja lista-parejas)))

(check-equal? (expande-lista '((#t . 3) ("LPP" . 2) (b . 4)))
               '(#t #t #t "LPP" "LPP" b b b b))

;-----
; Ejercicio 3: ;
;-----;

;----;
;a) ;
;----;

(define (suma-n-izq n lista-parejas)
(map (lambda (pareja)
  (cons (+ (car pareja) n)
        (cdr pareja)))
lista-parejas))

(check-equal? (suma-n-izq 10 '((1 . 3) (0 . 9) (5 . 8) (4 . 1)))
               '((11 . 3) (10 . 9) (15 . 8) (14 . 1)))

;----;
```

```

; b) ;
;----;

(define (aplica-2 func lista-parejas)
  (map (lambda (pareja)
          (func (car pareja) (cdr pareja)))
        lista-parejas))

(check-equal? (aplica-2 + '((2 . 3) (1 . -1) (5 . 4))) '(5 0 9))
(check-equal? (aplica-2 (lambda (x y)
                           (if (even? x)
                               y
                               (* y -1))) '((2 . 3) (1 . 3) (5 . 4) (8 . 10)))
               '(3 -3 -4 10))

;----;
; c) ;
;----;

(define (filtra-simbolos lista-simbolos lista-num)
  (filter (lambda (pareja)
            (= (string-length (symbol->string (car pareja))) (cdr pareja)))
          (map cons lista-simbolos lista-num)))

(check-equal? (filtra-simbolos '(este es un ejercicio de examen) '(2 1 2 9 1
6))
               '((un . 2) (ejercicio . 9) (examen . 6)))

;-----;
; Ejercicio 4: ;
;-----;

;----;
; a) ;
;----;

(define (mi-index-of lista dato)
  ; La expresión lambda crea una función. La llamamos con la
  ; pareja resultante de foldl. De esta forma evitamos construir
  ; una función auxiliar.
  ; También estaría bien definir una función auxiliar a la que
  ; se le pase la pareja resultante de foldl.
  ((lambda (pareja)
     (if (car pareja)
         (cdr pareja)
         #f))
   (foldl (lambda (elemento resultado)
            (cond
              ((car resultado) resultado)
              ((equal? dato elemento) (cons #t (cdr resultado)))
              (else (cons #f (+ (cdr resultado) 1))))))
         (cons #f 0)
         lista)))

```

```

(check-equal? (mi-index-of '(a b c d c) 'c) 2)
(check-equal? (mi-index-of '(1 2 3 4 5) 10) #f)

;----;
; b) ;
;----;

(define (busca-mayor mayor? lista)
  (foldl (lambda (dato resultado)
    (if (mayor? dato resultado)
        dato
        resultado))
    (car lista) (cdr lista)))

(check-equal? (busca-mayor > '(8 3 4 1 9 6 2)) 9)
(check-equal? (busca-mayor char>? '(#\e #\u #\i)) #\u)
(check-equal? (busca-mayor string>? '("hola" "adios")) "hola")
(check-equal? (busca-mayor (lambda (x y)
  (> (string-length x)
      (string-length y)))
  '("hola" "adios")) "adios")

;----;
; c) ;
;----;

(define (posicion-mayor mayor? lista)
  (mi-index-of lista (busca-mayor mayor? lista)))

(check-equal? (posicion-mayor > '(8 3 4 1 9 6 2)) 4)
(check-equal? (posicion-mayor char>? '(#\e #\u #\i)) 1)
(check-equal? (posicion-mayor string>? '("hola" "adios")) 0)

;-----;
; Ejercicio 4: ;
;-----;

;----;
; a) ;
;----;

; Funciones de la práctica anterior
; -----
;

(define (crea-parejas elemento lista)
  (if (null? lista)
      '()
      (cons (cons elemento (car lista))
            (crea-parejas elemento (cdr lista)))))

(define (producto-cartesiano lista-1 lista-2)

```

```

(if (null? lista-1)
  '()
  (append (crea-parejas (car lista-1) lista-2)
          (producto-cartesiano (cdr lista-1) lista-2)))))

(define palos '(\u2660 \u2663 \u2665 \u2666))
(define valores '(\u0410 \u0422 \u0433 \u0444 \u0455 \u0466 \u0477 \u0488 \u0499 \u0400 \u041a \u041c))

(define (baraja-poker)
(crea-baraja
(producto-cartesiano palos valores)))

(define (inserta-pos dato pos lista)
(if (= pos 0)
  (cons dato lista)
  (cons (car lista)
    (inserta-pos dato (- pos 1) (cdr lista)))))

(define (mezcla lista)
(if (null? lista)
  '()
  (inserta-pos (car lista)
    (random 0 (length lista))
    (mezcla (cdr lista)))))

; Comentar (random-seed ...), o fijar otra semilla, para provocar ;
; otros ejemplos distintos a los que aparecen en los check-equal? ;
(random-seed 0)
;

(define baraja (mezcla (baraja-poker)))
;baraja ; => (4\ 0\ 7\ 5\ Q\ 4\ Q\ 8\ 8\ 3\ 2\ 9\ J\ 4\ 5\ 9\ ...
(check-equal? baraja '(4\ 0\ 7\ 5\ Q\ 4\ Q\ 8\ 8\ 3\ 2\ 9\
                     J\ 4\ 5\ 9\ K\ 8\ A\ 4\ 3\ 6\ 6\ 2\ K\
                     J\ 6\ 7\ K\ 6\ Q\ 0\ 2\ 5\ 2\ K\ 9\ 3\
                     A\ A\ J\ 0\ 3\ A\ 7\ Q\ 5\ 8\ 0\ J\ 9\))

; fin de funciones de la práctica anterior
; -----
;

(define (añade-carta carta n-cartas manos)
(cond
  ((null? manos) (list (list carta)))
  ((< (length (car manos)) n-cartas) (cons (cons carta (car manos)) (cdr
manos)))
  (else (cons (list carta) manos)))

(check-equal? (añade-carta 'K\ 3 '() '((K\)))
  (check-equal? (añade-carta 'J\ 3 '((2\ 2\)) (5\ 3\ J\)) '((J\ 2\ 2\)) (5\ 3\ J\))
  (check-equal? (añade-carta '3\ 3 '((5\ 3\ J\))) '((3\)) (5\ 3\ J\)))

```

```

(define (reparte n-manos n-cartas baraja)
  (foldl (lambda (dato resultado)
    (if (or (< (length resultado) n-manos)
            (< (length (car resultado)) n-cartas))
        (añade-carta dato n-cartas resultado)
        resultado)
      '()
      baraja))

;(reparte 3 5 baraja) ; => ((4♦ J♦ 9♠ 2♣ 3♠) (8♥ 8♦ Q♠ 4♣ Q♥) (5♦ 7♦ 7♣ 0♣ 4♥))
(check-equal? (reparte 3 5 baraja) '((4♦ J♦ 9♠ 2♣ 3♠)
                                         (8♥ 8♦ Q♠ 4♣ Q♥)
                                         (5♦ 7♦ 7♣ 0♣ 4♥)))

;----;
; b) ;
;----;

(define orden '#\2 #\3 #\4 #\5 #\6 #\7 #\8 #\9 #\0 #\J #\Q #\K #\A))

(define (valor-carta carta orden)
  (+ 1 (index-of orden (string-ref (symbol->string carta) 0)))))

(define (mano-ganadora lista-manos)
  (posicion-mayor > (map (lambda (mano)
    (busca-mayor > (map (lambda (carta)
      (valor-carta carta orden))
      mano)))
    lista-manos)))

(define lista-manos (reparte 3 5 baraja))
;lista-manos ; => ((4♦ J♦ 9♠ 2♣ 3♠) (8♥ 8♦ Q♠ 4♣ Q♥) (5♦ 7♦ 7♣ 0♣ 4♥))
(check-equal? (mano-ganadora lista-manos) 1)

```

## PRACTICA 5

---

### Ejercicio 1

a) Implementa una versión recursiva iterativa de la función (`concat lista`) que toma como argumento una lista de cadenas y devuelve una cadena resultante de concatenar todas las palabras de la lista.

La función `concat` deberá llamar a la función `concat-iter` que es la que implementa propiamente la versión iterativa usando recursión por la cola.

Ejemplo:

```
(concat '("hola" "y" "adiós")) ; => "holayadiós"
(concat-iter '("hola" "y" "adiós") "") ; => "holayadiós"
```

b) Define utilizando recursión por la cola la función (`min-max lista`) que recibe una lista numérica y devuelve una pareja con el mínimo y el máximo de sus elementos.

Ejemplo:

```
(min-max '(2 5 9 12 5 0 4)) ; => (0 . 12)
(min-max-iter '(5 9 12 5 0 4) (cons 2 2)) ; => (0 . 12)
```

## Ejercicio 2

a) Implementa utilizando recursión por la cola las funciones `expande-pareja` y `expande-parejas` de la práctica 3.

Ejemplo:

```
(expande-pareja (cons 'a 4)) ; => (a a a a)
(expande-parejas '(#t . 3) '("LPP" . 2) '(b . 4))
; => (#t #t #t "LPP" "LPP" b b b b)
```

b) Implementa utilizando recursión por la cola la función (`rotar k lista`) que mueve `k` elementos de la cabeza de la lista al final. **No es necesario utilizar una función iterativa auxiliar**, puedes hacer que la propia función `rotar` sea iterativa usando el parámetro `lista` como el parámetro donde acumular el resultado.

Ejemplo:

```
(rotar 4 '(a b c d e f g)) ; => (e f g a b c d)
```

## Ejercicio 3

a) Implementa utilizando recursión por la cola la función `mi-foldl` que haga lo mismo que la función de orden superior `foldl`.

```
(mi-foldl string-append "****" '("hola" "que" "tal")) => "talquehola****"
(mi-foldl cons '() '(1 2 3 4)) ; => (4 3 2 1)
```

b) Implementa una versión con recursión por la cola del predicado (`prefijo-lista? lista1 lista2`) que comprueba si la primera lista es prefijo de la segunda. Suponemos que siempre la primera lista será más pequeña que la segunda.

Ejemplos:

```
(prefijo-lista? '(a b c) '(a b c d e)) => #t
(prefijo-lista? '(b c) '(a b c d e)) => #f
```

## Ejercicio 4

Realiza una implementación que utilice la [técnica de la memoization](#) del algoritmo que devuelve la serie de [Pascal](#).

```
(define diccionario (crea-diccionario))
(pascal-memo 8 4 diccionario) ; => 70
(pascal-memo 40 20 diccionario) ; => 137846528820
```

## Ejercicio 5

a) Usando gráficos de tortuga implementa la figura recursiva conocida como *curva de Koch*. Debes definir una función recursiva ([koch nivel trazo](#)) que dibuje una curva de Koch de nivel [nivel](#) y de longitud [trazo](#).

Como pista, para dibujar una curva de Koch de nivel  $n$  y longitud  $l$ , se deberán dibujar 4 curvas de Koch de nivel  $n-1$  y longitud  $l/3$ . En estas 4 curvas consecutivas, el ángulo de inclinación de la segunda curva con respecto a la primera es de 60 grados.

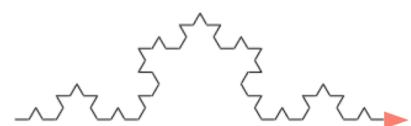
Puedes ver ejemplos de las curvas de nivel 1, 2 y 3 en las siguientes figuras:



(koch 1 600)

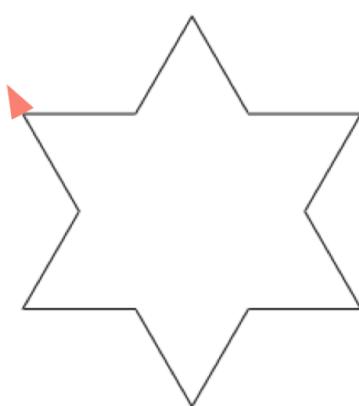


(koch 2 600)

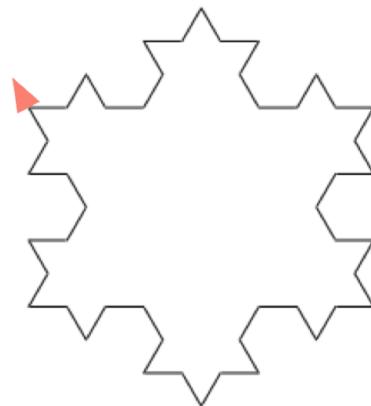


(koch 3 600)

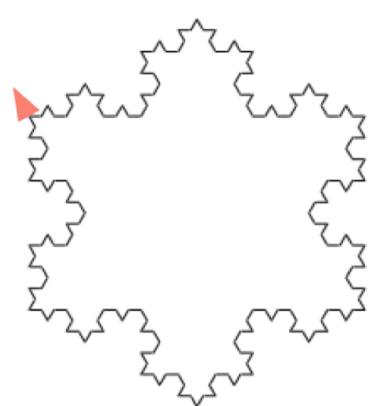
b) Implementa la función ([copo-nieve nivel trazo](#)) que, usando la función anterior, dibuje el [copo de nieve de Koch](#) que puedes ver en los siguientes ejemplos.



(copo-nieve 1 200)



(copo-nieve 2 200)

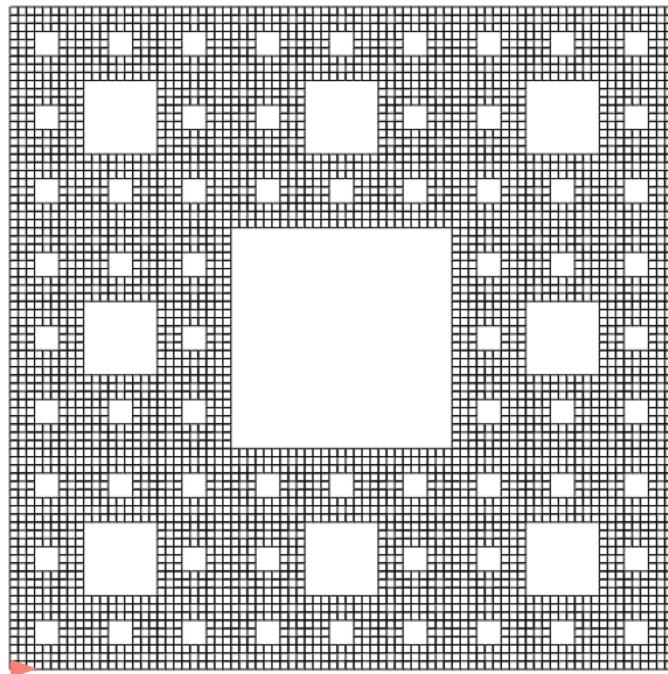


(copo-nieve 3 200)

## Ejercicio 6

Define la función (`alfombra-sierpinski tam`) que construya la Alfombra de Sierpinski (una variante del Triágulo de Sierpinski que hemos visto en teoría) de lado `tam` píxeles utilizando gráficos de tortuga.

Por ejemplo, la llamada a (`alfombra-sierpinski 500`) debe dibujar la siguiente figura:



## SOLUCION PRACTICA 5

```
#lang racket
(require rackunit)
(require graphics/turtles)

;;;;;;
; Ejercicio 1 ;
;;;;;;

;;;;;;
; a) ;
;;;;;;

(define (concat-iter lista-cadenas res)
  (if (null? lista-cadenas)
      res
      (concat-iter (cdr lista-cadenas) (string-append res (car lista-cadenas)))))

(check-equal? (concat-iter '("hola" "y" "adiós") "") "holayadiós")

(define (concat lista-cadenas)
  (concat-iter lista-cadenas ""))
  
(check-equal? (concat '("hola" "y" "adiós")) "holayadiós")
```

```

;;;;;
; b) ;
;;;;;

(define (min-max-pareja dato pareja)
  (cons (min dato (car pareja))
        (max dato (cdr pareja)))))

(check-equal? (min-max-pareja 3 '(10 . 20)) '(3 . 20))
(check-equal? (min-max-pareja 30 '(10 . 20)) '(10 . 30))
(check-equal? (min-max-pareja 15 '(10 . 20)) '(10 . 20))

(define (min-max-iter lista res)
  (if (null? lista)
      res
      (min-max-iter (cdr lista)
                    (min-max-pareja (car lista) res)))))

(define (min-max lista)
  (min-max-iter (cdr lista) (cons (car lista) (car lista)))))

(check-equal? (min-max '(2 5 9 12 5 0 4)) '(0 . 12))

;;;;;;
; Ejercicio 2 ;
;;;;;;

;;;;;
; a) ;
;;;;;

(define (expande-pareja-iter pareja res)
  (if (= 0 (cdr pareja))
      res
      (expande-pareja-iter (cons (car pareja)
                                 (- (cdr pareja) 1))
                           (cons (car pareja) res)))))

(define (expande-pareja pareja)
  (expande-pareja-iter pareja '()))

(check-equal? (expande-pareja (cons 'a 4)) '(a a a a))

(define (expande-parejas-iter lista-parejas res)
  (if (null? lista-parejas)
      res
      (expande-parejas-iter (cdr lista-parejas)
                            (append res (expande-pareja (car lista-parejas))))))

(define (expande-parejas . lista-parejas)
  (expande-parejas-iter lista-parejas '()))

```

```

(check-equal? (expande-parejas '(#t . 3) '("LPP" . 2) '(b . 4)) '(#t #t #t "LPP"
"LPP" b b b b))

;;;;;;
; b) ;
;;;;;;

(define (rotar n lista)
  (if (= n 0)
      lista
      (rotar (- n 1) (append (cdr lista) (list (car lista))))))

(check-equal? (rotar 4 '(a b c d e f g)) '(e f g a b c d))

;;;;;;;;
; Ejercicio 3 ;
;;;;;;;;
;;;;;;
; a) ;
;;;;;;

(define (mi-foldl funcion resultado lista)
  (if (null? lista)
      resultado
      (mi-foldl funcion (funcion (car lista) resultado) (cdr lista)))))

(check-equal? (mi-foldl string-append "****" '("hola" "que" "tal"))
"talquehola****")
(check-equal? (mi-foldl cons '() '(1 2 3 4)) '(4 3 2 1))

;;;;;;
; b) ;
;;;;;;

(define (prefijo-lista? lista1 lista2)
  (or (null? lista1)
      (and (equal? (car lista1) (car lista2))
           (prefijo-lista? (cdr lista1) (cdr lista2)))))

(check-true (prefijo-lista? '(a b c) '(a b c d e)))
(check-false (prefijo-lista? '(b c) '(a b c d e)))

;;;;;;;;
; Ejercicio 4 ;
;;;;;;;;
;;;;;;
; diccionario ;
;;;;;;;;
(define (crea-diccionario) ;
```

```

(mcons '*diccionario* '()))
; ;
;

(define (busca key dic)
  (cond
    ((null? dic) #f)
    ((equal? key (mcar (mcar dic)))
     (mcar dic))
    (else (busca key (mcdr dic)))))

(define (get key dic)
  (define record (busca key (mcdr dic)))
  (if (not record)
      #f
      (mcdr record)))

(define (put key value dic)
  (define record (busca key (mcdr dic)))
  (if (not record)
      (set-mcdr! dic
                  (mcons (mcons key value)
                         (mcdr dic)))
      (set-mcdr! record value))
  value))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (pascal-memo fil col dic)
  (cond ((= col 0) 1)
        ((= col fil) 1)
        ((not (equal? (get (cons fil col) dic) #f)) (get (cons fil col) dic))
        (else (put (cons fil col) (+ (pascal-memo (- fil 1) (- col 1) dic)
                                       (pascal-memo (- fil 1) col dic))))))

(define diccionario (crea-diccionario))
(check-equal? (pascal-memo 8 4 diccionario) 70)
(check-equal? (pascal-memo 40 20 diccionario) 137846528820)

;;;;;;;;;;;;;;;;;
; Ejercicio 5 ;
;;;;;;;;;;;;;;;;;

;;;;;;
; a) ;
;;;;;;

(define (koch nivel trazo)
  (if (= 0 nivel)
      (draw trazo)
      (begin
        (koch (- nivel 1) (/ trazo 3))
        (turn 60)
        (koch (- nivel 1) (/ trazo 3))
        (turn -120)
        (koch (- nivel 1) (/ trazo 3)))

```

```
(turn 60)
(koch (- nivel 1) (/ trazo 3)))))

(define (koch-ventana nivel trazo)
  (turtles #t)
  (clear)
  (turn 180)
  (move (/ trazo 2))
  (turn 180)
  (koch nivel trazo))

(koch-ventana 1 600)
(koch-ventana 2 600)
(koch-ventana 3 600)

;;;;;;
; b) ;
;;;;;;

(define (copo-nieve nivel trazo)
  (begin
    (koch nivel trazo)
    (turn -120)
    (koch nivel trazo)
    (turn -120)
    (koch nivel trazo)))

(define (copo-nieve-ventana nivel trazo)
  (turtles #t)
  (clear)
  (turn 120)
  (move (/ trazo 3))
  (turn 60)
  (move (/ trazo 3))
  (turn 180)
  (copo-nieve nivel trazo))

(copo-nieve-ventana 1 200)
(copo-nieve-ventana 2 200)
(copo-nieve-ventana 3 200)

;;;;;;;;
; Ejercicio 6 ;
;;;;;;;;
;;;

(define (dibuja-cuadrado w)
  (begin
    (draw w)
    (turn 90)
    (draw w)
    (turn 90)
    (draw w)
    (turn 90))
```

```

(draw w)
(turn 90))

(define (alfombra-sierpinski w)
  (if (> w 20)
      (begin
        (alfombra-sierpinski (/ w 3))
        (move (/ w 3))
        (alfombra-sierpinski (/ w 3))
        (move (/ w 3))
        (alfombra-sierpinski (/ w 3))
        (turn 90)(move (/ w 3))(turn -90)
        (alfombra-sierpinski (/ w 3))
        (turn 90)(move (/ w 3))(turn -90)
        (alfombra-sierpinski (/ w 3))
        (turn 180)(move(/ w 3))(turn -180)
        (alfombra-sierpinski (/ w 3))
        (turn 180)(move(/ w 3))(turn -180)
        (alfombra-sierpinski (/ w 3))
        (turn -90) (move (/ w 3)) (turn 90)
        (alfombra-sierpinski (/ w 3))
        (turn -90) (move (/ w 3)) (turn 90)) ;; volvemos a la posición original
      (dibuja-cuadrado w)))
    )

(define (alfombra-sierpinski-ventana w)
  (turtles #t)
  (clear)
  (turn -90)
  (move (/ w 2))
  (turn -90)
  (move (/ w 2))
  (turn 180)
  (alfombra-sierpinski w))

(alfombra-sierpinski-ventana 600)

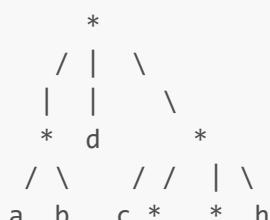
```

## PRACTICA 6

---

### Ejercicio 1

- a) Escribe la lista estructurada correspondiente a la siguiente representación gráfica por niveles. Para comprobar si la has definido correctamente puedes intentar obtener algunos de los elementos de la lista, como mostramos en el [check-equal?](#) que hay a continuación.





```
(define lista-a '(_))
(check-equal? (caddr (caddr lista-a)) 'h)
```

b) Dibuja la representación en niveles de las siguientes listas estructuradas

```
(define lista-b1 '((2 (3)) (4 2) ((2) 3)))
(define lista-b2 '((b) (c (a)) d (a)))
```

c) Dada la definición de **cuadrado-estruct** vista en teoría:

```
(define (cuadrado-estruct elem)
  (cond ((null? elem) '())
        ((hoja? elem) (* elem elem))
        (else (cons (cuadrado-estruct (car elem))
                     (cuadrado-estruct (cdr elem))))))
```



1. Indica qué devuelve la expresión **(cuadrado-estruct lista-b1)**. La lista **lista-b1** es la definida en el apartado anterior.
2. En la evaluación de la expresión anterior, indica cuáles son los argumentos que se pasan por parámetro en las llamadas recursivas a **cuadrado-estruct** marcadas con **1** y **2**.
3. En la evaluación de la expresión anterior, indican qué devuelven las llamadas recursivas marcadas con **1** y **2**.

d) Para entender el funcionamiento de las funciones de orden superior que trabajan sobre listas estructuradas es muy importante entender qué devuelve la expresión **map** que se aplica a la lista.

Dada la definición de **(nivel-hoja-fos dato lista)** vista en teoría, indica qué devuelve la siguiente expresión (es la expresión **map** que hay en su cuerpo). La lista **lista-b2** es la definida en el apartado anterior. Utiliza el dibujo que has hecho en el ejercicio anterior para entender el funcionamiento de la expresión.

```
(map (lambda (elem)
           (if (hoja? elem)
               (if (equal? elem 'a) 0 -1)
               (nivel-hoja-fos 'a elem)))
      lista-b2)
```

## Ejercicio 2

- a) Implementa la función recursiva **(cuenta-pares lista)** que recibe una lista estructurada y cuenta la cantidad de números pares que contiene. Implementa dos versiones de la función, una con **recursión pura** y

otra con **funciones de orden superior**.

Ejemplos:

```
(cuenta-pares '(1 (2 3) 4 (5 6))) ; => 3
(cuenta-pares '(((1 2) 3 (4) 5) (((6))))) ; => 3
```

b) Implementa la función recursiva (**todos-positivos lista**) que recibe una lista estructurada con números y comprueba si todos sus elementos son positivos. Implementa dos versiones de la función, una con **recursión pura** y otra con **funciones de orden superior**.

Ejemplos:

```
(todos-positivos-fos '(1 (2 (3 (-3)))) 4)) ; => #f
(todos-positivos-fos '(1 (2 (3 (3)))) 4)) ; => #t
```

### Ejercicio 3

Implementa la función (**cumplen-predicado pred lista**) que devuelva una lista con todos los elementos de lista estructurada que cumplen un predicado. Implementa dos versiones, una **recursiva pura** y otra usando **funciones de orden superior**.

Ejemplo:

```
(cumplen-predicado even? '(1 (2 (3 (4))) (5 6))) ; => (2 4 6)
(cumplen-predicado pair? '(((1 . 2) 3 (4 . 3) 5) 6)) ; => ((1 . 2) (4 . 3))
```

Utilizando la función anterior implementa las siguientes funciones:

- Función (**busca-mayores n lista-num**) que recibe una lista estructurada con números y un número **n** y devuelve una lista plana con los números de la lista original mayores que **n**.

```
(busca-mayores 10 '(-1 (20 (10 12) (30 (25 (15))))) ; => (20 12 30 25 15)
```

- Función (**empieza-por char lista-pal**) que recibe una lista estructurada con símbolos y un carácter **char** y devuelve una lista plana con los símbolos de la lista original que comienzan por el carácter **char**.

```
(empieza-por #\m '((hace (mucho tiempo)) (en) (una galaxia ((muy muy)
lejana))))
; => (mucho muy muy)
```

## Ejercicio 4

a) Implementa la función recursiva (`sustituye-elem elem-old elem-new lista`) que recibe como argumentos una lista estructurada y dos elementos, y devuelve otra lista con la misma estructura, pero en la que se ha sustituido las ocurrencias de `elem-old` por `elem-new`.

Ejemplo:

```
(sustituye-elem 'c 'h '(a b (c d (e c)) c (f (c) g)))
; => (a b (h d (e h)) h (f (h) g))
```

b) Implementa la función recursiva (`diff-listas l1 l2`) que tome como argumentos dos listas estructuradas con la misma estructura, pero con diferentes elementos, y devuelva una lista de parejas que contenga los elementos que son diferentes.

Ejemplos:

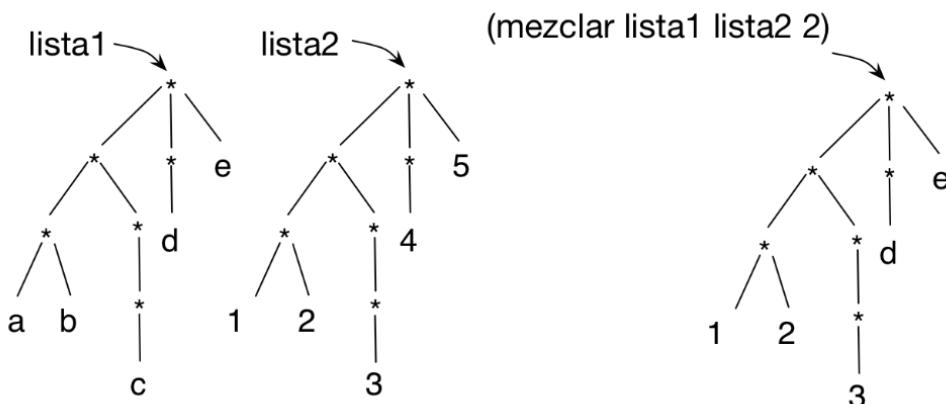
```
(diff-listas '(a (b ((c)) d e) f) '(1 (b ((2)) 3 4) f))
; => ((a . 1) (c . 2) (d . 3) (e . 4))

(diff-listas '((a b) c) '((a b) c))
; => ()
```

## Ejercicio 5

Dos funciones sobre niveles:

a) Define la función recursiva (`mezclar lista1 lista2 n`) que reciba dos listas estructuradas con la misma estructura y un número que indica un nivel. Devuelve una nueva lista estructurada con la misma estructura que las listas originales, con los elementos de `lista1` que tienen un nivel menor o igual que `n` y los elementos de `lista2` que tienen un nivel mayor que `n`.



```
(define lista1 '(((a b) ((c))) (d) e))
(define lista2 '(((1 2) ((3))) (4) 5))
```

```
(mezclar lista1 lista2 2) ; ⇒ (((1 2) ((3))) (d) e)
```

b) Implementa la función (**nivel-elemento lista**) que reciba una lista estructurada y devuelva una pareja (**elem . nivel**), donde la parte izquierda es el elemento que se encuentra a mayor nivel y la parte derecha el nivel en el que se encuentra. Puedes definir alguna función auxiliar si lo necesitas. Puedes hacerlo con recursión o con funciones de orden superior.

```
(nivel-elemento '(2 (3))) ; ⇒ (3 . 2)
(nivel-elemento '((2) (3 (4)((((5)) 6)) 7)) 8)) ; ⇒ (5 . 8)
```

## SOLUCION PRACTICA 6

```
#lang racket
(require rackunit)

(define (for-all? predicado lista)
  (or (null? lista)
      (and (predicado (car lista))
           (for-all? predicado (cdr lista)))))

(define (hoja? elem)
  (not (list? elem)))

;-----;
; Ejercicio 1 ;
;-----;

;----;
; a) ;
;----;

(define lista-a '((a b) d (c (e) (f g) h)))
;En estos ejemplos se usa una llamada a función por cada nivel (por cada lista):
(check-equal? (car (car lista-a)) 'a)
(check-equal? (cadr (car lista-a)) 'b)
(check-equal? (car (caddr lista-a)) 'c)
(check-equal? (cadr lista-a) 'd)
(check-equal? (car (cadr (caddr lista-a))) 'e)
(check-equal? (car (caddr (caddr lista-a))) 'f)
(check-equal? (cadr (caddr (caddr lista-a))) 'g)
(check-equal? (cadddr (caddr lista-a)) 'h)

;----;
; b) ;
;----;

(define lista-b1 '((2 (3)) (4 2) ((2) 3)))
;          *
```

```

;      /   |   \
;      /   |       \
;      *   *       *
;  / \   / \   / \
; 2   * 4   2   *   3
;      |           |
;      3           2

(define lista-b2 '((b) (c (a)) d (a)))
;      *
;      /|  \\
;      * * d *
;  / /|   |
; b c *   a
;      |
;      a

;----;
; c) ;
;----;

; 1) ((4 (9)) (16 4) ((4) 9))
; 2) 1 -> (cuadrado-estruct '(2 (3)))
;      2 -> (cuadrado-estruct '((4 2) ((2) 3)))
; 3) 1 -> (4 (9))
;      2 -> ((16 4) ((4) 9))

;----;
; d) ;
;----;

; (-1 2 -1 1)

;-----;
; Ejercicio 2 ;
;-----;

;----;
; a) ;
;----;

(define (cuenta-pares elem)
  (cond ((null? elem) 0)
        ((hoja? elem) (if (even? elem) 1 0))
        (else (+ (cuenta-pares (car elem))
                  (cuenta-pares (cdr elem))))))

(check-equal? (cuenta-pares '(1 (2 3) 4 (5 6))) 3)
(check-equal? (cuenta-pares '(((1 2) 3 (4) 5) (((6)))))) 3

(define (cuenta-pares-fos lista)
  (foldr +
         0

```

```

        (map (lambda (elem)
            (if (hoja? elem)
                (if (even? elem) 1 0)
                (cuenta-pares-fos elem))) lista)))

(check-equal? (cuenta-pares-fos '(1 (2 3) 4 (5 6))) 3)
(check-equal? (cuenta-pares-fos '(((1 2) 3 (4) 5) (((6))))) 3)

;----;
; b) ;
;----;

(define (todos-positivos elem)
  (cond
    ((null? elem) #t)
    ((hoja? elem) (positive? elem))
    (else (and (todos-positivos (car elem))
                (todos-positivos (cdr elem))))))

(check-false (todos-positivos '(1 (2 (3 (-3))) 4)))
(check-true (todos-positivos '(1 (2 (3 (3))) 4)))

(define (todos-positivos-fos lista)
  (for-all? (lambda (elem)
              (if (hoja? elem)
                  (positive? elem)
                  (todos-positivos-fos elem))) lista))

(check-false (todos-positivos-fos '(1 (2 (3 (-3))) 4)))
(check-true (todos-positivos-fos '(1 (2 (3 (3))) 4)))

;-----;
; Ejercicio 3 ;
;-----;

(define (cumplen-predicado pred elem)
  (cond ((null? elem) '())
        ((hoja? elem)
         (if (pred elem)
             (list elem)
             '()))
        (else (append (cumplen-predicado pred (car elem))
                      (cumplen-predicado pred (cdr elem))))))

(check-equal? (cumplen-predicado even? '(1 (2 (3 (4))) (5 6))) '(2 4 6))
(check-equal? (cumplen-predicado pair? '(((1 . 2) 3 (4 . 3) 5) 6)) '((1 . 2) (4 . 3)))

(define (cumplen-predicado-fos pred lista)
  (foldr append
        '()
        (map (lambda (elem)

```

```

(if (hoja? elem)
    (if (pred elem) (list elem) '())
    (cumplen-predicado-fos pred elem)))
lista)))

(check-equal? (cumplen-predicado-fos even? '(1 (2 (3 (4))) (5 6))) '(2 4 6))
(check-equal? (cumplen-predicado-fos pair? '(((1 . 2) 3 (4 . 3) 5) 6)) '((1 . 2)
(4 . 3)))

(define (busca-mayores n lista-num)
  (cumplen-predicado (lambda (elem) (> elem n)) lista-num))

(check-equal? (busca-mayores 10 '(-1 (20 (10 12) (30 (25 (15)))))) '(20 12 30 25
15))

(define (empieza-por char lista-pal)
  (cumplen-predicado (lambda (elem)
    (equal? char (string-ref (symbol->string elem) 0)))
  lista-pal))

(check-equal? (empieza-por #\m
  '((hace (mucho tiempo)) (en) (una galaxia ((muy muy
lejana))))
  '(mucho muy muy)))

;-----;
; Ejercicio 4 ;
;-----;

;----;
; a) ;
;----;

(define (sustituye-elem elem-old elem-new elem)
  (cond ((null? elem) '())
        ((hoja? elem)
         (if (equal? elem-old elem)
             elem-new
             elem)))
        (else (cons (sustituye-elem elem-old elem-new (car elem))
                    (sustituye-elem elem-old elem-new (cdr elem))))))

(check-equal? (sustituye-elem 'c 'h '(a b (c d (e c)) c (f (c) g))) '(a b (h d (e
h)) h (f (h) g)))

;----;
; b) ;
;----;

(define (diff-listas l1 l2)
  (cond ((null? l1) '())
        ((hoja? l1)
         (if (equal? l1 l2)

```

```

'()
(list (cons 11 12)))
(else (append (diff-listas (car 11) (car 12))
              (diff-listas (cdr 11) (cdr 12)))))

(check-equal? (diff-listas '(a (b ((c)) d e) f) '(1 (b ((2)) 3 4) f)) '((a . 1) (c
. 2) (d . 3) (e . 4)))
(check-equal? (diff-listas '((a b) c) '((a b) c)) '())

;-----;
; Ejercicio 5 ;
;-----;

;----;
; a) ;
;----;

(define (mezclar lista1 lista2 n)
  (cond ((null? lista1) '())
        ((hoja? lista1) (if (< n 0) lista2 lista1))
        (else (cons (mezclar (car lista1) (car lista2) (- n 1))
                    (mezclar (cdr lista1) (cdr lista2) n)))))

(define lista1 '(((a b) ((c))) (d) e))
(define lista2 '(((1 2) ((3))) (4) 5))
(check-equal? (mezclar lista1 lista2 2) '(((1 2) ((3))) (d) e))

;----;
; b) ;
;----;

(define (incrementa-nivel pareja)
  (cons (car pareja) (+ (cdr pareja) 1)))

(define (mas-profundo pareja1 pareja2)
  (if (or (null? pareja2)
           (> (cdr pareja1) (cdr pareja2)))
      pareja1
      pareja2))

(define (nivel-elemento elem)
  (cond
    ((null? elem) '())
    ((hoja? elem) (cons elem 0))
    (else (mas-profundo
            (incrementa-nivel (nivel-elemento (car elem)))
            (nivel-elemento (cdr elem))))))

;Versión con FOS
;(define (nivel-elemento lista)
;  (incrementa-nivel (foldr mas-profundo
;                           '()
;                           (map (lambda (elem)
;
```

```

;
;           (if (hoja? elem)
;                   (cons elem 0)
;                   (nivel-elemento elem)))
;           lista)))))

(check-equal? (nivel-elemento '(2 (3))) '(3 . 2))
(check-equal? (nivel-elemento '((2) (3 (4)((((5)) 6)) 7)) 8)) '(5 . 8))

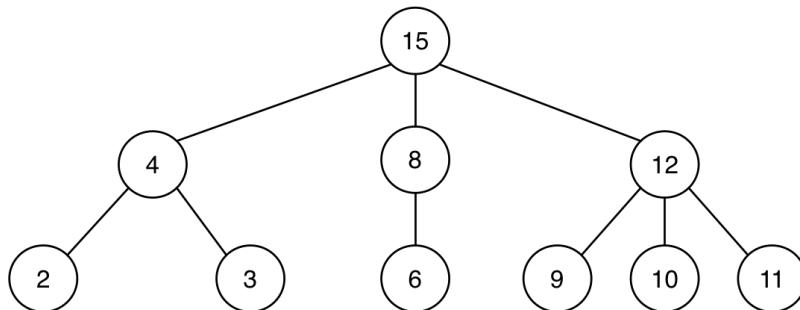
```

## PRACTICA 7

---

### Ejercicio 1

a.1) Escribe la sentencia en Scheme que define el siguiente árbol genérico y escribe **utilizando las funciones de la barrera de abstracción de árboles** una expresión que devuelva el número 10.



```

(define arbol '(- - - - -))
(check-equal? ----- 10)

```

a.2) Las funciones que suman los datos de un árbol utilizando recursión mutua y que hemos visto en teoría son las siguientes:

```

(define (suma-datos-arbol arbol)
  (+ (dato-arbol arbol)
     (suma-datos-bosque (hijos-arbol arbol)))

(define (suma-datos-bosque bosque)
  (if (null? bosque)
      0
      (+ (suma-datos-arbol (car bosque))
         (suma-datos-bosque (cdr bosque)))))

```

Si realizamos la siguiente llamada a la función **suma-datos-bosque**, siendo **arbol** el definido en el apartado anterior:

```
(suma-datos-bosque (hijos-arbol arbol))
```

1. ¿Qué devuelve la invocación a `(suma-datos-arbol (car bosque))` que se realiza dentro de la función?
2. ¿Qué devuelve la primera llamada recursiva a `suma-datos-bosque`?

Escribe la contestación a estas preguntas como comentarios en el fichero de la práctica.

a.3) La función de orden superior que hemos visto en teoría y que realiza también la suma de los datos de un árbol es:

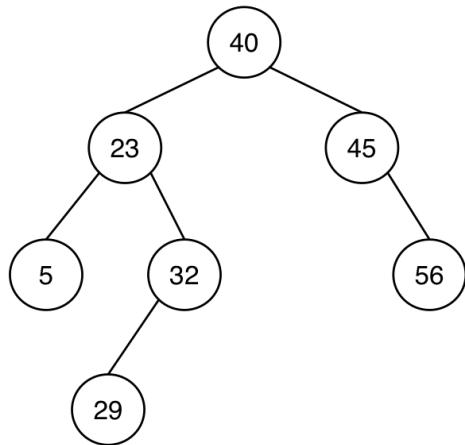
```
(define (suma-datos-arbol-fos arbol)
  (foldr + (dato-arbol arbol)
    (map suma-datos-arbol-fos (hijos-arbol arbol))))
```

Si realizamos la siguiente llamada a la función, siendo `arbol` el definido en el apartado anterior:

```
(suma-datos-arbol-fos arbol)
```

1. ¿Qué devuelve la invocación a `map` dentro de la función?
2. ¿Qué invocaciones se realizan a la función `+` durante la ejecución de `foldr` sobre la lista devuelta por la invocación a `map`? Enuméralas en orden, indicando sus parámetros y el valor devuelto en cada una de ellas.

b.1) Escribe la sentencia en Scheme que define el siguiente árbol binario y escribe **utilizando las funciones de la barrera de abstracción de árboles binarios** una expresión que devuelva el número 29.



```
(define arbolb '(- - - - -))
(check-equal? - - - - - 29)
```

## Ejercicio 2

a) Implementa dos versiones de la función `(to-string-arbol arbol)` que recibe un árbol de símbolos y devuelve la cadena resultante de concatenar todos los símbolos en recorrido preorder. Debes implementar

una versión con recursión mutua y otra (llamada `to-string-arbol-fos`) con una única función en la que se use funciones de orden superior.

Ejemplo:

```
(define arbol2 '(a (b (c (d)) (e)) (f)))
(to-string-arbol arbol2) ; => "abcdef"
```

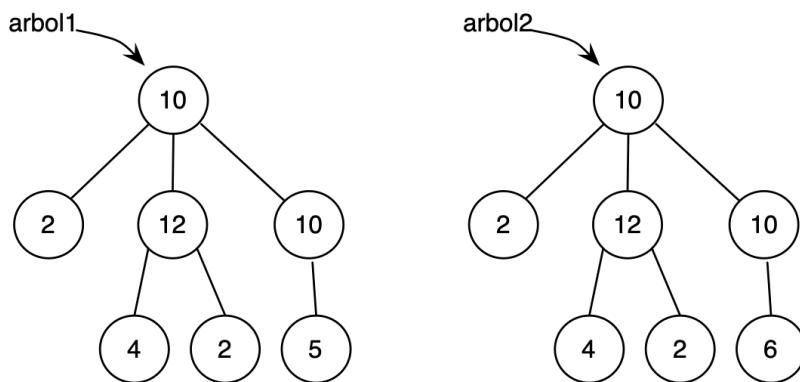
b) Implementa dos versiones de la función (`veces-arbol dato arbol`) que recibe un árbol y un dato y comprueba el número de veces que aparece el dato en el árbol. Debes implementar una función con recursión mutua y otra con funciones de orden superior.

```
(veces-arbol 'b '(a (b (c) (d)) (b (b) (f)))) ; => 3
(veces-arbol 'g '(a (b (c) (d)) (b (b) (f)))) ; => 0
```

### Ejercicio 3

a) Implementa dos versiones de la función (`hojas-cumplen pred arbol`) que recibe un predicado y un árbol y devuelve una lista con todas aquellas hojas del árbol que cumplen el predicado. Una función con recursión mutua y otra con funciones de orden superior.

Para evitar complicar la función de orden superior, suponemos que el árbol inicial que pasamos como parámetro no es un árbol hoja.



```
(define arbol1 '(10 (2) (12 (4) (2)) (10 (5))))
(define arbol2 '(10 (2) (12 (4) (2)) (10 (6))))
(hojas-cumplen even? arbol1) ; => '(2 4 2)
(hojas-cumplen even? arbol2) ; => '(2 4 2 6)
```

b) Implementa dos versiones del predicado (`todas-hojas-cumplen? pred arbol`) que comprueba si todas las hojas de un árbol cumplen un determinado predicado. Una función con recursión mutua y otra con funciones de orden superior.

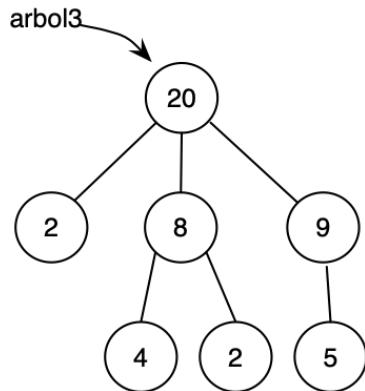
No debes usar la función anterior, tienes que hacer un recorrido por todo el árbol. Para la función de orden superior puedes usar la función `for-all?` implementada en el tema 2.

```
(todas-hojas-cumplen? even? arbol1) ; => #f
(todas-hojas-cumplen? even? arbol2) ; => #t
```

## Ejercicio 4

- a) Implementa, utilizando funciones de orden superior, la función (**suma-raices-hijos arbol**) que devuelva la suma de las raíces de los hijos de un árbol genérico.

Ejemplo:



```
(define arbol3 '(20 (2) (8 (4) (2)) (9 (5))))
(suma-raices-hijos arbol3) ; => 19
(suma-raices-hijos (cadr (hijos-arbol arbol3))) ; => 6
```

- b) Implementa dos versiones, una con recursión mutua y otra con funciones de orden superior, de la función (**raices-mayores-arbol? arbol**) que recibe un árbol y comprueba que su raíz sea mayor que la suma de las raíces de los hijos y que todos los hijos (nos referimos a todos los descendientes) cumplen también esta propiedad.

Ejemplos:

```
(raices-mayores-arbol? arbol3) ; => #t
(raices-mayores-arbol? '(20 (2) (8 (4) (5)) (9 (5)))) ; => #f
```

- c) Define la función (**comprueba-raices-arbol arbol**) que recibe un arbol y que devuelve otro arbol en el que los nodos se han sustituido por 1 o 0 según si son mayores que la suma de las raíces de sus hijos o no.

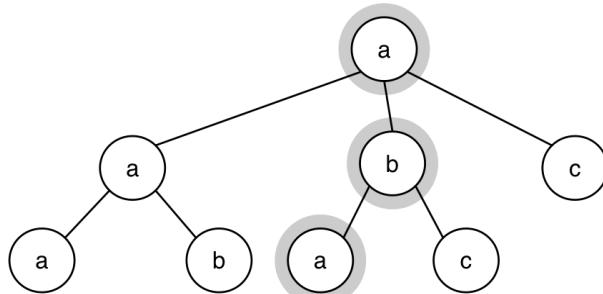
Ejemplos:

```
(comprueba-raices-arbol arbol3) ; => (1 (1) (1 (1) (1)) (1 (1)))
(comprueba-raices-arbol '(20 (2) (8 (4) (5)) (9 (5))))
; => (1 (1) (0 (1) (1)) (1 (1)))
```

## Ejercicio 5

a) Define la función (`es-camino? lista arbol`) que debe comprobar si la secuencia de elementos de la lista se corresponde con un camino del árbol que empieza en la raíz y que termina exactamente en una hoja. Suponemos que `lista` contiene al menos un elemento

Por ejemplo, la lista `(a b a)` sí que es camino en el siguiente árbol, pero la lista `(a b)` no.

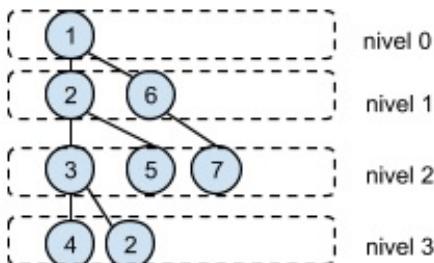


Ejemplos: suponiendo que `arbol` es el árbol definido por la figura anterior:

```

(es-camino? '(a b a) arbol) => #t
(es-camino? '(a b) arbol) => #f
(es-camino? '(a b a b) arbol) => #f
  
```

b) Escribe la función (`nodos-nivel nivel arbol`) que reciba un nivel y un árbol genérico y devuelva una lista con todos los nodos que se encuentran en ese nivel.



Ejemplos, suponiendo que `arbol` es el árbol definido por la figura anterior:

```

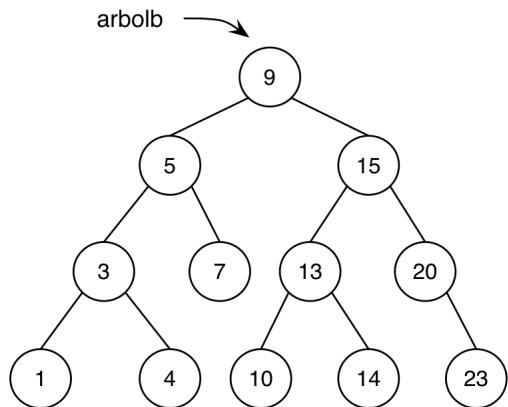
(nodos-nivel 0 arbol) => '(1)
(nodos-nivel 1 arbol) => '(2 6)
(nodos-nivel 2 arbol) => '(3 5 7)
(nodos-nivel 3 arbol) => '(4 2)
  
```

## Ejercicio 6

Dado un árbol binario y un camino definido como una lista de símbolos: '`< > =`' en el que:

- `<`: indica que nos vamos por la rama izquierda
- `>`: indica que nos vamos por la rama derecha
- `=`: indica que nos quedamos con el dato de ese nodo.

Implementa la función (`camino-b-tree b-tree camino`) que devuelva una lista con los datos recogidos por el camino.



```
(camino-b-tree b-tree '(= < < = > =)) => '(9 3 4)
(camino-b-tree b-tree '(> = < < =)) => '(15 10)
```

## SOLUCION PRACTICA 7

```
#lang racket
(require rackunit)

;-----
; Implementación FOS exist? y for-all?
;-----

(define (exists? predicado lista)
  (and (not (null? lista))
       (or (predicado (car lista))
           (exists? predicado (cdr lista)))))

(define (for-all? predicado lista)
  (or (null? lista)
      (and (predicado (car lista))
           (for-all? predicado (cdr lista)))))

;-----
; Barrera de abstracción arbol genérico
;-----

(define (dato-arbol arbol)
  (car arbol))

(define (hijos-arbol arbol)
  (cdr arbol))

(define (hoja-arbol? arbol)
  (null? (hijos-arbol arbol)))
```

```

(define (nuevo-arbol dato lista-arboles)
  (cons dato lista-arboles))

;-----;
; Barrera de abstracción arbol binario ;
;-----;

(define (dato-arbolb arbol)
  (car arbol))

(define (hijo-izq-arbolb arbol)
  (cadr arbol))

(define (hijo-der-arbolb arbol)
  (caddr arbol))

(define arbolb-vacio '())

(define (vacio-arbolb? arbol)
  (equal? arbol arbolb-vacio))

(define (hoja-arbolb? arbol)
  (and (vacio-arbolb? (hijo-izq-arbolb arbol))
       (vacio-arbolb? (hijo-der-arbolb arbol)))))

(define (nuevo-arbolb dato hijo-izq hijo-der)
  (list dato hijo-izq hijo-der))

;-----;
; Ejercicio 1 ;
;-----;

;-----;
; a.1) ;
;-----;

(define arbol '(15 (4 (2
                        (3))
                     (8 (6))
                     (12 (9)
                          (10)
                          (11)))))

(check-equal? (dato-arbol (cadr (hijos-arbol (caddr (hijos-arbol arbol)))))) 10)

;-----;
; a.2) ;
;-----;

; 1. (suma-datos-arbol (car bosque)) -> 9
; 2. (suma-datos-bosque (cdr bosque)) -> 56 (14+42)
;                               + ---
; (suma-datos-bosque (hijos-arbol arbol)) 65

```

;Comprobación:

```
(define (suma-datos-arbol arbol)
  (+ (dato-arbol arbol)
      (suma-datos-bosque (hijos-arbol arbol)))))

(define (suma-datos-bosque bosque)
  (if (null? bosque)
      0
      (+ (suma-datos-arbol (car bosque))
          (suma-datos-bosque (cdr bosque)))))

(check-equal? (suma-datos-bosque (hijos-arbol arbol)) 65)
(check-equal? (suma-datos-arbol (car (hijos-arbol arbol))) 9)
(check-equal? (suma-datos-bosque (cdr (hijos-arbol arbol))) 56)
(check-equal? (suma-datos-arbol (cadr (hijos-arbol arbol))) 14)
(check-equal? (suma-datos-arbol (caddr (hijos-arbol arbol))) 42)
```

```
;-----;
; a.3) ;
;-----;
```

```
; 1. (map suma-datos-arbol-fos (hijos-arbol arbol)) -> (9 14 42)
; 2. (+ 42 15) -> 57
;     (+ 14 57) -> 71
;     (+ 9 71) -> 80
;
```

;Comprobación:

```
(define (suma-datos-arbol-fos arbol)
  (foldr + (dato-arbol arbol)
           (map suma-datos-arbol-fos (hijos-arbol arbol)))))

(check-equal? (map suma-datos-arbol-fos (hijos-arbol arbol)) '(9 14 42))

(check-equal? (foldr + (dato-arbol arbol) '(42)) (+ 42 15))
(check-equal? (foldr + (dato-arbol arbol) '(14 42)) (+ 14 (+ 42 15)))
(check-equal? (foldr + (dato-arbol arbol) '(9 14 42)) (+ 9 (+ 14 (+ 42 15))))
```

```
;-----;
; b.1) ;
;-----;
```

```
(define arbolb '(40 (23 (5 () ())
                           (32 (29 () ())
                               ())))
                  (45 ())
                  (56 () ())))
(check-equal? (dato-arbol (hijo-izq-arbolb (hijo-der-arbolb (hijo-izq-arbolb
arbolb)))) 29)
```

```
;-----;
; Ejercicio 2 ;
```

```

;-----;

;----;
; a) ;
;----;

(define (to-string-arbol arbol)
  (string-append (symbol->string (dato-arbol arbol))
    (to-string-bosque (hijos-arbol arbol)))))

(define (to-string-bosque bosque)
  (if (null? bosque)
    ""
    (string-append (to-string-arbol (car bosque))
      (to-string-bosque (cdr bosque))))))

(define arbol2.1 '(a (b (c (d)
                           (e))
                           (f))))
  (check-equal? (to-string-arbol arbol2.1) "abcdef"))

(define (to-string-arbol-fos arbol)
  (foldr string-append ""
    (cons (symbol->string (dato-arbol arbol))
      (map to-string-arbol-fos (hijos-arbol arbol)))))

  (check-equal? (to-string-arbol-fos arbol2.1) "abcdef"))

;----;
; b) ;
;----;

(define (veces-arbol dato arbol)
  (+ (if (equal? dato (dato-arbol arbol)) 1 0)
    (veces-bosque dato (hijos-arbol arbol)))))

(define (veces-bosque dato bosque)
  (if (null? bosque)
    0
    (+ (veces-arbol dato (car bosque))
      (veces-bosque dato (cdr bosque))))))

(define arbol2.2 '(a (b (c)
                           (d))
                           (b (b))
                           (f))))
  (check-equal? (veces-arbol 'b arbol2.2) 3)
  (check-equal? (veces-arbol 'g arbol2.2) 0)

(define (veces-arbol-fos dato arbol)
  (foldr + (if (equal? dato (dato-arbol arbol)) 1 0)
    (map (lambda (subarbol)
      (veces-arbol-fos dato subarbol))
        (hijos-arbol arbol)))))


```

```

(check-equal? (veces-arbol-fos 'b arbol2.2) 3)
(check-equal? (veces-arbol-fos 'g arbol2.2) 0)

;-----
; Ejercicio 3 ;
;-----

;---
; a) ;
;---;

(define (hojas-cumplen pred arbol)
  (if (and (hoja-arbol? arbol)
            (pred (dato-arbol arbol)))
      (list (dato-arbol arbol))
      (hojas-cumplen-bosque pred (hijos-arbol arbol)))))

(define (hojas-cumplen-bosque pred bosque)
  (if (null? bosque)
      '()
      (append (hojas-cumplen pred (car bosque))
              (hojas-cumplen-bosque pred (cdr bosque))))))

(define arbol3.1 '(10 (2
                        (12 (4
                              (2))
                              (10 (5)))))
(define arbol3.2 '(10 (2
                        (12 (4
                              (2))
                              (10 (6)))))
(check-equal? (hojas-cumplen even? arbol3.1) '(2 4 2))
(check-equal? (hojas-cumplen even? arbol3.2) '(2 4 2 6))

; Suponemos que el árbol que se pasa como parámetro
; no es un árbol hoja
(define (hojas-cumplen-fos pred arbol)
  (foldr append
         '()
         (map (lambda (subarbol)
                (if (hoja-arbol? subarbol)
                    (if (pred (dato-arbol subarbol))
                        (list (dato-arbol subarbol))
                        '())
                        (hojas-cumplen-fos pred subarbol)))
                (hijos-arbol arbol)))))

; Alternativa (aplicable a un árbol hoja) en la que se saca la comparación de
; si es hoja fuera de la expresión lambda
(define (hojas-cumplen-fos2 pred arbol)
  (if (and (hoja-arbol? arbol)
            (pred (dato-arbol arbol))))

```

```

(list (dato-arbol arbol))
(foldr append
      '()
      (map (lambda (subarbol)
                  (hojas-cumplen-fos2 pred subarbol))
            (hijos-arbol arbol)))))

(check-equal? (hojas-cumplen-fos2 even? '(2)) '(2))

(check-equal? (hojas-cumplen-fos even? arbol3.1) '(2 4 2))
(check-equal? (hojas-cumplen-fos even? arbol3.2) '(2 4 2 6))

;----;
; b) ;
;----;

(define (todas-hojas-cumplen? pred arbol)
  (if (hoja-arbol? arbol)
      (pred (dato-arbol arbol))
      (todas-hojas-cumplen-bosque? pred (hijos-arbol arbol)))))

(define (todas-hojas-cumplen-bosque? pred bosque)
  (or (null? bosque)
      (and (todas-hojas-cumplen? pred (car bosque))
           (todas-hojas-cumplen-bosque? pred (cdr bosque))))))

(check-false (todas-hojas-cumplen? even? arbol3.1))
(check-true (todas-hojas-cumplen? even? arbol3.2))

; Suponemos que el árbol que se pasa como parámetro
; no es un árbol hoja

(define (todas-hojas-cumplen-fos? pred arbol)
  (for-all? (lambda (subarbol)
              (if (hoja-arbol? subarbol)
                  (pred (dato-arbol subarbol))
                  (todas-hojas-cumplen-fos? pred subarbol))) (hijos-arbol arbol)))

(check-false (todas-hojas-cumplen-fos? even? arbol3.1))
(check-true (todas-hojas-cumplen-fos? even? arbol3.2))

; Alternativa (aplicable a un árbol hoja) en la que se saca la comparación de
; si es hoja fuera de la expresión lambda
(define (todas-hojas-cumplen-fos2? pred arbol)
  (if (hoja-arbol? arbol)
      (pred (dato-arbol arbol))
      (for-all? (lambda (subarbol)
                  (todas-hojas-cumplen-fos2? pred subarbol)))
            (hijos-arbol arbol)))))

(check-false (todas-hojas-cumplen-fos2? even? '(3)))
(check-true (todas-hojas-cumplen-fos2? even? '(2)))

;-----;
```

```

; Ejercicio 4 ;
;-----;

;----;
; a) ;
;----;

(define (suma-raices-hijos arbol)
  (foldr + 0 (map dato-arbol (hijos-arbol arbol)))))

(define arbol4.1 '(20 (2)
                      (8 (4)
                          (2))
                      (9 (5))))
(check-equal? (suma-raices-hijos arbol4.1) 19)
(check-equal? (suma-raices-hijos (cadr (hijos-arbol arbol4.1))) 6)

;----;
; b) ;
;----;

(define (raices-mayores-arbol? arbol)
  (and (> (dato-arbol arbol) (suma-raices-hijos arbol))
       (raices-mayores-bosque? (hijos-arbol arbol)))))

(define (raices-mayores-bosque? bosque)
  (or (null? bosque)
      (and (raices-mayores-arbol? (car bosque))
           (raices-mayores-bosque? (cdr bosque))))))

(check-true (raices-mayores-arbol? arbol4.1))
(define arbol4.2 '(20 (2)
                      (8 (4)
                          (5))
                      (9 (5))))
(check-false (raices-mayores-arbol? arbol4.2))

; Solución 1 (usando for-all?)
(define (raices-mayores-arbol-fos? arbol)
  (and (> (dato-arbol arbol) (suma-raices-hijos arbol))
       (for-all? raices-mayores-arbol-fos? (hijos-arbol arbol)))))

(check-true (raices-mayores-arbol-fos? arbol4.1))
(check-false (raices-mayores-arbol-fos? arbol4.2))

; Solución 2 (usando foldl)
(define (raices-mayores-arbol-fos2? arbol)
  (foldl (lambda (subarbol resultado)
            (and resultado
                  (raices-mayores-arbol-fos2? subarbol)))
         (> (dato-arbol arbol) (suma-raices-hijos arbol))
         (hijos-arbol arbol)))))

(check-true (raices-mayores-arbol-fos2? arbol4.1))

```

```
(check-false (raices-mayores-arbol-fos2? arbol4.2))

(define arbol4.3 '(25 (2)
                        (10 (4)
                            (5 (6)))
                        (9 (5)))))

(check-false (raices-mayores-arbol? arbol4.3))

;----;
; c) ;
;----;

(define (comprueba-raices-arbol arbol)
  (nuevo-arbol (if (> (dato-arbol arbol) (suma-raices-hijos arbol))
                  1
                  0)
    (comprueba-raices-bosque (hijos-arbol arbol)))))

(define (comprueba-raices-bosque bosque)
  (if (null? bosque)
      '()
      (cons (comprueba-raices-arbol (car bosque))
            (comprueba-raices-bosque (cdr bosque)))))

;Alternativa FOS:
;(define (comprueba-raices-arbol arbol)
;  (nuevo-arbol (if (> (dato-arbol arbol) (suma-raices-hijos arbol))
;                  1
;                  0)
;                (map comprueba-raices-arbol (hijos-arbol arbol)))))

(check-equal? (comprueba-raices-arbol arbol4.1) '(1 (1)
                                                    (1 (1) (1))
                                                    (1 (1))))
(check-equal? (comprueba-raices-arbol arbol4.2) '(1 (1)
                                                    (0 (1) (1))
                                                    (1 (1)))))

;-----;
; Ejercicio 5 ;
;-----;

;----;
; a) ;
;----;

; Suponemos que lista tiene al menos un elemento
(define (es-camino? lista arbol)
  (and (equal? (car lista) (dato-arbol arbol))
       (es-camino-bosque? (cdr lista) (hijos-arbol arbol)))))

(define (es-camino-bosque? lista bosque)
```

```

(cond
  ((and (null? bosque) (null? lista)) #t)
  ((or (null? bosque) (null? lista)) #f)
  (else (or (es-camino? lista (car bosque))
             (es-camino-bosque? lista (cdr bosque))))))

; Alternativa con expresiones lógicas:
;(define (es-camino-bosque? lista bosque)
;  (or (and (null? lista) (null? bosque))
;       (and (not (null? lista))
;            (not (null? bosque))
;            (or (es-camino? lista (car bosque))
;                 (es-camino-bosque? lista (cdr bosque)))))))

; Alternativa FOS:
;(define (es-camino? lista arbol)
;  (and (equal? (car lista) (dato-arbol arbol))
;       (or (and (null? (cdr lista)) (hoja-arbol? arbol))
;            (exists? (lambda (subarbol)
;                      (es-camino? (cdr lista) subarbol))
;                    (hijos-arbol arbol))))))

(define arbol5.1 '(a (a (a)
                           (b))
                         (b (a)
                             (c))
                           (c)))
  (check-true (es-camino? '(a b a) arbol5.1))
  (check-false (es-camino? '(a b) arbol5.1))
  (check-false (es-camino? '(a b a b) arbol5.1))

;----;
; b) ;
;----;

(define (nodos-nivel nivel arbol)
  (if (= nivel 0)
      (list (dato-arbol arbol))
      (nodos-nivel-bosque (- nivel 1) (hijos-arbol arbol)))))

(define (nodos-nivel-bosque nivel bosque)
  (if (null? bosque)
      '()
      (append (nodos-nivel nivel (car bosque))
              (nodos-nivel-bosque nivel (cdr bosque)))))

;Alternativa FOS:
;(define (nodos-nivel nivel arbol)
;  (if (= nivel 0)
;      (list (dato-arbol arbol))
;      (foldl (lambda (subarbol lista)
;                (append lista (nodos-nivel (- nivel 1) subarbol)))
;              '()
;              (hijos-arbol arbol)))))


```

```

(define arbol5.2 '(1 (2 (3 (4
                            (2))
                            (5))
                            (6 (7))))
  (check-equal? (nodos-nivel 0 arbol5.2) '(1))
  (check-equal? (nodos-nivel 1 arbol5.2) '(2 6))
  (check-equal? (nodos-nivel 2 arbol5.2) '(3 5 7))
  (check-equal? (nodos-nivel 3 arbol5.2) '(4 2))
  (check-equal? (nodos-nivel 4 arbol5.2) '()))

;-----;
; Ejercicio 6 ;
;-----;

(define (camino-arbolb arbolb camino)
  (cond ((or (null? camino) (vacío-arbolb? arbolb))
         '())
        ((equal? (car camino) '<)
         (camino-arbolb (hijo-izq-arbolb arbolb) (cdr camino)))
        ((equal? (car camino) '>)
         (camino-arbolb (hijo-der-arbolb arbolb) (cdr camino)))
        (else
         (cons (dato-arbolb arbolb)
               (camino-arbolb arbolb (cdr camino))))))

(define arbolb6 '(9 (5 (3 (1 () ())
                           (4 () ()))
                           (7 () ()))
                           (15 (13 (10 () ())
                                     (14 () ()))
                           (20 ()
                             (23 () ()))))))
  (check-equal? (camino-arbolb arbolb6 '(= < < = > =)) '(9 3 4))
  (check-equal? (camino-arbolb arbolb6 '(> = < < =)) '(15 10))

```

## PRACTICA 8

---

### Ejercicio 1

- a) Implementa en Swift la función recursiva `prefijos(prefijo:palabras:)` que recibe una cadena y un array de palabras. Devuelve un array de `Bool` con los booleanos resultantes de comprobar si la cadena es prefijo de cada una de las palabras de la lista.

Ejemplo:

```

let array = ["anterior", "antígona", "antena"]
let prefijo = "ante"
print("prefijos(prefijo: \(prefijo), palabras: \(array))")
print(prefijos(prefijo: prefijo, palabras: array))
// Imprime:
// prefijos(prefijo: ante, palabras: ["anterior", "antígona", "antena"])
// [true, false, true]

```

b) Implementa en Swift la función recursiva `parejaMayorParImpar(numeros:)` que recibe un array de enteros positivos y devuelve una pareja con dos enteros: el primero es el mayor número impar y el segundo el mayor número par. Si no hay ningún número par o impar se devolverá un 0.

```

let numeros = [10, 201, 12, 103, 204, 2]
print("parejaMayorParImpar(numeros: \(numeros))")
print(parejaMayorParImpar(numeros: numeros))
// Imprime:
// parejaMayorParImpar(numeros: [10, 201, 12, 103, 204, 2])
// (201, 204)

```

## Ejercicio 2

a) Implementa en Swift la **función recursiva** `compruebaParejas(_:funcion:)` con el siguiente perfil:

```
([Int], (Int) -> Int) -> [(Int, Int)]
```

La función recibe dos parámetros: un `Array` de enteros y una función que recibe un entero y devuelve un entero. La función devolverá un array de tuplas que contiene las tuplas formadas por aquellos números contiguos del primer array que cumplan que el número es el resultado de aplicar la función al número situado en la posición anterior.

Ejemplo:

```

func cuadrado(x: Int) -> Int {
    return x * x
}
print(compruebaParejas([2, 4, 16, 5, 10, 100, 105], funcion: cuadrado))
// Imprime [(2,4), (4,16), (10,100)]

```

b) Implementa en Swift la **función recursiva** `coinciden(parejas: [(Int,Int)], funcion: (Int)->Int)` que devuelve un array de booleanos que indica si el resultado de aplicar la función al primer número de cada pareja coincide con el segundo.

```

let array = [(2,4), (4,14), (4,16), (5,25), (10,100)]
func cuadrado(x: Int) -> Int {
    return x * x
}
print("Resultado coinciden: \(coinciden(parejas: array, funcion: cuadrado))\n")
// Imprime: Resultado coinciden: [true, false, true, true]

```

## Ejercicio 3

Supongamos que estamos escribiendo un programa que debe tratar movimientos de cuentas bancarias. Define un enumerado `Movimiento` con valores asociados con el que podamos representar:

- Depósito (valor asociado: `(Double)`)
- Cargo de un recibo (valor asociado: `(String, Double)`)
- Cajero (valor asociado: `(Double)`)

Y define la función `aplica(movimientos:[Movimiento])` que reciba un array de movimientos y devuelva una pareja con el dinero resultante de acumular todos los movimientos y un array de Strings con todos los cargos realizados.

Ejemplo:

```

let movimientos: [Movimiento] = [.deposito(830.0), .cargoRecibo("Gimnasio", 45.0),
    .deposito(400.0), .cajero(100.0), .cargoRecibo("Fnac", 38.70)]
print(aplica(movimientos))
//Imprime (1046.3, ["Gimnasio", "Fnac"])

```

## Ejercicio 4

Implementa en Swift un tipo enumerado recursivo que permita construir árboles binarios de enteros. El enumerado debe tener

- un caso en el que guardar tres valores: un `Int` y dos árboles binarios (el hijo izquierdo y el hijo derecho)
- otro caso constante: un árbol binario vacío

Llamaremos al tipo `ArbolBinario` y a los casos `nodo` y `vacio`.

Impleméntalo de forma que el siguiente ejemplo funcione correctamente:

```

let arbol: ArbolBinario = .nodo(8, .nodo(2, .vacio, .vacio), .nodo(12, .vacio,
    .vacio))

```

Implementa también la función `suma(arbolb:)` que reciba una instancia de árbol binario y devuelva la suma de todos sus nodos:

```
print(suma(arbolb: arbol))
// Imprime: 22
```

## Ejercicio 5

Implementa en Swift un tipo enumerado recursivo que permita construir árboles de enteros usando el mismo enfoque que en Scheme: un nodo está formado por un dato (un `Int`) y una colección de árboles hijos. Llamaremos al tipo `Arbol`.

Impleméntalo de forma que el siguiente ejemplo funcione correctamente:

```
/*
Definimos el árbol

    10
   / | \
  3  5  8
  |
  1

*/
let arbol1 = Arbol.nodo(1, [])
let arbol3 = Arbol.nodo(3, [arbol1])
let arbol5 = Arbol.nodo(5, [])
let arbol8 = Arbol.nodo(8, [])
let arbol10 = Arbol.nodo(10, [arbol3, arbol5, arbol8])
```

Implementa también la función `suma(arbol:cumplen:)` que reciba una instancia de árbol y una función `(Int) -> Bool` que comprueba una condición sobre el nodo. La función debe devolver la suma de todos los nodos del árbol que cumplan la condición.

Implementa la función usando la misma estrategia que ya utilizamos en Scheme de definir una función auxiliar `suma(bosque:cumplen:)` y una recursión mutua.

```
func esPar(x: Int) -> Bool {
    return x % 2 == 0
}

print("La suma del árbol es: \(suma(arbol: arbol10, cumplen: esPar))")
// Imprime: La suma del árbol genérico es: 18
```

## Ejercicio 6

a) Define la función `maxOpt(_ x: Int?, _ y: Int?) -> Int?` que devuelve el máximo de dos enterosopcionales. En el caso en que ambos sean `nil` se devolverá `nil`. En el caso en que uno sea `nil` y el otro no se devolverá el entero que no es `nil`. En el caso en que ningún parámetro sea `nil` se devolverá el mayor.

Ejemplo:

```
let res1 = maxOpt(nil, nil)
let res2 = maxOpt(10, nil)
let res3 = maxOpt(-10, 30)
print("res1 = \(String(describing: res1))")
print("res2 = \(String(describing: res2))")
print("res3 = \(String(describing: res3))")
// Imprime:
// res1 = nil
// res2 = Optional(10)
// res3 = Optional(30)
```

b1) Escribe una nueva versión del ejercicio 1b) que permita recibir números negativos y que devuelva una pareja de `(Int?, Int?)` con `nil` en la parte izquierda y/o derecha si no hay número impares o pares.

Ejemplo:

```
let numeros2 = [-10, 202, 12, 100, 204, 2]
print("parejaMayorParImpar2(numeros: \(numeros2))")
print(parejaMayorParImpar2(numeros: numeros2))
// Imprime:
// parejaMayorParImpar2(numeros: [-10, 202, 12, 100, 204, 2])
// (nil, Optional(204))
```

b2) Escribe la función `sumaMaxParesImpares(numeros: [Int]) -> Int` que llama a la función anterior y devuelve la suma del máximo de los pares y el máximo de los impares. El array de números tendrá como mínimo un elemento, por lo que el valor devuelto por la función será un `Int` (no será `Int?`).

```
print("sumaMaxParesImpares(numeros: \(numeros2))")
print(sumaMaxParesImpares(numeros: numeros2))
// Imprime:
// sumaMaxParesImpares(numeros: [-10, 202, 12, 100, 204, 2])
// 204
```

## SOLUCION PRACTICA 8

```
// Solución Práctica 8: Programación funcional en Swift (1)
```

```
/*
=====
Ejercicio 1
=====
*/



/*
1a) Función recursiva prefijos(prefijo:palabra:)

func prefijos(prefijo: String, palabras: [String]) -> [Bool] {
    if (palabras.isEmpty) {
        return []
    } else {
        let primera = palabras[0]
        let resto = Array(palabras.dropFirst())
        return [primera.hasPrefix(prefijo)] + prefijos(prefijo: prefijo, palabras:
resto)
    }
}

let array = ["anterior", "antígona", "antena"]
let prefijo = "ante"

print("prefijos(prefijo: \(prefijo), palabras: \(array))")
print(prefijos(prefijo: prefijo, palabras: array))



/*
1b) Función recursiva parejaMayorParImpar(numeros:)

func mayorParImpar(numero: Int, pareja: (Int, Int)) -> (Int, Int) {
    if (numero.isMultiple(of: 2)) {
        return (pareja.0, max(pareja.1, numero))
    } else {
        return (max(pareja.0, numero), pareja.1)
    }
}

func parejaMayorParImpar(numeros:[Int]) -> (Int, Int) {
    if (numeros.isEmpty) {
        return (0, 0)
    } else {
        let primero = numeros[0]
        let resto = Array(numeros.dropFirst())
        let parejaResto = parejaMayorParImpar(numeros: resto)
        return mayorParImpar(numero: primero, pareja: parejaResto)
    }
}

let numeros = [10, 201, 12, 103, 204, 2]
print("parejaMayorParImpar(numeros: \(numeros))")
print(parejaMayorParImpar(numeros: numeros))
```

```
/*
=====
Ejercicio 2
=====
*/



/*
2a) Función recursiva compruebaParejas(_:funcion:)
*/
func compruebaParejas(_ numeros: [Int], funcion: (Int) -> Int) -> [(Int, Int)] {
    if numeros.count <= 1 {
        return []
    } else {
        let primero = numeros[0]
        let segundo = numeros[1]
        let resto = Array(numeros.dropFirst())
        if funcion(primero) == segundo {
            return [(primero, segundo)] + compruebaParejas(resto, funcion:
funcion)
        } else {
            return compruebaParejas(resto, funcion: funcion)
        }
    }
}

func cuadrado(x: Int) -> Int {
    return x * x
}

let numeros2 = [2, 4, 16, 5, 10, 100, 105]
print("compruebaParejas(\(numeros2), funcion: cuadrado)")
print(compruebaParejas(numeros2, funcion: cuadrado))

/*
2b) Función recursiva coinciden(parejas: [(Int, Int)], funcion: (Int) -> Int)
*/
func coinciden(parejas: [(Int, Int)], funcion: (Int) -> Int) -> [Bool] {
    if parejas.isEmpty {
        return []
    } else {
        let primera = parejas[0]
        let resto = Array(parejas.dropFirst())
        return [funcion(primera.0) == primera.1] +
coinciden(parejas: resto, funcion: funcion)
    }
}

let parejas2 = [(2,4), (4,14), (4,16), (5,25), (10,100)]
print("coinciden(parejas: \((parejas2), funcion: cuadrado))")
print(coinciden(parejas: parejas2, funcion: cuadrado))
```

```
/*
=====
Ejercicio 3: aplica(movimientos:)
=====
*/



enum Movimiento {
    case deposito(Double)
    case cargoRecibo(String, Double)
    case cajero(Double)
}

func acumula(movimiento: Movimiento, total: (Double, [String])) -> (Double, [String]) {
    switch movimiento {
        case let .deposito(cantidad):
            return (total.0 + cantidad, total.1)
        case let .cargoRecibo(concepto, cantidad):
            return (total.0 - cantidad, [concepto] + total.1)
        case let .cajero(cantidad):
            return (total.0 - cantidad, total.1)
    }
}

func aplica(movimientos: [Movimiento]) -> (Double, [String]) {
    if movimientos.isEmpty {
        return (0, [])
    } else {
        let mov = movimientos[0]
        let resto = Array(movimientos.dropFirst())
        let restoAcumulado = aplica(movimientos: resto)
        return acumula(movimiento: mov, total: restoAcumulado)
    }
}

let movimientos: [Movimiento] = [.deposito(830.0), .cargoRecibo("Gimnasio", 45.0),
    .deposito(400.0), .cajero(100.0), .cargoRecibo("Fnac", 38.70)]
print("aplica(\(movimientos)): \(aplica(movimientos: movimientos))")
// Imprime (1046.3, ["Gimnasio", "Fnac"])



/*
=====
Ejercicio 4: ArbolBinario
=====
*/
indirect enum ArbolBinario {
    case nodo(Int, ArbolBinario, ArbolBinario)
    case vacio
}
```

```

func suma(arbolb: ArbolBinario) -> Int {
    switch arbolb {
        case let .nodo(dato, hijoIzq, hijoDer):
            return dato + suma(arbolb: hijoIzq) + suma(arbolb: hijoDer)
        case .vacio:
            return 0
    }
}

let arbol: ArbolBinario = .nodo(8, .nodo(2, .vacio, .vacio), .nodo(12, .vacio, .vacio))
print("La suma del árbol binario es: \(suma(arbolb: arbol))")

/*
=====
Ejercicio 5: ArbolGenérico
=====
*/

```

```

indirect enum Arbol {
    case nodo(Int, [Arbol])
}

```

```

/*
Definimos el árbol

    10
   / | \
  3  5  8
  |
  1
*/

```

```

func suma(arbol: Arbol, cumplen condicion: (Int) -> Bool) -> Int {
    switch arbol {
        case let .nodo(dato, hijos):
            if (condicion(dato)) {
                return dato + suma(bosque: hijos, cumplen: condicion)
            }
            return suma(bosque: hijos, cumplen: condicion)
    }
}

func suma(bosque: [Arbol], cumplen condicion: (Int) -> Bool) -> Int {
    if bosque.isEmpty {
        return 0
    } else {
        let primero = bosque[0]
        let resto = Array(bosque.dropFirst())
        return suma(arbol: primero, cumplen: condicion) + suma(bosque: resto, cumplen: condicion)
    }
}

```

```

        }
    }

// También se puede implementar la recursión usando
// la llamada a first del array que devuelve un opcional.
// Si existe primero se entra en la recursión y si no
// se devuelve el caso base

/*
func suma(bosque: [Arbol], cumplen condicion: (Int) -> Bool) -> Int {
    if let primero = bosque.first {
        let resto = Array(bosque.dropFirst())
        return suma(arbol: primero, cumplen: condicion) + suma(bosque: resto,
cumplen: condicion)
    } else {
        return 0
    }
}
*/

```

```

let arbol1 = Arbol.nodo(1, [])
let arbol3 = Arbol.nodo(3, [arbol1])
let arbol5 = Arbol.nodo(5, [])
let arbol8 = Arbol.nodo(8, [])
let arbol10 = Arbol.nodo(10, [arbol3, arbol5, arbol8])

func esPar(x: Int) -> Bool {
    return x % 2 == 0
}

print("La suma del árbol genérico es: \(suma(arbol: arbol10, cumplen: esPar))")

```

```

/*
=====
Ejercicio 6
=====
*/

```

```

/*
6a) función maxOpt(_ x: Int?, _ y: Int?) -> Int?
*/

```

```

func maxOpt(_ x: Int?, _ y: Int?) -> Int? {
    if let num1 = x, let num2 = y {
        return max(num1, num2)
    } else if let num1 = x {
        return num1
    } else if let num2 = y {
        return num2
    } else {
        return nil
    }
}

```

```
}

let res1 = maxOpt(nil, nil)
let res2 = maxOpt(10, nil)
let res3 = maxOpt(-10, 30)
print("res1 = \(String(describing: res1))")
print("res2 = \(String(describing: res2))")
print("res3 = \(String(describing: res3))")

/*
6b1) Función recursiva parejaMayorParImpar2(numeros: [Int]) -> (Int?, Int?)
*/

func mayorParImpar2(numero: Int, pareja: (Int?, Int?)) -> (Int?, Int?) {
    if (numero.isMultiple(of: 2)) {
        return (pareja.0, maxOpt(pareja.1, numero))
    } else {
        return (maxOpt(pareja.0, numero), pareja.1)
    }
}

func parejaMayorParImpar2(numeros:[Int]) -> (Int?, Int?) {
    if (numeros.isEmpty) {
        return (nil, nil)
    } else {
        let primero = numeros[0]
        let resto = Array(numeros.dropFirst())
        let parejaResto = parejaMayorParImpar2(numeros: resto)
        return mayorParImpar2(numero: primero, pareja: parejaResto)
    }
}

let numeros6 = [-10, 202, 12, 100, 204, 2]
print("parejaMayorParImpar2(numeros: \(numeros6))")
print(parejaMayorParImpar2(numeros: numeros6))

/*
6b2) Función recursiva sumaMaxParesImpares(numeros: [Int]) -> Int
*/

func sumaMaxParesImpares(numeros: [Int]) -> Int {
    let pareja = parejaMayorParImpar2(numeros: numeros)
    if let num1 = pareja.0, let num2 = pareja.1 {
        return num1 + num2
    } else if let num1 = pareja.0 {
        return num1
    } else {
        return pareja.1!
    }
}
```

```
print("sumaMaxParesImpares(numeros: \$(numeros6))")
print(sumaMaxParesImpares(numeros: numeros6))
```

## PRACTICA 9

---

### Ejercicio 1

a) Indica qué devuelven las siguientes expresiones:

a.1)

```
let nums = [1,2,3,4,5,6,7,8,9,10]
nums.filter{$0 % 3 == 0}.count
```

a.2)

```
let nums2 = [1,2,3,4,5,6,7,8,9,10]
nums2.map{$0+100}.filter{$0 % 5 == 0}.reduce(0,+)
```

a.3)

```
let cadenas = ["En", "un", "lugar", "de", "La", "Mancha"]
cadenas.sorted{$0.count < $1.count}.map{$0.count}
```

a.4)

```
let cadenas2 = ["En", "un", "lugar", "de", "La", "Mancha"]
cadenas2.reduce([]) {(res: [(String, Int)], c: String) -> [(String, Int)] in
    res + [(c, c.count)]}.sorted(by: {$0.1 < $1.1})
```

b) Explica qué hacen las siguientes funciones y pon un ejemplo de su funcionamiento:

b.1)

```
func f(nums: [Int], n: Int) -> Int {
    return nums.filter{$0 == n}.count
}
```

b.2)

```
func g(nums: [Int]) -> [Int] {
    return nums.reduce([], {(res: [Int], n: Int) -> [Int] in
        if !res.contains(n) {
            return res + [n]
        } else {
            return res
        }
    })
}
```

b.3)

```
func h(nums: [Int], n: Int) -> ([Int], [Int]) {
    return nums.reduce(([[],[]], {(res: ([Int],[Int]), num: Int) -> ([Int],[Int]) in
        if (num >= n) {
            return (res.0, res.1 + [num])
        } else {
            return ((res.0 + [num], res.1))
        }
    })
}
```

c) Implementa las siguientes funciones con funciones de orden superior.

c.1) Función `suma(palabras:contienen:)`:

```
suma(palabras: [String], contienen: Character) -> Int
```

que recibe una array de cadenas y devuelve la suma de las longitudes de las cadenas que contiene el carácter que se pasa como parámetro.

c.2) Función `sumaMenoresMayores(nums:pivote:)`:

```
sumaMenoresMayores(nums: [Int], pivote: Int) -> (Int, Int)
```

que recibe un array de números y un número pivot y devuelve una tupla con la suma de los números menores y mayores o iguales que el pivot.

## Ejercicio 2

Define un tipo enumerado con un árbol genérico, tal y como hicimos en el último ejercicio de la práctica anterior, que tenga como genérico el tipo de dato que contiene.

En el siguiente ejemplo vemos cómo debería poderse definir con el mismo tipo genérico un árbol de enteros y un árbol de cadenas:

```
let arbolInt: Arbol = .nodo(53, [.nodo(13, []), .nodo(32, []), .nodo(41, [.nodo(36, []), .nodo(39, [])])])
let arbolString: Arbol = .nodo("Zamora", [.nodo("Buendía", [.nodo("Albeza", []), .nodo("Berenguer", []), .nodo("Bolardo", [])]), .nodo("Galván", [])])
```

Define las funciones genéricas `toArray` y `toArrayFOS` que devuelvan un array con todos los componentes del árbol usando un recorrido *preorden* (primero la raíz y después los hijos). La primera la debes implementar con recursión mutua y la segunda usando funciones de orden superior.

Ejemplo:

```
print(toArray(arbol: arbolInt))
// Imprime: [53, 13, 32, 41, 36, 39]
print(toArrayFOS(arbol: arbolString))
// Imprime: ["Zamora", "Buendía", "Albeza", "Berenguer", "Bolardo", "Galván"]
```

### Ejercicio 3

Implementa en Swift la función `imprimirListadosNotas(alumnos:)` que recibe un array de tuplas, en donde cada tupla contiene información de la evaluación de un alumno de LPP (nombreAlumno, notaParcial1, notaParcial2, notaCuestionarios, añosMatriculacion) y que debe imprimir por pantalla los siguientes listados:

- listado 1: array ordenado por nombre del alumno (orden alfabético creciente)
- listado 2: array ordenado por la nota del parcial 1 (orden decreciente de nota)
- listado 3: array ordenado por la nota del parcial 2 (orden creciente de nota)
- listado 4: array ordenado por año de matriculación y nota del cuestionario (orden decreciente de año y nota)
- listado 5: array ordenado por nota que necesita obtener en el parcial 3 para aprobar la asignatura en la convocatoria de Junio (orden decreciente de nota necesaria)

Las ordenaciones hay que realizarlas usando la función `sorted`.

!!! Note "Nota" Para que los listados se muestren formateados con espacios, puedes usar la siguiente función (para ello también debes incluir el import que se indica)

```
```swift
import Foundation

func imprimirListadoAlumnos(_ alumnos: [(String, Double, Double, Double, Int)]) {
    print("Alumno  Parcial1  Parcial2  Cuest  Años")
    for alu in alumnos {
```

```

        alu.0.withCString {
            print(String(format:"%-10s %5.2f      %5.2f  %5.2f  %3d", $0,
alu.1,alu.2,alu.3,alu.4))
        }
    }
```

```

Ejemplo:

```

let listaAlumnos = [("Pepe", 8.45, 3.75, 6.05, 1),
                    ("Maria", 9.1, 7.5, 8.18, 1),
                    ("Jose", 8.0, 6.65, 7.96, 1),
                    ("Carmen", 6.25, 1.2, 5.41, 2),
                    ("Felipe", 5.65, 0.25, 3.16, 3),
                    ("Carla", 6.25, 1.25, 4.23, 2),
                    ("Luis", 6.75, 0.25, 4.63, 2),
                    ("Loli", 3.0, 1.25, 2.19, 3)]
imprimirListadosNotas(listaAlumnos)

```

Algunos de los listados que se deben mostrar serían los siguientes:

**LISTADO ORIGINAL**

| Alumno | Parcial1 | Parcial2 | Cuest | Años |
|--------|----------|----------|-------|------|
| Pepe   | 8.45     | 3.75     | 6.05  | 1    |
| Maria  | 9.10     | 7.50     | 8.18  | 1    |
| Jose   | 8.00     | 6.65     | 7.96  | 1    |
| Carmen | 6.25     | 1.20     | 5.41  | 2    |
| Felipe | 5.65     | 0.25     | 3.16  | 3    |
| Carla  | 6.25     | 1.25     | 4.23  | 2    |
| Luis   | 6.75     | 0.25     | 4.63  | 2    |
| Loli   | 3.00     | 1.25     | 2.19  | 3    |

**LISTADO ORDENADO por Parcial1**

| Alumno | Parcial1 | Parcial2 | Cuest | Años |
|--------|----------|----------|-------|------|
| Maria  | 9.10     | 7.50     | 8.18  | 1    |
| Pepe   | 8.45     | 3.75     | 6.05  | 1    |
| Jose   | 8.00     | 6.65     | 7.96  | 1    |
| Luis   | 6.75     | 0.25     | 4.63  | 2    |
| Carmen | 6.25     | 1.20     | 5.41  | 2    |
| Carla  | 6.25     | 1.25     | 4.23  | 2    |
| Felipe | 5.65     | 0.25     | 3.16  | 3    |
| Loli   | 3.00     | 1.25     | 2.19  | 3    |

**LISTADO ORDENADO por Nota para aprobar en Junio**

| Alumno | Parcial1 | Parcial2 | Cuest | Años |
|--------|----------|----------|-------|------|
| Maria  | 9.10     | 7.50     | 8.18  | 1    |
| Jose   | 8.00     | 6.65     | 7.96  | 1    |
| Pepe   | 8.45     | 3.75     | 6.05  | 1    |

|        |      |      |      |   |
|--------|------|------|------|---|
| Carmen | 6.25 | 1.20 | 5.41 | 2 |
| Carla  | 6.25 | 1.25 | 4.23 | 2 |
| Luis   | 6.75 | 0.25 | 4.63 | 2 |
| Felipe | 5.65 | 0.25 | 3.16 | 3 |
| Loli   | 3.00 | 1.25 | 2.19 | 3 |

## Ejercicio 4

Dado el array `listaAlumnos` del ejercicio anterior, utiliza funciones de orden superior para obtener los datos requeridos en cada caso

A) Número de alumnos que han aprobado primer parcial y suspendido el segundo

```
print(listaAlumnos. _____ )
// Resultado: 5
```

B) Nota que deben tener en el parcial 3 para sacar un 5 en la nota final

```
print(listaAlumnos. _____ )
// Resultado:
// [1.8370370370388, -4.1407407407407391, -2.027777777777768,
7.061111111111127, 10.2777777777777777, 7.8851851851862, 8.088888888888903,
12.646296296296295]
```

C) Nota media de todos los alumnos en forma de tupla (`media_p1`, `media_p2`, `media_cuest`)

```
var tupla = listaAlumnos. _____ )
tupla = (tupla.0 / Double(listaAlumnos.count), tupla.1 /
Double(listaAlumnos.count), tupla.2 / Double(listaAlumnos.count))
print(tupla)
// Resultado: (6.681249999999995, 2.762499999999997, 5.226250000000003)
```

## Ejercicio 5

Implementa la función `construye` con el siguiente perfil:

```
func construye(operador: Character) -> (Int, Int) -> Int
```

La función recibe un operador que puede ser uno de los siguientes caracteres: `+`, `-`, `*`, `/` y debe devolver una clausura que reciba dos argumentos y realice la operación indicada sobre ellos.

Ejemplo:

```
var f = construye(operator: "+")
print(f(2,3))
// Imprime 5
f = construye(operator: "-")
print(f(2,3))
// Imprime -1
```

## SOLUCION PRACTICA 9

```
// Solución Práctica 9: Programación funcional en Swift: clausuras y funciones de
orden superior

/*
-----
Ejercicio 1: Cuestiones
-----
*/



print("Ejercicio 1")
print("=====")
print("--- Apartado a")

let nums = [1,2,3,4,5,6,7,8,9,10]
let resultado1 = nums.filter{$0 % 3 == 0}.count
print("""
  a.1) Solución: \(resultado1)
  """
)
// Devuelve la cantidad de números divisibles por 3: 3

let nums2 = [1,2,3,4,5,6,7,8,9,10]
let resultado2 = nums2.map{$0 + 100}.filter{$0 % 5 == 0}.reduce(0, +)
print("""
  a.2) Solución: \(resultado2)
  """
)
// Devuelve la suma de todos los números incrementados en 100 múltiplos de 5: 215

let cadenas = ["En", "un", "lugar", "de", "La", "Mancha"]
let resultado3 = cadenas.sorted{$0.count < $1.count}.map{$0.count}
print("""
  a.3) Solución: \(resultado3)
  """
)
// Devuelve la longitud de cada cadena del array ordenado de menor a mayor por
// la longitud de las cadenas: [2, 2, 2, 2, 5, 6]
```

```

let cadenas2 = ["En", "un", "lugar", "de", "La", "Mancha"]
let resultado4 = cadenas2.reduce([]) {
    (res: [String, Int], c: String) -> [(String, Int)] in
    res + [(c, c.count)]}.sorted(by: {$0.1 < $1.1})
print("""
a.3) Solución: \(resultado4)
)\n"""
)

// Devuelve un array de tuplas con cada cadena y su longigud, ordenado
// de menor a mayor por la longitud (segundo elemento de la tupla):
// [("En", 2), ("un", 2), ("de", 2), ("La", 2), ("lugar", 5), ("Mancha", 6)]

print("--- Apartado b")

func f(nums: [Int], n: Int) -> Int {
    return nums.filter{$0 == n}.count
}

// Explicación de funcionamiento:
// Dado un array de enteros y un número, devuelve el
// número de veces que se repite el número en el array

// Ejemplo:
print("""
b.1) Solución: Cuenta cuantos hay igual a n en un array de enteros.
f(nums: [1, 2, 3, 2, 4, 5, 2, 2], n: 2) = \(f(nums: [1, 2, 3, 2, 4, 5, 2, 2], n: 2) )\n"""
"""

func g(nums: [Int]) -> [Int] {
    return nums.reduce([], {(res: [Int], n: Int) -> [Int] in
        if !res.contains(n) {
            return res + [n]
        } else {
            return res
        }
    })
}

// Explicación de funcionamiento:
// Elimina duplicados de un array

// Ejemplo:

print("""
b.2) Solución: Elimina de un Array de enteros los valores repetidos.
g(nums: [1, 2, 3, 2, 4, 3, 2, 2]) = \(g(nums: [1, 2, 3, 2, 4, 3, 2, 2]) )\n"""
"""

func h(nums: [Int], n: Int) -> ([Int], [Int]) {

```

```

        return nums.reduce(([[],[]], {(res: ([Int],[Int]), num: Int ) -> ([Int],[Int])
in
            if (num >= n) {
                return (res.0, res.1 + [num])
            } else {
                return ((res.0 + [num], res.1))
            }
        })
    }

// Explicación de funcionamiento:
// Divide un array de enteros en dos partes: números menores que n
// y mayores o iguales que n

let numeros2 = [102, 23, 56, 231, 12, 452]
let pivote = 100
let dosArrays = h(nums: numeros2, n: pivote)

print("""
    b.3) Solución: Devuelve una pareja indicando cuantos elementos de un array de
    enteros
                    (nums) son menores y cuantos mayores o iguales que un valor dado
    (n).
        h(nums: [1, 2, 3, 4, 5], n: 3) = \(
            h(nums: [1, 2, 3, 4, 5], n: 3) )\n
    """)

print("Apartado c")

func suma(palabras: [String], contienen: Character) -> Int {
    return palabras.filter() {$0.contains(contienen)}.reduce(0) {$0 + $1.count}
}

print("""
    c.1)
        suma(palabras: \(cadenas), contienen: "a") = \(
            suma(palabras:    cadenas , contienen: "a") )\n
    """)

// c.2)
func sumaMenoresMayores(nums: [Int], pivote: Int) -> (Int, Int) {
    return nums.reduce((0,0)) {($1<pivote) ? ($0.0+$1, $0.1) : ($0.0, $0.1+$1)}
}

print("""
    c.2)
        sumaMenoresMayores(nums: \(nums), pivote: 8) = \(
            sumaMenoresMayores(nums:    nums , pivote: 8) )\n
    """)

/*
-----
Ejercicio 2: Arbol Genérico
-----
*/

```

```

indirect enum Arbol<T> {
    case nodo(T, [Arbol])
}

let arbolInt: Arbol = .nodo(53, [
    .nodo(13, []),
    .nodo(32, []),
    .nodo(41, [
        .nodo(36, []),
        .nodo(39, [])
    ])
])

let arbolString: Arbol = .nodo("Zamora", [
    .nodo("Buendía", [
        .nodo("Albeza", []),
        .nodo("Berenguer", []),
        .nodo("Bolardo", [])
    ]),
    .nodo("Galván", [])
])

/* Versión recursión mutua */

func toArray<T>(arbol: Arbol<T>) -> [T] {
    switch arbol {
        case let .nodo(dato, hijos):
            return [dato] + toArray(bosque: hijos)
    }
}

func toArray<T>(bosque: [Arbol<T>]) -> [T] {
    if let primero = bosque.first {
        let resto = Array(bosque.dropFirst())
        return toArray(arbol: primero) + toArray(bosque: resto)
    } else {
        return []
    }
}

print("Ejercicio 2")
print("=====")
print("-- Recursión mutua:")
print(toArray(arbol: arbolInt))
print(toArray(arbol: arbolString))

/* Versión FOS */

func toArrayFOS<T>(arbol: Arbol<T>) -> [T] {
    switch arbol {
        case let .nodo(dato, hijos):
            return hijos.map(toArrayFOS).reduce([dato], +)
    }
}

```

```
    }

}

print("-- FOS:")
print(toArrayFOS(arbol: arbolInt))
print(toArrayFOS(arbol: arbolString))

/*
-----
Ejercicio 3: Listado de notas
-----
*/



import Foundation

// Para hacer más legible las condiciones definimos
// el typealias en donde damos nombre a cada uno de
// los componentes de la tupla
typealias Calificacion = (nombre: String,
                           parcial1: Double,
                           parcial2: Double,
                           parcial3: Double,
                           años: Int)

func imprimirListadoAlumnos(_ alumnos: [Calificacion]) {
    print("Alumno  Parcial1  Parcial2  Parcial3  Años")
    for alu in alumnos {
        alu.0.withCString {
            print(String(format:"%-10s %5.2f      %5.2f      %5.2f  %3d", $0,
alu.1,alu.2,alu.3,alu.4))
        }
    }
}

func imprimirListadosNotas(_ alumnos: [Calificacion]) {
    var alumnosOrdenados: [Calificacion]

    print("LISTADO ORIGINAL")
    imprimirListadoAlumnos(alumnos)

    print("LISTADO ORDENADO por Nombre")
    alumnosOrdenados = alumnos.sorted(by:
        {a1, a2 in a1.nombre < a2.nombre})
    imprimirListadoAlumnos(alumnosOrdenados)

    print("LISTADO ORDENADO por Parcial1 (decreciente)")
    alumnosOrdenados = alumnos.sorted(by:
        {a1, a2 in a1.parcial1 > a2.parcial1})
    imprimirListadoAlumnos(alumnosOrdenados)

    print("LISTADO ORDENADO por Parcial2 (creciente)")
    alumnosOrdenados = alumnos.sorted(by: {$0.parcial2 < $1.parcial2})
    imprimirListadoAlumnos(alumnosOrdenados)
}
```

```

print("LISTADO ORDENADO por Año de Matriculación y Parcial3 (decreciente año y
nota)")
alumnosOrdenados = alumnos.sorted(by:
    {a1, a2 in
        if a1.años == a2.años {
            return a1.parcial3 < a2.parcial3
        } else {
            return a1.años > a2.años
        }
    })
imprimirListadoAlumnos(alumnosOrdenados)

print("LISTADO ORDENADO por Nota final (decreciente)")
// Nota para aprobar en Junio =
alumnosOrdenados = alumnos.sorted(by:
    {a1, a2 in
        let notaAlumno1 = a1.parcial1 * 0.34 +
                        a1.parcial2 * 0.33 +
                        a1.parcial3 * 0.34
        let notaAlumno2 = a2.parcial1 * 0.34 +
                        a2.parcial2 * 0.33 +
                        a2.parcial3 * 0.34
        return notaAlumno1 > notaAlumno2 })
imprimirListadoAlumnos(alumnosOrdenados)
}

// DEMOSTRACIÓN

print("Ejercicio 3")
print("===== ")
let listaAlumnos: [Calificacion] = [
    ("Pepe", 8.45, 3.75, 6.05, 1),
    ("Maria", 9.1, 7.5, 8.18, 1),
    ("Jose", 8.0, 6.65, 7.96, 1),
    ("Carmen", 6.25, 1.2, 5.41, 2),
    ("Felipe", 5.65, 0.25, 3.16, 3),
    ("Carla", 6.25, 1.25, 4.23, 2),
    ("Luis", 6.75, 0.25, 4.63, 2),
    ("Loli", 3.0, 1.25, 2.19, 3)]
imprimirListadosNotas(listaAlumnos)

/*
-----
Ejercicio 4: Expresiones de clausura
-----
*/
print("\nEjercicio 4")
print("===== \n")

// A) Número de alumnos que han aprobado primer parcial y suspendido el segundo
print(listaAlumnos.filter { $0.parcial1 >= 5 && $0.parcial2 < 5}.count)

```

```

// B) Alumnos que han aprobado la asignatura
print(listaAlumnos.map {
    (alumno) -> (String, Double) in
    let nota = alumno.parcial1 * 0.34 +
        alumno.parcial2 * 0.33 +
        alumno.parcial3 * 0.34
    return (alumno.nombre, nota)}
    .filter {$0.1 >= 5}
    .map {$0.0})

// C) Nota media de todos los alumnos: tupla (media_p1, media_p2, media_cuest)
var tupla = listaAlumnos.reduce((0,0,0), {(result, alumno) in
    return (result.0 + alumno.1, result.1 +
    alumno.2, result.2 + alumno.3)
})
tupla = (tupla.0 / Double(listaAlumnos.count), tupla.1 /
Double(listaAlumnos.count), tupla.2 / Double(listaAlumnos.count))
print(tupla)

/*
-----
Ejercicio 5: construye(operador:)
-----
*/
func construye(operador op: Character) -> (Int, Int) -> Int {
    return {(op1: Int, op2: Int) -> Int in
        switch op {
            case "*":
                return op1 * op2
            case "/":
                return op1 / op2
            case "-":
                return op1 - op2
            default:
                return op1 + op2
        }
    }
}

var f = construye(operador: "+")
print(f(2,3))
// Imprime 5
f = construye(operador: "-")
print(f(2,3))
// Imprime -1

```

## PRACTICA 10

---

### Ejercicio 1

a) ¿Qué se imprime al ejecutar el siguiente programa? Reflexiona sobre el funcionamiento del código, compruébalo con el compilador y experimenta haciendo cambios y comprobando el resultado.

```

var x = 0

func construye() -> () -> Int {
    var x = 10
    return {
        x = x + 5
        return x
    }
}

func usa(funcion: () -> Int) {
    var x = 20
    print(funcion())
}

let g = construye()
usa(funcion: g)
usa(funcion: g)

```

b) El siguiente código tiene errores, pero los comentarios indican correctamente su intención. Indica cuáles son los errores, cómo solucionarlos y qué se imprimirá al ejecutarlo. Después compruébalo con el compilador y experimenta haciendo cambios y comprobando el resultado.

```

var y = 0
var almacen: [() -> Int] = []

func usa2(funcion: () -> Int) {
    var y = 20
    almacen.append(funcion)
}

usa2 {y = y + 5
      return y}

// Obtenemos la clausura guardada en almacen
let h = almacen[0]()
// Invocamos a la clausura
print(h)
print(h)

```

c) El siguiente código usa observadores de propiedades y una variable del tipo (estática).

¿Qué se imprime al final de su ejecución? Reflexiona sobre el funcionamiento del código, compruébalo con el compilador y experimenta haciendo cambios y comprobando el resultado.

```

struct Valor {
    var valor: Int = 0 {
        willSet {
            Valor.z += newValue
        }
        didSet {
            if valor > 10 {
                valor = 10
            }
        }
    }
    static var z = 0
}

var c1 = Valor()
var c2 = Valor()
c1.valor = 20
c2.valor = 8
print(c1.valor + c2.valor + Valor.z)

```

## Ejercicio 2

Supongamos la siguiente clase **MisPalabras**:

```

class MisPalabras {
    var guardadas: [String] = []
    func guarda(palabra: String) {
        guardadas = guardadas + [palabra]
    }
}

let palabras = MisPalabras()
palabras.guarda(palabra: "Hola")
palabras.guarda(palabra: "me")
palabras.guarda(palabra: "llamo")
palabras.guarda(palabra: "Yolanda")
print(palabras.guardadas)
// ["Hola", "me", "llamo", "Yolanda"]

```

Debes añadir una **propiedad calculada** **logitud** que devuelva la suma de las longitudes de todas las palabras guardadas (usa una función de orden superior para calcular esta suma).

Haz también que sea una **propiedad modificable** de la siguiente forma:

- Si se intenta asignar un valor mayor o igual que la longitud de las cadenas guardadas, o un número negativo, no se hace nada.
- Si se asigna un valor menor que la longitud de las cadenas se deben dejar guardadas sólo las palabras que suman esa longitud, recortando la última de ellas si es necesario.

Ejemplo:

```
print(palabras.longitud)
// 18
palabras.longitud = 10
print(palabras.guardadas)
// ["Hola", "me", "llam"]
```

!!! Nota "Ayuda" Puedes utilizar la siguiente función `recorta` para recortar una palabra:

```
func recorta(_ palabra: String, hasta: Int) -> String {
    if hasta >= palabra.count {
        return palabra
    } else {
        let start = palabra.startIndex
        let end = palabra.index(start, offsetBy: hasta)
        return String(palabra[start..

```

### Ejercicio 3

En este ejercicio vamos a trabajar con estructuras y clases geométricas: `Punto`, `Tamaño`, `Rectangulo` y `Circulo`. Vamos a definir propiedades almacenadas y propiedades calculadas para todas las figuras geométricas.

- Para usar la función `sqrt` debes importar la librería `Foundation`:

```
import Foundation
```

- El valor de la constante matemática *pi* lo puedes obtener con la propiedad `Double.pi`.

#### **Estructuras Punto y Tamaño**

Las debes declarar tal y como aparecen en los apuntes.

#### **Estructura Rectangulo**

- Propiedades de instancia almacenadas:
  - **origen** (**Punto**) que contiene las coordenadas de la esquina inferior izquierda del rectángulo.
  - **tamaño** (**Tamaño**) que contiene las dimensiones del rectángulo.
- Propiedades de instancia calculadas:
  - **centro** (**Punto**, de lectura y escritura) que devuelve el centro del rectángulo. El **setter** modifica la posición del rectángulo manteniendo fijo su tamaño.
  - **area** (**Double**, sólo lectura ) que devuelve el área del rectángulo.

### Estructura **Circulo**

- Propiedades de instancia almacenadas:
  - **centro** (**Punto**) que contiene las coordenadas del centro del círculo.
  - **radio** (**Double**) que contiene la longitud del radio.
- Propiedades de instancia calculadas:
  - **area** (**Double**, de lectura y escritura) que devuelve el área del círculo. El **setter** modifica el tamaño del círculo (su radio), manteniéndolo en la misma posición.

### Estructura **AlmacenFiguras**

- Propiedades almacenadas:
  - **rectangulos** y **circulos** que contienen respectivamente arrays de rectángulos y círculos inicializados a arrays vacíos.
- Propiedades calculadas:
  - **numFiguras** (**Int**) que devuelve el número total de figuras creadas.
  - **areaTotal** (**Double**) que devuelve la suma total de las áreas de todas las figuras creadas.
- Métodos:
  - **añade(rectangulo:)** y **añade(circulo:)** que añaden un rectángulo y un círculo al array correspondiente.

!!!Note "Nota" La definición anterior del almacén de figuras no es demasiado correcta, porque se utiliza una variable distinta para cada tipo de figura, sin generalizar. En la práctica de la semana que viene veremos cómo mejorarlo utilizando protocolos.

Implementa las estructuras anteriores y escribe algún ejemplo de código en el que se creen al menos un rectángulo y un círculo, se prueben sus propiedades, se añadan al almacén de figuras y se prueben sus métodos.

### Ejercicio 4

Tenemos que escribir un programa que permita definir resultados de partidos de fútbol y calcular la puntuación de un conjunto de equipos una vez que se han jugado esos partidos.

Escribe código en Swift que permita resolver el problema, **utilizando structs**.

Un ejemplo de ejecución del código debería ser cómo sigue:

```
-----
Puntuación antes de los partidos:
Real Madrid: 0 puntos
Barcelona: 0 puntos
```

```
Atlético Madrid: 0 puntos
Valencia: 0 puntos
Athlétic Bilbao: 0 puntos
Sevilla: 0 puntos
-----
Resultados:
Real Madrid 0 - Barcelona 3
Sevilla 1 - Athlétic Bilbao 1
Valencia 2 - Atlético Madrid 1
-----
Puntuación después de los partidos:
Real Madrid: 0 puntos
Barcelona: 3 puntos
Atlético Madrid: 0 puntos
Valencia: 3 puntos
Athlétic Bilbao: 1 puntos
Sevilla: 1 puntos
```

## SOLUCION PRACTICA 10

```
// Solución práctica 10: Programación Orientada a Objetos en Swift (1)

import Foundation

/*
=====
Ejercicio 1
=====
*/
print("\nEjercicio 1\n")
print("a)")
// a)

// Variable global
var x = 0

func construye() -> () -> Int {
    // Variable x capturada por la clausura
    // que se crea en este ámbito
    var x = 10
    // Se construye la clausura y se devuelve.
    // Captura la variable x anterior
    return {
        x = x + 5
        return x
    }
}

func usa(funcion: () -> Int) {
```

```
// Variable local
// La clausura no la usa. La comentamos para evitar el warning del compilador
// var x = 20
// Se llama a la función y se imprime el resultado
print(funcion())
}

// Llamamos a construye y devuelve la clausura
let g = construye()
// Pasamos la clausura a la función usa
usa(funcion: g)
// Imprime 15
usa(funcion: g)
// Imprime 20

// b)
print("b)")

var array : [() -> Int] = []

func foo() -> Void {
    var x = 0
    array.append {
        x = x + 10
        return x
    }
}

foo()
foo()
print(array[0]()) // Imprime 10
print(array[0]()) // Imprime 20
print(array[1]()) // Imprime 10

/*
=====
Ejercicio 2
=====
*/
print("\nEjercicio 2\n")
print("a")

struct Valor {
    // Propiedad de la instancia
    var valor: Int = 0 {
        willSet {
            // Antes de actualizar el valor de la instancia
            // actualiza el valor de la propiedad z de Valor (propiedad del tipo)
            Valor.z += newValue
        }
        didSet {
            // Despues de actualizar el valor de la instancia
            // corrige su valor para que no sea mayor que 10
        }
    }
}
```

```
        if valor > 10 {
            valor = 10
        }
    }
static var z = 0
}

var c1 = Valor()
var c2 = Valor()
c1.valor = 20
c2.valor = 8
print(c1.valor + c2.valor + Valor.z)
// Imprime 46

/*
=====
Ejercicio 2
=====
*/



class Base {
    // Propiedades almacenadas
    var a = 100
    var b = 2
    var c = 10

    // Propiedad calculada
    var x: Int {a+b}

    // Método de instancia
    func suma() -> Int {
        return a+b
    }

    // Propiedad del tipo
    static var valor = 0

    // Método del tipo
    static func incrementa() -> Int {
        valor += 1
        return valor
    }
}

class Derivada: Base {
    // No podemos sobreescribir el valor de una propiedad almacenada
    // override var a = 200
    // Pero sí que podemos sobreescribir el getter y el setter
    override var a: Int {
        get {
            super.a + 1000
        }
    }
}
```

```
        }
        set(nuevoValor) {
            print("Actualizando valor de a con : \(nuevoValor)")
            super.a = nuevoValor
        }
    }

// Sí que podemos añadir un observador en la clase derivada
override var c: Int {
    willSet(nuevoValor) {
        print("Se va a cambiar el valor de c: \(nuevoValor)")
    }
}

// Sí que podemos sobreescribir el getter de una
// propiedad calculada y añadirle un setter
override var x: Int {
    get {a*b}
    set(nuevoValor) {
        a = nuevoValor
        b = nuevoValor
    }
}

// Sobreescribimos el método de instancia
// y dentro llamamos al método de la clase padre
override func suma() -> Int {
    let suma = super.suma()
    return suma + 1000
}
}

print("\nb")

var d = Derivada()
print(d.a) // Se ejecuta getter clase derivada - Imprime 1100 (super.a + 1000)
d.a = 400 // Se ejecuta setter clase derivada - Guarda 400 en super.a e imprime
mensaje
print(d.a) // Imprime 1400 (super.a + 1000)
print(d.x) // Se ejecuta getter clase derivada de la propiedad calculada x -
Imprime 2800 (a*b)
d.x = 0 // Se ejecuta setter clase derivada de la propiedad calculada x -
Actualiza con el valor 0 a y b
print(d.a) // Imprime 1000
print(d.b) // Imprime 0
d.c = 10 // Se ejecuta el observador de la propiedad c de clase derivada -
// Imprime "Se va a cambiar el valor de c a 10"

// Accedemos a la propiedad y método estático de la clase
// base en la clase derivada

print(Derivada.valor) // Imprime 0
print(Derivada.incrementa()) // Imprime 1
```

```

print(d.suma()) // Imprime 2000 (a + b + 1000)

/*
Resumen:
- ¿Se puede sobreescribir el valor de una propiedad almacenada? ¿Y calculada?
    Se pueden sobreescribir los getters y setters. Si la propiedad padre es de solo
    lectura, la propiedad sobreescrita puede ser de lectura y escritura. Una
    propiedad
        padre de lectura-escritura no se puede sobreescribir con una de sólo lectura.
        Se puede comprobar en las variables 'a' y 'x' de la clase derivada.
- ¿Se puede añadir un observador a una propiedad de la clase base en una clase
    derivada?
    Sí. Lo hemos hecho en la variable 'c' de la clase derivada.
- ¿Hereda la clase derivada propiedades y métodos estáticos de la clase base?
    Sí. Lo hemos comprobado accediendo a valor e incrementa() en la clase derivada
- ¿Cómo se puede llamar a la implementación de un método de la clase base en una
    sobreescritura
        de ese mismo método en la clase derivada?
        Usando 'super'. Lo hemos hecho en el método 'suma()' de la clase derivada.
*/

```

```

/*
=====
Ejercicio 3: Partidos de fútbol
=====

// /*
//     El ejercicio puede solucionarse de varias formas.
//     Proponemos a continuación una en la que se modela el problema
//     con tres estructuras:
//
//     Equipo - Contiene el nombre y los puntos de un equipo y un método
//             para actualizar las puntuaciones
//     Partido - Nombres y goles del partido
//     Liga - Contiene array de equipos y partidos jugados, así como los
//            métodos para actualizar una jornada
// */

struct Partido {
    let local : String
    let golesLocal : Int
    let visitante : String
    let golesVisitante : Int
}

struct Equipo {
    let nombre : String
    var puntos : Int

    mutating func actualizaPuntos(puntos: Int) {

```

```
        self.puntos += puntos
    }
}

struct Liga {
    var equipos: [Equipo]
    var partidos: [Partido]

    func mostrarPuntuaciones() {
        for e in equipos {
            print("\(e.nombre): \(e.puntos) puntos")
        }
    }

    func mostrarPartidos() {
        for p in partidos {
            print("\(p.local) \(p.golesLocal) - \(p.visitante) \
(p.golesVisitante)")
        }
    }

    func buscarEquipo(nombre:String) -> Int? {
        var i = 0
        var encontrado = false
        while !encontrado && i < equipos.count {
            encontrado = equipos[i].nombre == nombre
            if !encontrado {
                i+=1
            }
        }
        if !encontrado {
            return nil
        }
        else {
            return i
        }
    }

    func puntosPartido(favor:Int, contra:Int) -> Int {
        if favor > contra {
            return 3
        }
        else if favor == contra {
            return 1
        }
        else {
            return 0
        }
    }

    mutating func añadirPartidos(jornada:[Partido]) {
        self.partidos+=jornada
        actualizarPuntuaciones(jornada:jornada)
    }
}
```

```
    mutating func actualizarPuntuaciones(jornada:[Partido]) {
        for p in jornada {
            if let i = buscarEquipo(nombre:p.local) {
                equipos[i].actualizaPuntos(puntos:puntosPartido(favor:p.golesLocal,
                    contra:p.golesVisitante))
            }
            if let i = buscarEquipo(nombre:p.visitante) {
                equipos[i].actualizaPuntos(puntos:puntosPartido(favor:p.golesVisitante,
                    contra:p.golesLocal))
            }
        }
    }

// DEMOSTRACIÓN

print("\nEjercicio 3\n")

var liga: Liga

liga = Liga(equipos:[Equipo(nombre:"Real Madrid", puntos:0),
    Equipo(nombre:"Barcelona", puntos:0),
    Equipo(nombre:"Atlético Madrid", puntos:0),
    Equipo(nombre:"Valencia", puntos:0),
    Equipo(nombre:"Athlétic Bilbao", puntos:0),
    Equipo(nombre:"Sevilla", puntos:0)],
    partidos:[])

print("\nPuntuación antes de los partidos:")
liga.mostrarPuntuaciones()

var partidos = [Partido(local:"Real Madrid", golesLocal:0,
    visitante:"Barcelona", golesVisitante:3),
    Partido(local:"Sevilla", golesLocal:1,
    visitante:"Athlétic Bilbao", golesVisitante:1),
    Partido(local:"Valencia", golesLocal:2,
    visitante:"Atlético Madrid", golesVisitante:1),
    Partido(local:"Liverpool", golesLocal:4, visitante:"Barcelona",
    golesVisitante:0)]

liga.añadirPartidos(jornada:partidos)
print("\nResultados:")
liga.mostrarPartidos()

print("\nPuntuación después de los partidos:")
liga.mostrarPuntuaciones()

/*
```

```
=====
Ejercicio 4: Figuras geométricas
=====

*/
// Trabajamos con coordenadas de pantalla,
// donde la esquina superior izquierda tiene las coordenadas
// (0,0), la coordenada X crece hacia la derecha y la Y
// crece hacia abajo

// Función auxiliar que calcula el área de un triángulo

func areaTriangulo(_ p1: Punto, _ p2: Punto, _ p3: Punto) -> Double {
    let det = p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y) + p3.x * (p1.y - p2.y)
    return abs(det)/2
}

struct Punto {
    var x = 0.0, y = 0.0
}

struct Tamaño {
    var ancho = 0.0, alto = 0.0
}

class Figura {
    var origen: Punto
    var tamaño: Tamaño

    var area: Double {
        return tamaño.ancho * tamaño.alto
    }

    var centro: Punto {
        get {
            let centroX = origen.x + (tamaño.ancho / 2)
            let centroY = origen.y + (tamaño.alto / 2)
            return Punto(x: centroX, y: centroY)
        }
        set {
            origen.x = newValue.x - (tamaño.ancho / 2)
            origen.y = newValue.y - (tamaño.alto / 2)
        }
    }

    init(origen: Punto, tamaño: Tamaño) {
        self.origen = origen
        self.tamaño = tamaño
    }
}

class Cuadrilatero: Figura {
    var p1, p2, p3, p4: Punto
```

```
override var centro: Punto {
    get {
        super.centro
    }
    set {
        let incX = newValue.x - centro.x
        let incY = newValue.y - centro.y
        // Se llama al setter de la figura
        super.centro = newValue
        // Se actualiza las coordenadas de los puntos
        p1 = Punto(x: p1.x + incX, y: p1.y + incY)
        p2 = Punto(x: p2.x + incX, y: p2.y + incY)
        p3 = Punto(x: p3.x + incX, y: p3.y + incY)
        p4 = Punto(x: p4.x + incX, y: p4.y + incY)
    }
}

override var area: Double {
    let areaTriangulo1 = areaTriangulo(p1, p2, p3)
    let areaTriangulo2 = areaTriangulo(p3, p4, p1)
    return areaTriangulo1 + areaTriangulo2
}

init(p1: Punto, p2: Punto, p3: Punto, p4: Punto) {
    self.p1 = p1
    self.p2 = p2
    self.p3 = p3
    self.p4 = p4
    let minX = min(p1.x, p2.x, p3.x, p4.x)
    let minY = min(p1.y, p2.y, p3.y, p4.y)
    let maxX = max(p1.x, p2.x, p3.x, p4.x)
    let maxY = max(p1.y, p2.y, p3.y, p4.y)
    let alto = maxY - minY
    let ancho = maxX - minX
    super.init(origen: Punto(x: minX, y: minY), tamaño: Tamaño(ancho: ancho,
alto: alto))
}
}

class Circulo: Figura {
    var radio: Double {
        didSet {
            let incRadio = radio - oldValue
            origen.x -= incRadio
            origen.y -= incRadio
            let alto = radio * 2
            let ancho = radio * 2
            tamaño = Tamaño(ancho: ancho, alto: alto)
        }
    }
    override var area: Double {
        get {
            Double.pi * radio * radio
        }
    }
}
```

```
        }
        set {
            let radioCuadrado = newValue / Double.pi
            radio = radioCuadrado.squareRoot()
        }
    }

    init(centro: Punto, radio: Double) {
        self.radio = radio
        let minX = centro.x - radio
        let minY = centro.y - radio
        let alto = radio * 2
        let ancho = radio * 2
        super.init(origen: Punto(x: minX, y: minY), tamaño: Tamaño(ancho: ancho,
alto: alto))
    }
}

struct AlmacenFiguras {
    var figuras = [Figura]()

    var numFiguras: Int {
        return figuras.count
    }

    var areaTotal: Double {
        return figuras.reduce(0) {$0 + $1.area}
    }

    mutating func añade(figura: Figura) {
        figuras.append(figura)
    }

    mutating func desplaza(incX: Double, incY: Double) {
        for figura in figuras {
            let centro = figura.centro
            let nuevoCentro = Punto(x: centro.x + incX, y: centro.y + incY)
            figura.centro = nuevoCentro
        }
    }
}

print("\nEjercicio 4")

// Cuadrilatero

var cuadrilatero = Cuadrilatero(p1: Punto(x: 2, y: 6), p2: Punto(x: 5, y: 2),
                                 p3: Punto(x: 8, y: 7), p4: Punto(x: 3, y: 8))
print("\nCuadrilátero")
print("-----")
print("p1: \(cuadrilatero.p1), p2: \(cuadrilatero.p2)")
print("p3: \(cuadrilatero.p3), p4: \(cuadrilatero.p4)")
```

```
print("Origen: \u201corigen\u201d")
print("Tamaño: \u201ctamaño\u201d")
print("Centro: \u201ccentro\u201d")
print("Área: \u201carea\u201d")
let nuevoCentro = Punto(x: 10, y: 10)
print("Movemos el centro a la posición \u201cnuevoCentro\u201d")
cuadrilatero.centro = nuevoCentro
print("Origen: \u201corigen\u201d")
print("Centro: \u201ccentro\u201d")
print("Área: \u201carea\u201d")
print("p1: \u201cp1\u201d, p2: \u201cp2\u201d")
print("p3: \u201cp3\u201d, p4: \u201cp4\u201d")

// Círculo

var circulo = Círculo(centro: Punto(x: 12, y: 12), radio: 5)
print("\nCírculo")
print("-----")
print("Centro: \u201ccirculo.centro\u201d")
print("Radio: \u201ccirculo.radio\u201d")
print("Área: \u201ccirculo.area\u201d")
print("Origen: \u201ccirculo.origen\u201d")
print("Tamaño: \u201ccirculo.tamaño\u201d")
let nuevoRadio = 3.0
print("Nuevo radio: \u201cnuevoRadio\u201d")
circulo.radio = nuevoRadio
print("Radio: \u201ccirculo.radio\u201d")
print("Centro: \u201ccirculo.centro\u201d")
print("Origen: \u201ccirculo.origen\u201d")
print("Tamaño: \u201ccirculo.tamaño\u201d")
let nuevaArea = 100.0
circulo.area = nuevaArea
print("Nueva area: \u201cnuevaArea\u201d")
print("Centro: \u201ccirculo.centro\u201d")
print("Radio: \u201ccirculo.radio\u201d")
print("Área: \u201ccirculo.area\u201d")
print("Origen: \u201ccirculo.origen\u201d")
print("Tamaño: \u201ccirculo.tamaño\u201d")

// Almacén

var almacen = AlmacenFiguras()
almacen.añade(figura: cuadrilatero)
almacen.añade(figura: circulo)
print("Total áreas de las figuras: \u201calmacen.areaTotal\u201d")
almacen.desplaza(incX: 100, incY: 100)
print("Nuevos centros de las figuras:")
for figura in almacen.figuras {
    print("Centro: \u201cfigura.centro\u201d")
}
```

# Lenguajes y Paradigmas de Programación

---

## Datos académicos de la asignatura

**Departamento de Ciencia de la Computación e Inteligencia Artificial**

**6 créditos ECTS:** 1 clase de teoría de 2 h. y 1 clase de prácticas de 2 h. a la semana

**Profesores:**

- Antonio Botía ([e-mail](#)): Grupos de prácticas: 2, 9 y 401
- Domingo Gallardo ([e-mail](#)): Profesor coordinador de la asignatura. Grupos de teoría: 3, 4 y 40. Grupos de prácticas: 402
- Francisco Martínez ([e-mail](#)): Grupos de prácticas: 4, 5, 6 y 8
- Cristina Pomares ([e-mail](#)): Grupos de teoría: 1 y 2. Grupos de prácticas: 1, 3 y 7.

## Recursos de la asignatura

- [Ficha de la asignatura](#)
- [Apuntes de la asignatura \(teoría, seminarios y prácticas\)](#)
- [Sitio Moodle](#) abierto y accesible a toda la comunidad educativa, contiene los apuntes, transparencias, prácticas y otros materiales docentes
- [Foro de consultas y anuncios](#) en el sitio Moodle (sólo accesible a estudiantes)

## Objetivos y competencias

**Objetivos:**

- ¿Qué elementos son comunes a los lenguajes de programación?
- ¿Qué familias o paradigmas de lenguajes podemos identificar?
- ¿Qué es la programación funcional?
- ¿Cómo es un lenguaje multi-paradigma que combina la programación funcional y la programación orientada a objetos?

Dominando estos contenidos será mucho más fácil aprender nuevos lenguajes de programación, identificar sus aspectos esenciales e incluso ser capaz de diseñar lenguajes específicos orientados a dominios concretos.

**Competencias:**

- Conocer y diferenciar las características de los distintos paradigmas de programación (programación funcional, procedural y orientada a objetos) e identificarlas en lenguajes de programación concretos.
- Diferenciar entre tiempo de ejecución y tiempo de compilación en distintos ámbitos: detección de errores o definición, creación o ámbito de vida de variables.
- Conocer los principios básicos de la programación funcional: recursión, inmutabilidad, funciones como objetos de primera clase, funciones de orden superior, expresiones lambda (clausuras).
- Conocer los problemas derivados del uso de la mutación en los lenguajes de programación imperativos y la forma de trabajar con estructuras inmutables en lenguajes declarativos y funcionales.
- Utilizar la abstracción y la recursión para diseñar correctamente procedimientos y estructuras de datos (listas y árboles).

- Ser capaz de diseñar, implementar y corregir programas funcionales, en concreto utilizando el lenguaje de programación Scheme.
- Ser capaz de implementar programas sencillos en Swift, en los que se utilicen las características multi-paradigma del lenguaje.
- Comparar el paradigma orientado a objetos con el paradigma procedural clásico, reconociendo las ventajas que aporta en cuanto a abstracción, reutilización y modificación de código.

## Temario

### Bloques temáticos

La asignatura se divide en 3 bloques temáticos, todos ellos de igual duración, en los que se utilizará el lenguaje de programación que aparece entre paréntesis:

1. Programación funcional (Scheme): temas 1 y 2
2. Procesos y estructuras recursivas (Scheme): temas 3 y 4
3. Programación funcional en Swift y programación orientada a objetos (Swift): temas 5 y 6

Los lenguajes de programación se introducirán mediante seminarios impartidos en las clases de prácticas.

- Seminario 1. **El lenguaje de programación Scheme**: Primitivas. Tipos de datos básicos. Símbolos. Cadenas. Listas. Definición de funciones.
- Seminario 2. **El lenguaje de programación Swift**. Intérprete y scripts. Tipos de datos básicos. Operadores. Estructuras de control. Ámbito de variables. Tipos de datos compuestos: tuplas, arrays y colecciones. Recorriendo colecciones. Valores mutables e inmutables. Inicialización. Tipos de referencia y valor en Swift.

### Temas

- Tema 1. **Lenguajes de programación**: Historia de los lenguajes de programación. Elementos de los lenguajes de programación. Abstracción. Paradigmas de programación. Compiladores e intérpretes.
- Tema 2. **Programación Funcional**: Características e historia del paradigma de Programación Funcional. Diferencias con el paradigma imperativo. Características declarativas del paradigma funcional. Definición de funciones. Funciones como datos de primer orden. La forma especial lambda. Ámbito de variables y clausuras. Datos compuestos en Scheme: parejas. Construcción, recorrido y operaciones sobre listas. Listas con elementos compuestos. Listas de listas.
- Tema 3. **Procedimientos recursivos**: Diseño de funciones recursivas. Recursión mutua. Procesos recursivos e iterativos. Memoization. Recursión y gráficos de tortuga.
- Tema 4. **Estructuras recursivas**: Estructuras de datos recursivas: listas estructuradas y árboles.
- Tema 5. **Programación funcional en Swift**: Lenguajes multiparadigma. Programación funcional en Swift. Valoresopcionales. Listas. Recursión pura y recursión por la cola. Funciones como datos de primer orden. Clausuras y funciones anónimas. Funciones de orden superior: mappings y filtros de colecciones.
- Tema 6. **Programación Orientada a Objetos en Swift**: Características e historia del paradigma de Programación Orientada a Objetos. Estructuras y clases en Swift. Herencia. Conceptos avanzados de POO en Swift: Extensiones, Protocolos y Genéricos. Gestión de errores.

El calendario de temas, prácticas y exámenes se puede ver en la siguiente figura.

| Semana         | Clase de teoría  | Clase de prácticas   |
|----------------|--|--|
| 1 (27 enero)   | T0. Presentación asignatura.<br>T1. Historia lenguajes programación (para estudiar en casa).<br>T2.1. Prog. funcional, declarativa e imperativa. Modelos de evaluación de expresiones.<br>T2.2.1-2.2.4. Funciones y formas especiales. Formas especiales en Scheme. Forma especial quote y símbolos. Listas. | Seminario de Scheme  |
| 2 (3 febrero)  | T2.2.5.-2.2.6. Recursión. Recursión sobre listas.<br>T2.3. Tipos de datos compuestos en Scheme. Parejas. Box & pointer.  | Cuestionario 1: Seminario Scheme<br>P1: Introducción a Scheme. Modelos de evaluación. Listas.                |
| 3 (10 febrero) | T2.4. Listas. Funciones recursivas que construyen listas.<br>T2.5.1.- 2.5.2. Funciones como datos de primera clase. Forma especial lambda. Funciones como argumentos de otras funciones.   | Cuestionario 2: P1<br>P2: Recursión. Parejas. Box & pointer.   |
| 4 (17 febrero) | T2.5.3.-2.5.6. Funciones que devuelven funciones. Funciones en estructuras de datos. Generalización. Funciones de orden superior.  | Cuestionario 3: P2<br>P3: Listas. Recursión que construyen listas. Lambda y funciones que reciben funciones. |
| 5 (24 febrero) | T3 Procesos recursivos. El coste de la recursión. Procesos iterativos. Memoization. Recursión y gráficos de tortuga.   | Cuestionario 4: P3<br>P4: Funciones de orden superior  |
| 6 (2 marzo)    | <b>PARCIAL 1: Temas 1 y 2 (Jueves 5 marzo, 13:00 h.)</b>   | Cuestionario 5: P4<br>Repaso   |
| 7 (9 marzo)    | T4.1. Listas estructuradas. Funciones recursivas sobre listas estructuradas.   | P5: Procesos recursivos.   |
| 8 (16 marzo)   | Jueves: fiesta. Clases lunes y martes: repaso.   | P6: Listas estructuradas.  |
| 9 (23 marzo)   | T4.2. Árboles.<br>T4.3. Árboles binarios.  | Cuestionario 6: P5<br>P6: Listas estructuradas.  |
| 10 (30 marzo)  | T5.1.-5.5. Programación funcional con Swift (1). Introducción. Funciones. Tipos función. Recursión. Tipos.   | Cuestionario 7: P6<br>P7: Árboles.   |
| 11 (6 abril)   | Jueves: Semana Santa. Clases lunes y martes: seminario Swift.  | Seminario Swift  |
| 12 (20 abril)  | Lunes y jueves: fiesta. Clases martes: repaso.   | Repasso  |
| 13 (27 abril)  | <b>PARCIAL 2: Temas 3 y 4 (Jueves 30 abril, 13:00 h.)</b>  | Cuestionario 8: P7   |
| 14 (4 mayo)    | T5.6.-5.10. PF Swift (2). Opcionales. Inmutabilidad. Clausuras. Funciones de orden superior. Genéricos.  | Cuestionario 9: Seminario Swift<br>P8: Programación funcional con Swift (1)                                  |
| 15 (11 mayo)   | T6.7.-6.11. POO avanzada en Swift. Protocolos. Casting de tipos. Extensiones. Funciones operador. Genéricos.   | Cuestionario 10: P8<br>P9: Programación funcional con Swift (2)  |
| 16 (18 mayo)   | Repasso  | Cuestionario 11: P9<br>P10: POO avanzada en Swift  |
|                | <b>PARCIAL 3: Temas 5 y 6 (Miércoles 27 mayo)</b>  |  |

## Prácticas

Las prácticas son fundamentales en la asignatura y sirven para comprender, trabajar y profundizar los conceptos y competencias estudiados en las clases de teoría.

Para el desarrollo de las prácticas utilizaremos los siguientes lenguajes de programación y entornos de desarrollo:

- [Racket](#) (versión de Scheme, lenguaje de programación funcional)
- [Swift](#) (lenguaje multiparadigma creado por Apple, con conceptos modernos de programación funcional y programación orientada a objetos)

Cada práctica consistirá en la resolución de una **hoja de ejercicios** con 5 o 6 pequeños problemas de programación relacionados con la teoría vista durante la semana.

Se realizarán 11 sesiones de ejercicios de prácticas de una semana de duración y 2 seminarios sobre los anteriores lenguajes de programación.

La hoja de ejercicios estará disponible el viernes y se dispondrá de toda la semana siguiente para su realización y su entrega en Moodle. La entrega se cerrará el viernes por la noche.

En la clase de prácticas se trabajará en esa hoja de ejercicios. Durante la sesión de prácticas el profesor estará disponible para resolver dudas y dar pistas sobre cómo atacar los problemas. También durante la semana se podrán consultar las dudas que puedan surgir en el foro de la asignatura (en Moodle) y con tutorías a los profesores. Es preferible el foro, porque de esta forma las contestaciones y aclaraciones serán compartidas con el resto de compañeros.

Al final de la semana de realización de la hoja de ejercicios se publicará su solución.

## Cuestionarios

Al comienzo de cada sesión de prácticas se realizará un **quesionario sobre la hoja de ejercicios de la semana anterior**. La superación de los cuestionarios de cada bloque sumará 0,5 puntos a la nota del parcial de ese bloque (ver el apartado de evaluación).

Los cuestionarios constarán de 3 preguntas cortas **sobre la práctica y sobre los conceptos de teoría asociados**. Dentro de cada bloque temático deberás haber **aprobado los cuestionarios previos para poder realizar el cuestionario actual**. Se permitirá recuperar los cuestionarios previos suspendidos, **una vez por sesión** y siempre **dentro del tiempo** destinado a la realización de los mismos. Sólo se podrán recuperar cuestionarios del bloque temático actual.

El cuestionario se realizará en Moodle y se evaluará con una calificación de 0 a 10 (aprobando con una nota  $\geq 5$ ). Constará de 3 preguntas de igual puntuación (3,333 puntos). En cada pregunta se permitirán 3 intentos y cada intento fallado descontará 1/3 de la puntuación de la pregunta (1,111 puntos). De esta forma, para aprobar un cuestionario se podrán fallar como máximo 4 intentos.

Tendrás que tener cuidado al escribir las respuestas de los cuestionarios. Deberás escribirlas de forma totalmente correcta, **incluyendo los espacios en blanco**. Deberás fijarte en los ejemplos que proporcionamos en teoría y en los resultados de la ejecución de expresiones en el intérprete del lenguaje.

A continuación puedes ver ejemplos incorrectos y correctos de contestación de un cuestionario.

Dada la siguiente función:

```
(define (foo par)
  (cons (cons par par) par))
```

¿Qué devuelve Scheme al evaluar la siguiente expresión?

```
(foo 1)
```

Respuesta: ((1.1).1)



Respuesta: ((1.1) . 1)



Respuesta: ( (1 . 1) . 1)



**Respuestas incorrectas**

Respuesta: ((1 . 1) . 1)



Welcome to [DrRacket](#), version 7.5 [3m].

Language: racket, with debugging [custom]; memory limit: 128 MB.

```
> (define (foo par)
  (cons (cons par par) par))
> (foo 1)
((1 . 1) . 1)
> |
```

**Respuesta correcta**

## Normas para realización de los cuestionarios

- Cada cuestionario tendrá una duración máxima de 10 minutos.
- La realización de cuestionarios comenzará puntualmente a los 5 minutos de empezar la clase de prácticas y durará 20 minutos, para permitir recuperar algún cuestionario anterior suspendido.
- Deberás realizar los cuestionarios en el grupo de prácticas en el que está matriculado y en el aula en la que se imparte ese grupo de prácticas.
- Se deberán realizar en los ordenadores del aula. De forma excepcional, si no hay ordenadores libres, podrás utilizar tu portátil, pero sentado cerca de la mesa del profesor y con la pantalla en su dirección.

- Durante la realización del cuestionario únicamente deberás tener abierta la ventana del navegador en la que estás realizándolo. Está prohibido abrir cualquier otra ventana o aplicación.
- Durante la realización del cuestionario no deberás tener acceso al móvil.
- Una vez realizado un cuestionario, quedará bloqueado y no se podrá repetir hasta que haya pasado exactamente una semana. Por ello es muy importante que todas las semanas asistas al mismo grupo de prácticas. Una vez comenzados los cuestionarios no será posible realizar cambios de grupo en las prácticas.

## Comportamiento indebido en la realización de cuestionarios

Cualquiera de las siguientes acciones se consideran un comportamiento indebido, y se penalizarán con la anulación de todos los cuestionarios del bloque temático:

- Realizar un cuestionario fuera de tu grupo de prácticas.
- Tener abiertas otras aplicaciones mientras se está realizando el cuestionario.
- Realizar fotografías o capturas de pantallas de las preguntas del cuestionario.

## Uso del software iTALC

Durante el desarrollo de las sesiones prácticas es obligatorio usar el ordenador del aula y no el portátil personal. El profesor de prácticas abrirá el programa iTALC, con el que podrá comprobar cómo estás desarrollando los ejercicios y podrá haceros recomendaciones cuando lo considere oportuno. Además se utilizará iTALC para vigilar la contestación de los cuestionarios.

## Horarios

La distribución de grupos del curso 2019-20 es la siguiente:

## Grupos LPP 2019-20

|             | Lunes                            | Martes                             | Miércoles                          | Jueves                       | Viernes                    |
|-------------|----------------------------------|------------------------------------|------------------------------------|------------------------------|----------------------------|
| 8:30-9:00   |                                  |                                    |                                    |                              |                            |
| 9:00-9:30   | Grupo T2<br><b>Cristina</b>      |                                    |                                    | Grupo T1<br><b>Cristina</b>  |                            |
| 9:30-10:00  |                                  |                                    |                                    |                              |                            |
| 10:00-10:30 | Aulario 3 - 0001                 |                                    |                                    | Aulario 2 - D01              |                            |
| 10:30-11:00 | ARA                              |                                    |                                    |                              |                            |
| 11:00-11:30 | Grupo P3<br><b>Cristina</b>      |                                    | Grupo P7<br><b>Cristina</b>        | Grupo P1<br><b>Cristina</b>  | Grupo P9<br><b>Antonio</b> |
| 11:30-12:00 |                                  |                                    | L13                                | L17                          |                            |
| 12:00-12:30 | L23                              |                                    | Asociado a T1                      | Asociado a T1                | L23                        |
| 12:30-13:00 | Asociado a T2                    |                                    |                                    |                              | Asociado a T1              |
| 13:00-13:30 | Grupo T1 I2ADE<br><b>Domingo</b> | Grupo P401 I2ADE<br><b>Antonio</b> | Grupo P402 I2ADE<br><b>Domingo</b> | Grupo P2<br><b>Antonio</b>   |                            |
| 13:30-14:00 |                                  | L02                                | L13                                | L14                          |                            |
| 14:00-14:30 | Polivalente III - 006            |                                    |                                    | Asociado a T1                |                            |
| 14:30-15:00 |                                  |                                    |                                    |                              |                            |
| 15:00-15:30 | Grupo T3<br><b>Domingo</b>       |                                    |                                    | Grupo P5<br><b>Francisco</b> |                            |
| 15:30-16:00 |                                  |                                    |                                    | L23                          |                            |
| 16:00-16:30 | Aulario 2 - E14                  |                                    |                                    | Asociado a T4                |                            |
| 16:30-17:00 | Valenciano                       |                                    |                                    |                              |                            |
| 17:00-17:30 | Grupo P4<br><b>Francisco</b>     |                                    | Grupo P8<br><b>Francisco</b>       | Grupo T4<br><b>Domingo</b>   |                            |
| 17:30-18:00 | L13                              |                                    | L23                                | Aulario 2 - A01              |                            |
| 18:00-18:30 | Asociado a T3                    |                                    | Asociado a T4                      |                              |                            |
| 18:30-19:00 |                                  |                                    |                                    |                              |                            |
| 19:00-19:30 |                                  |                                    |                                    | Grupo P6<br><b>Francisco</b> |                            |
| 19:30-20:00 |                                  |                                    |                                    | L23                          |                            |
| 20:00-20:30 |                                  |                                    |                                    | Asociado a T4                |                            |
| 20:30-21:00 |                                  |                                    |                                    |                              |                            |

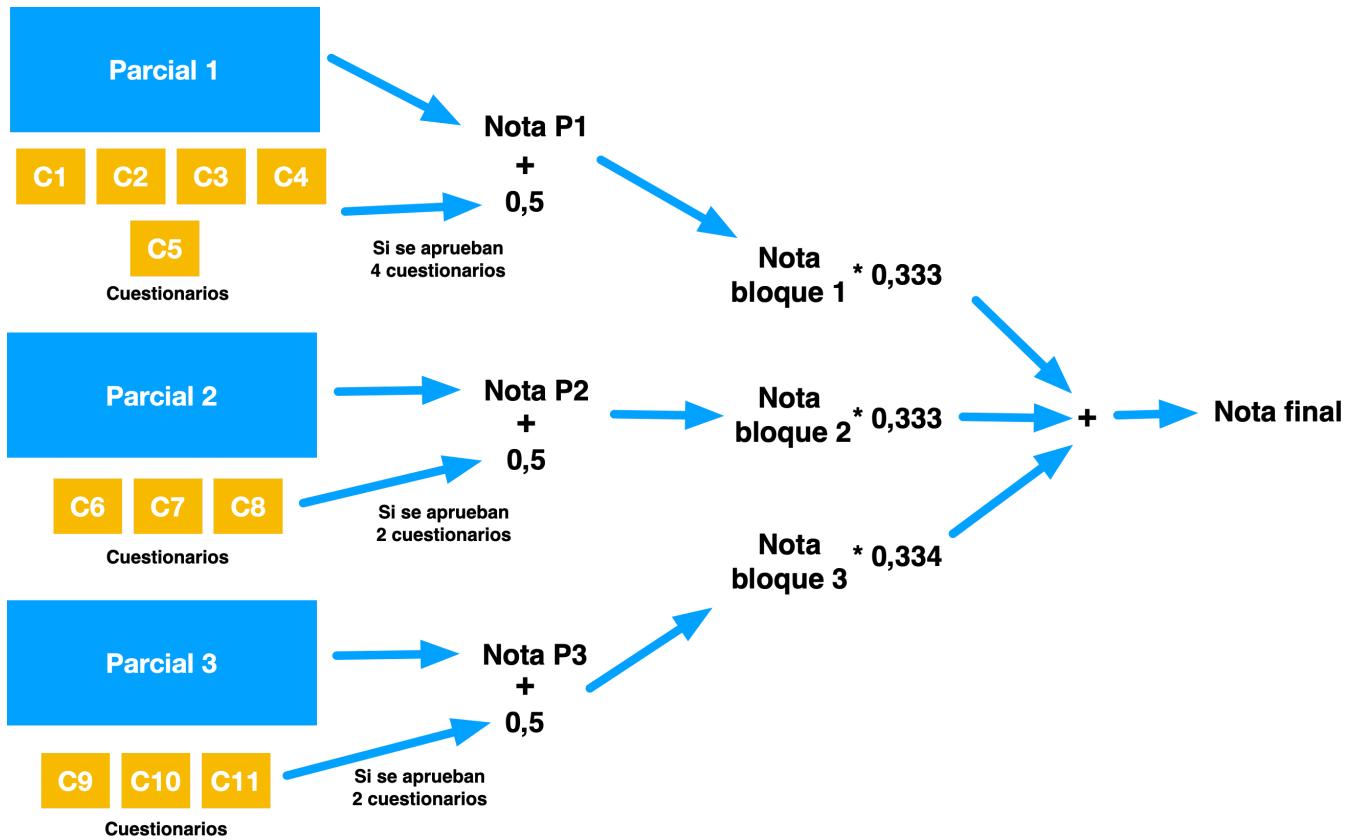
## Evaluación

Convocatoria normal C3 (evaluación continua)

Para cada bloque temático se realizará un examen parcial. En cada bloque temático se pueden obtener 0,5 puntos adicionales que se suman a la nota del parcial. Los puntos adicionales se obtienen si se aprueban n-1 cuestionarios del bloque.

Cada bloque pondrá 1/3 en la nota final. No hay nota mínima en ningún bloque.

La siguiente figura resume la evaluación:



!!! Danger "Sobre los dispositivos móviles" - Como norma básica de todas las pruebas de evaluación de esta asignatura (cuestionarios y exámenes escritos), durante la realización de los mismos no está permitido que llevéis encima ningún dispositivo con conexión a internet (smart phones, smart watches, tablets, etc). Antes de empezar la prueba, se deberán dejar dentro de las mochilas, y éstas en el suelo. En caso de no cumplir alguna esta norma, la prueba queda invalidada con calificación de 0.

## Convocatoria extraordinaria C4

En la convocatoria extraordinaria se realizará un examen final sobre todos los bloques temáticos cuya calificación representará el 100% de la nota de la asignatura.

## Consejos para aprender con éxito los contenidos de la asignatura

El consejo fundamental para aprobar la asignatura es **trabajar todas las semanas e intentar seguir el ritmo de la asignatura**. Los conceptos de la asignatura se van construyendo de forma progresiva y lo visto en una semana depende muchas veces de lo aprendido en semanas anteriores.

¿Cómo estudiar estos conceptos? Con la excepción de algunos temas y apartados concretos (como la historia de los lenguajes de programación o las características de los distintos paradigmas) la asignatura es fundamentalmente práctica. Cuando hablamos de *estudiar* los ejemplos de código queremos decir **entenderlos, no aprenderlos de memoria**. No tiene sentido aprender de memoria los ejercicios y los ejemplos vistos en clase. Hay que *trabajarlos*. Eso significa que, primero, hay que entenderlos sobre el papel y *muy importante* hay que **probar todos los ejemplos en el intérprete**. Y *probar* significa escribir los ejemplos y jugar con ellos, proponiendo pequeñas variantes, preguntándose *¿qué pasaría si...?* y probándolo.

En cuanto a las prácticas y a los ejercicios propuestos es fundamental pelearse con ellos e intentar hacerlos por uno mismo **sin ver ninguna solución**. Es la única forma de aprender: probando, equivocándose y encontrando la solución por uno mismo.

A la hora de enfrentarse con un problema es fundamental también **usar lápiz y papel** para probar enfoques y encontrar la solución más sencilla sobre el papel antes de probarla en el intérprete. Los ejercicios que proponemos no son excesivamente complicados. Todos se resuelven con muy pocas líneas de código y su codificación en el ordenador no tiene dificultad, una vez que se ha encontrado la solución que lo resuelve. Al usar el lápiz y papel también estarás practicando una situación similar a la que te vas a encontrar en los exámenes de la asignatura.

Resumiendo: trabajar todas las semanas, hacer uno mismo todos los ejercicios y no desanimarse ni descolgarse.

Son muy interesantes algunos comentarios de antiguos estudiantes que han aprobado la asignatura.

!!! quote "Cómo dominar los conceptos" Para superar la asignatura lo que hice fue estudiar mucho. Hay que practicar y sobre todo entender los ejercicios y no sabérselos de memoria. Una vez dominados los ejercicios yo mismo me propuse variantes de los mismos. Así es como se domina.

!!! quote "El problema del cambio de paradigma" El problema principal de la asignatura es enfrentarse a un cambio del paradigma de programación."

!!! quote "Trabajar día a día" Lo que hice fue tratar de llevar al día toda la asignatura, además de trabajar con material adicional para poder ampliar y profundizar conocimientos. LPP es una asignatura de fondo en la que tienes que mantener el ritmo de trabajo de principio a fin de cuatrimestre."

!!! quote "No copiar las prácticas" El mayor problema que creo que existe es que muchas personas se relajan y se copian las prácticas en cuanto les resultan un poco difíciles o les lleva algo mas del tiempo que les gustaría. Esta asignatura si no haces tu los ejercicios y te peleas con ellos es prácticamente imposible de sacar.

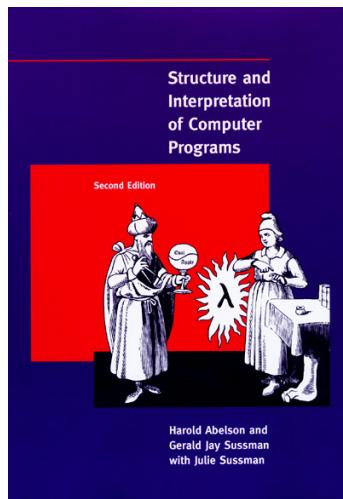
!!! quote "No memorizar" Otra de las cosas es que tienes que cambiar la forma de estudiar, no vale memorizar, ni hacer muchos ejercicios sin más. Tienes que entender bien el funcionamiento de la recursión para luego poder practicar con ejercicios, sino no sirve. [...] En mi opinión el problema de LPP para mucha gente es que para los exámenes se memorizan los ejercicios de prácticas de las soluciones que se dan en clase.

!!! quote "Plantearse uno mismo problemas" Los problemas que me encontré a la hora de cursar LPP fue que eran dos lenguajes completamente nuevos, otro tipo de programación que nunca había visto, otra forma de estudiar distinta. Para poder superarla simplemente tienes que hacer ejercicios y también plantearte tú mismo nuevos ejercicios.

## Bibliografía

En Moodle se publican los apuntes de la asignatura, con ejercicios, explicaciones y ejemplos de todos los conceptos estudiados, tanto en teoría como en práctica.

Para ampliar algunos conceptos se recomiendan los dos siguientes manuales:



- Harold Abelson y Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1996
- [Enlace a la edición on-line](#)
- Signatura en la Biblioteca Politécnica: I.06/ABE/STR



- Apple, *The Swift Programming Language (Swift 4.2)*, 2019
- [Web](#)

---

Lenguajes y Paradigmas de Programación, curso 2019-20

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante  
Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez

# Tema 1: Historia y conceptos de los lenguajes de programación

---

## Historia de los lenguajes de programación

De las máquinas de calcular a los computadores programables



Máquina de Schickard (1623)



Calculadora de Pascal (1650)

Desde las [primeras calculadoras mecánicas](#) diseñadas en el siglo XVII hasta los años 40 del siglo pasado se han inventado multitud de máquinas y computadores mecánicos, analógicos o electrónicos que han intentado acelerar y mejorar la precisión de los cálculos.

El punto culminante de todos los enfoques mecánicos para realizar cálculos fue el famoso **Motor Analítico** diseñado por **Charles Babbage** en 1840. La diferencia fundamental con todos los artefactos anteriores es que se trataba de una máquina de calcular **programable** mediante tarjetas perforadas (Babbage se inspiró en el telar de Jackquard en el que el diseño de los dibujos de las telas se podían configurar usando tarjetas perforadas). La máquina estaba diseñada para trabajar en base 10 y se podía conseguir que sus cálculos realizaran saltos condicionales y bucles.

Babbage trabajó durante más de 30 años para intentar construir la máquina. Tenía una enorme complejidad para la época y necesitaba muchísima financiación. En 1871 murió habiendo podido construir sólo una parte.

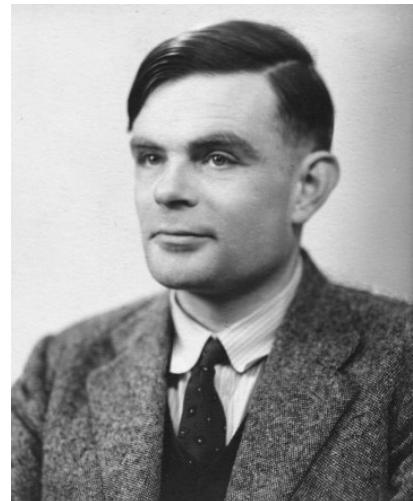
La matemática [Ada Lovelace](#) tuvo un papel fundamental en la divulgación de la máquina, de su sistema de programación y fue la primera que entendió sus posibilidades más allá del cálculo de fórmulas.

De forma poco habitual para la época, Ada fue educada en el campo de las ciencias y de las matemáticas. A principios de 1840, con veinticinco años, conoció el trabajo de Babbage y colaboró con él, dedicándose durante varios años a conocer y estudiar el diseño y el funcionamiento del motor analítico.

En 1843 publicó el trabajo "*Sketch of the analytical engine invented by Charles Babbage*" en el que describe el Motor Analítico, añade reflexiones propias sobre el alcance del invento y construye un ejemplo completo, con tablas y diagramas, de cómo hacer que la máquina produzca la secuencia de los números de Bernoulli. Se puede considerar estas tablas y diagramas como el **primer programa de un computador**.



Antes de que existiera ningún computador real, en 1936, el matemático inglés Alan Turing formalizó la idea abstracta de computador, utilizando un modelo muy sencillo de procesamiento: una máquina abstracta con un scanner que lee y escribe 0s y 1s de una cinta infinita (memoria) y se mueve y los escribe en función de una tabla definida en la máquina (programa). Con esta máquina abstracta (Máquina de Turing) Turing explora la idea de lo computable y lo no computable. ¿Existen problemas no computables para los que no es posible inventar un algoritmo que los resuelva? Turing demuestra que sí y establece con su trabajo los límites de la computación.



En el mismo trabajo Turing define el concepto de *máquina universal* que es capaz de leer de la cinta un programa cualquiera y simular su comportamiento en otra parte de la cinta. Esta idea tuvo un profundo impacto en el desarrollo de los computadores, porque mostraba que es posible escribir programas que tomen como datos otros programas. Esto abre la puerta a la idea de los programas almacenados en memoria (ya que son otros datos más) y a la creación de compiladores e intérpretes.

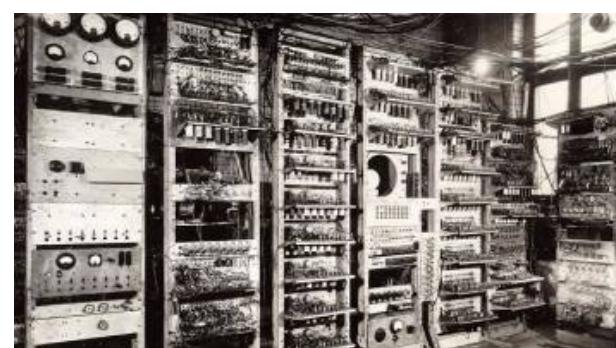
En la [década de 1940](#) hubo una explosión de máquinas de computación electrónicas y electromecánicas. Fue una década prodigiosa en la que se desarrollaron tecnologías cada vez más rápidas y resistentes, y se consiguieron enormes avances en la velocidad y precisión de los cálculos.

A mitad de esa década, en 1945, John Von Neumann, que trabajaba en la construcción del ENIAC, introdujo un avance fundamental. Propuso su famosa arquitectura en la que por primera vez se proponen las dos ideas claves de los computadores de propósito general: el programa almacenado en memoria y un conjunto de instrucciones de procesamiento que incluye el direccionamiento indirecto.



Y en 1948, tres años después, se construyó en la [universidad de Manchester](#) el primer computador electrónico digital de propósito general que utilizaba esta arquitectura (llamado Baby). Fue diseñado por Max Newmann usando la tecnología proporcionada por los ingenieros F.C. Williams y Tom Kilburn. Williams había inventado un dispositivo de memoria electrónico (la válvula de Williams) capaz de sustituir las lentes líneas de retraso de mercurio utilizadas hasta ese momento.

La máquina de Manchester fue el primer computador con un conjunto de instrucciones completo, capaz de realizar saltos, condicionales y direccionamiento indirecto. La primera ejecución de un programa fue el 21 de junio de 1948. En ese año Alan Turing se incorporó a la universidad de Manchester, como director del Laboratorio de Computación. Tres años después, con un diseño ampliado en el que también influyó Turing, una versión mucho mayor de la máquina se convirtió en el primer computador



disponible comercialmente, el Ferranti Mark I. El primero se instaló en la universidad de Manchester en febrero de 1951, un mes antes que el UNIVAC I fuera entregado al Departamento de Censo de los EEUU. Se vendieron otras 10 máquinas a Gran Bretaña, Canadá, Holanda e Italia.

El primer programa complejo de Inteligencia Artificial, un jugador de damas escrito por Christopher Strachey, se ejecutó en el verano de 1952 en el Ferranti Mark I en el Laboratorio de Computación de Manchester. Strachey escribió el programa animado por Turing y usando el manual de programación del Ferranti que Turing acababa de escribir. Turing participó también en el desarrollo de otros programas de IA, como un jugador de ajedrez basado en heurísticas.

## Los primeros lenguajes de programación

Los primeros computadores electrónicos se programaban directamente usando el conjunto de instrucciones del procesador, en código máquina, código hexadecimal

El primer lenguaje de un nivel algo más elevado que el código máquina es el ensamblador. Comienzan a crearse los primeros programas que procesan lenguajes de programación, aunque se trata de programas muy sencillos, ya que hay una relación casi directa entre la notación en ensamblador y el código hexadecimal que produce el ensamblador.

A finales de la década de los 40 se empiezan a intentar resolver con los primeros computadores los primeros problemas matemáticos distintos de operaciones numéricas: codificación y descodificación, problemas combinatorios como el coloreado del mapa o problemas de ordenación.

Uno de los primeros algoritmos de von Neumann realiza una ordenación de un conjunto de números. Von Neumann lo describe en una carta fechada en 1945. Utiliza el conjunto de instrucciones del EDSAC cuando todavía no se había construido. El programa fue estudiado por Donald Knuth en el artículo *Von Neumann's first Computer Program*, en donde documenta que había un bug en las primeras instrucciones. Es el primer bug escrito del que se tiene historia. Si Von Neumann hubiera podido ejecutar el programa en el EDSAC se hubiera dado cuenta del error y hubiera sido la primera depuración de un programa.

(5)

- (g) We now formulate a set of instructions to effect this 4-way decision between  $(\alpha) - (\delta)$ .  
We state again the contents of the short tanks already assigned:

$$\begin{array}{ll} 1.) \mathcal{N}_{n_{(20)}} & \bar{2}_1 \mathcal{W} m'_{(-20)} \quad \bar{3}_1 \mathcal{W} x_m^* \quad \bar{4}_1 \mathcal{W} y_m^*, \\ \bar{5}_1 \mathcal{N} m_{(20)} & \bar{6}_1 \mathcal{W} m'_{(-20)} \quad \bar{7}_1 \mathcal{W} l_{(\alpha-20)} \quad \bar{8}_1 \mathcal{W} l_{(\beta-20)} \\ \bar{9}_1 \mathcal{N} l_{(\epsilon-20)} & \bar{10}_1 \mathcal{W} l_{(\delta-20)} \quad \bar{11}_1 \dots \rightarrow C \end{array}$$

Now let the instructions occupy the (long tank) words  $1, 2, \dots$ :

$$\begin{array}{ll} 1.) T_1 - \bar{5}_1 & 0) \mathcal{N} m' - m_{(-20)} \\ 2.) \bar{9}_1 s \bar{7}_1 & 0) \mathcal{W} l_{\alpha}^{(20)} \quad \text{for } m' \leq m \\ 3.) O \rightarrow \bar{12}_1 & 12) \mathcal{N} l_{\alpha}^{(20)} \quad \text{for } m' \geq m \\ 4.) \bar{T}_1 - \bar{5}_1 & 0) \mathcal{W} m' - m_{(-20)} \\ 5.) \bar{10}_1 s \bar{8}_1 & 0) \mathcal{W} l_{\beta}^{(20)} \quad \text{for } m' \leq m \\ 6.) O \rightarrow \bar{12}_1 & 12) \mathcal{W} l_{\beta}^{(20)} \quad \text{for } m' \geq m \\ 7.) \bar{2}_1 - \bar{6}_1 & 0) \mathcal{W} m' - m_{(-20)} \\ 8.) \bar{12}_1 s \bar{9}_1 & 0) \mathcal{W} l_{\alpha}^{(20)} \dots \\ & \quad \text{i.e.} \\ & 0) \mathcal{W} l_{\alpha}^{(20)} l_{\beta}^{(20)} \quad \text{for } m' \leq m, m' \neq m \quad m' \leq m, m' \neq m \\ 9.) O \rightarrow \bar{11}_1 & \bar{11}_1 l_{\alpha} l_{\beta} l_{\delta} \rightarrow C \quad \text{i.e. for } \begin{cases} (2)(\alpha) \\ (2)(\beta) \\ (2)(\delta) \end{cases}, \text{ respectively.} \\ 10.) \bar{11}_1 \rightarrow C & \quad \text{for } (\alpha), (\beta), (\delta), \text{ respectively.} \end{array}$$

~~Below follows the first program of Von Neumann~~

Now

$$\bar{11}_1 l_{\alpha} l_{\beta} l_{\delta} \rightarrow C \quad \text{for } (\alpha), (\beta), (\delta), \text{ respectively.}$$

### Primer programa de Von Neumann

(Donald Knuth, "Von Neumann's first Computer Program", Journal of the ACM Computing Surveys (CSUR) Surveys, Volume 2 Issue 4, Dec. 1970, Pages 247-260)

### El nacimiento de los computadores comerciales

#### UNIVAC

El UNIVAC fue el primer computador comercial (1951). Con este computadora aparece por primera vez la figura del programador: manuales, cursos de formación, ofertas de empleo, etc.



## UNIVAC

### UNIVAC installations, 1951–1954<sup>[5]</sup>

| Date | Customer   | Comments   |
|------|--|--|
| 1951 | <a href="#">U.S. Census Bureau</a> , Suitland, MD                      | Not shipped until 1952                                     |
| 1952 | <a href="#">U.S. Air Force</a>   | <a href="#">Pentagon</a> , Arlington, VA                   |
| 1952 | <a href="#">U.S. Army Map Service</a>                                  | Washington, DC. Operated at factory April-September, 1952  |
| 1953 | <a href="#">New York University</a> (for the Atomic Energy Commission) | New York, NY   |
| 1953 | <a href="#">Atomic Energy Commission</a>                               | Livermore, CA  |
| 1953 | <a href="#">U.S. Navy</a>  | <a href="#">David W. Taylor Model Basin</a> , Bethesda, MD |
| 1954 | <a href="#">Remington Rand</a>   | Sales office, New York, NY                                 |
| 1954 | <a href="#">General Electric</a>                                       | Appliance Division, Louisville, KY. First business sale.   |
| 1954 | <a href="#">Metropolitan Life</a>                                      | New York, NY   |
| 1954 | <a href="#">U.S. Air Force</a>   | <a href="#">Wright-Patterson AFB</a> , Dayton, OH          |
| 1954 | <a href="#">U.S. Steel</a>   | Pittsburgh, PA   |
| 1954 | <a href="#">Du Pont</a>  | Wilmington, DE   |
| 1954 | <a href="#">U.S. Steel</a>   | Gary, IN   |
| 1954 | <a href="#">Franklin Life Insurance</a>                                | Springfield, IL  |
| 1954 | <a href="#">Westinghouse</a>   | Pittsburgh, PA   |
| 1954 | <a href="#">Pacific Mutual Life Insurance</a>                          | Los Angeles, CA  |
| 1954 | <a href="#">Sylvania Electric</a>                                      | New York, NY   |
| 1954 | <a href="#">Consolidated Edison</a>                                    | New York, NY   |

### Instalaciones comerciales del UNIVAC

## IBM 704

El IBM 704 fue el otro gran ordenador comercial de la década de los 50.

Tuvo una difusión mucho mayor que el UNIVAC: centros gubernamentales, universidades.

Los primeros lenguajes de programación de alto nivel se desarrollan para este computador.



Foto IBM 704

### Programando los primeros computadores

El UNIVAC I era una máquina interesante para programar, con su almacenamiento basado en líneas de retardo de mercurio y su propensión a fallar. Los programas se introducían en el computador tecleándolos en cintas magnéticas, una innovación importante en ese tiempo.

El trabajo con el IBM 704 en la Universidad de NY fue una experiencia radicalmente distinta de la del UNIVAC I. Fue construido para ejecutar aplicaciones científicas, y su principal innovación era una memoria de núcleo magnético, reemplazando la memoria de tubos Williams del IBM 701. También tenía una unidad aritmética en punto flotante. La máquina tenía el equivalente a 128 KB de memoria principal, 32 KB de memoria secundaria y cintas magnéticas que podía almacenar 5 MB de datos. Operaba a 0.04 MIPS y costó 3 millones de dólares en 1957.

George Sadowsky, [My Second Computer was a UNIVAC I](#)

### Los primeros lenguajes de alto nivel

Los primeros lenguajes de alto nivel se desarrollaron a finales de la década de los 50:

- [FORTRAN en 1956](#)
- [Lisp en 1958](#)

Ambos lenguajes planteaban dos enfoques muy distintos desde el principio:

- FORTRAN
  - [Primer lenguaje comercial, equipo de IBM dirigido por John W. Backus](#)
  - [Lenguaje imperativo:](#) estado, estructuras de control, contador de programa, celdas de memoria
  - [Lenguaje compilado](#)

- Lisp
  - Lenguaje diseñado en un departamento de investigación, un equipo del MIT dirigido por John McCarthy
  - Lenguaje funcional: funciones, recursión, listas, símbolos
  - Lenguaje interpretado

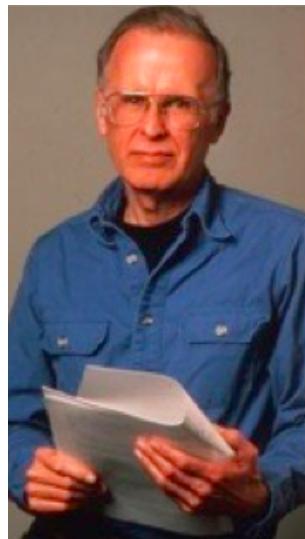
## FORTRAN

Desarrollado por IBM para programar el IBM 704. Algunos datos:

- Su nombre proviene de *FORmula TRANslating system*.
- El primer manual de FORTRAN se imprime en octubre de 1956 para el IBM 704.
- El primer compilador se comercializa en abril de 1956.

Cita de John Backus ([Wikipedia sobre FORTRAN](#)):

Una gran parte de mi trabajo viene del hecho de que soy perezoso. No me gustaba escribir programas, y cuando estaba trabajando en el IBM 701, escribiendo programas para calcular la trayectoria de misiles, comencé a trabajar en un sistema de programación que hiciera más fácil escribir programas.



John Backus

Example: It is required to compute the following quantities

$$P_i = \sqrt{\sin^2 (A_i B_i + C_i) + \cos^2 (A_i B_i - C_i)}$$

$$Q_i = \sin^2 (A_i + C_i) + \cos^2 (A_i - C_i)$$

for  $i = 1, \dots, 100$ . A possible FORTRAN program for this calculation follows.

| FOR<br>COMMENT      |                        | FORTAN STATEMENT                                   |
|---------------------|------------------------|--|
| STATEMENT<br>NUMBER | CONTINUATION<br>NUMBER |  |
| 1                   |                        | TRIGF(X, Y) = SINF (X+Y)**2+COSF(X-Y)**2           |
| 2                   |                        | DIMENSION A(100), B(100), C(100), P(100), Q(100)   |
| 3                   |                        | READ B, A, B, C                                    |
| 4                   |                        | DO 6 I = 1,100                                     |
| 5                   |                        | P(I) = SQRTF(TRIGF(A(I)*B(I), C(I)))               |
| 6                   |                        | Q(I) = TRIGF(A(I), C(I))                           |
| 7                   |                        | PRINT B, (A(I), B(I), C(I), P(I), Q(I), I = 1,100) |
| 8                   |                        | FORMAT (5F 10.4)                                   |
| 9                   |                        | STOP   |

### Ejemplo FORTRAN

Tomado del [manual de FORTRAN del IBM 704](#)

### Lisp

El otro lenguaje de alto nivel desarrollado en esa época es el Lisp. Desarrollado a finales de los [50 en el MIT por John McCarthy](#).

Aunque históricamente el nombre del lenguaje se solía escribir con letras mayúsculas (LISP), posteriormente se ha popularizado el uso de la mayúscula sólo para la primera letra (Lisp). Esta forma es más fiel al origen del nombre del lenguaje. *Lisp* no es un acrónimo, sino la contracción de la expresión *List Processing*. El procesamiento de listas es una de las características principales del Lisp.

McCarthy explica en un artículo de 1979 la historia inicial del Lisp:

[...] En el verano de 1956 durante el Dartmouth Summer Research Project on Artificial Intelligence, el primer estudio organizado de Inteligencia Artificial, tuve la idea de desarrollar un lenguaje algebraico para el procesamiento de listas. Quería usarlo para el desarrollo de trabajo en inteligencia artificial en el IBM 704. [...] John McCarthy, [History of LISP](#)



John McCarthy

Uno de los primeros manuales de Lisp publicados es el [manual de LISP](#) de 1960 para el IBM 704 escrito por Phyllis A. Fox del grupo de investigación del MIT dirigido por McCarthy.

Un ejemplo de código Lisp:

### 3. The Function LENGTH, defined Recursively

If we want to define LENGTH recursively, the expression is even simpler and easier:

To find the LENGTH of a list M:

If the list is null, take 0. Else,  
Add 1 to the length of CDR M.

The LISP defining expression is:

```
(LENGTH (LAMBDA (M) (COND
 0      1      2 2 2
  ((NULL M) 0)
 34     4      3
  (T (PLUS 1 (LENGTH (CDR M))))))
 3 4      5      6      6543210
```

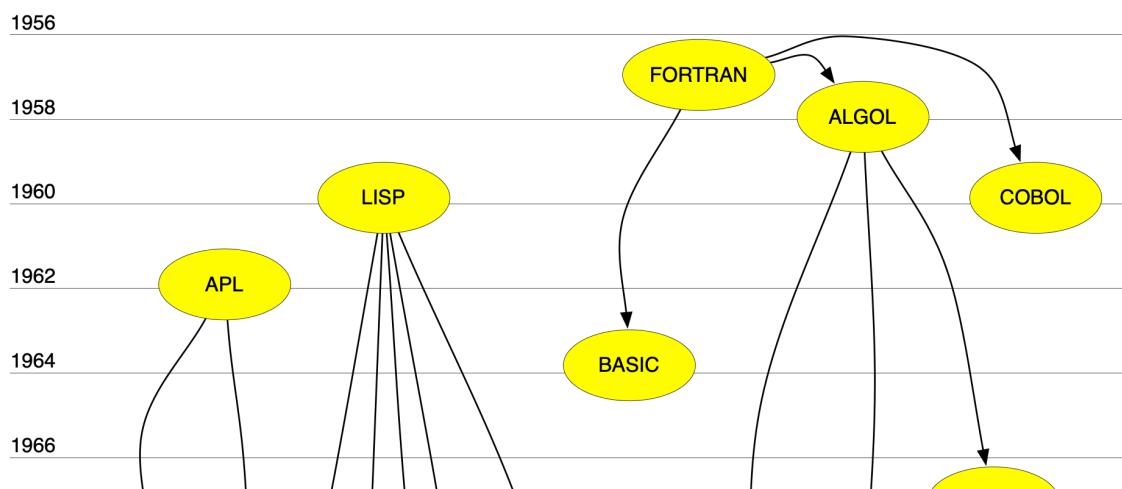
#### Ejemplo LISP

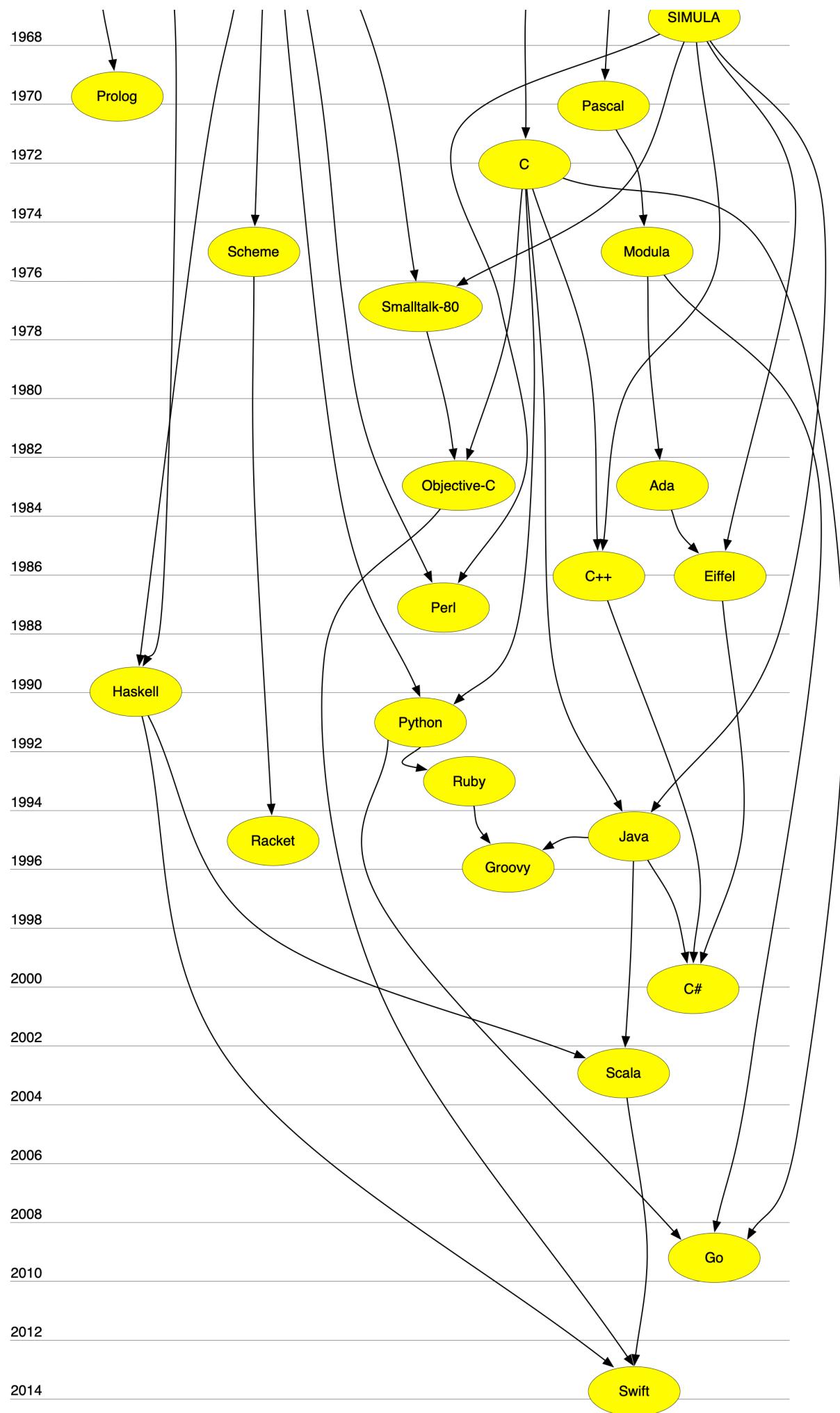
Tomado de "[The Programming Language LISP](#)", MIT Press, 1964

La explosión de los lenguajes de programación

Desde 1954 hasta la actualidad se han documentado más de 2.500 (consultar en [The Language List](#)). Entre 1952 y 1972 alrededor de 200 lenguajes. Una decena fueron realmente significativos y tuvieron influencia en el desarrollo de lenguajes posteriores.

#### Genealogía de los lenguajes de programación





## Genealogía de los LP

Algunas notas sobre la genealogía:

- APL es un lenguaje algebraico declarativo de especificación de funciones y circuitos lógicos. Su carácter declarativo ha tenido influencia en lenguajes como Prolog o Haskell.
- Lisp no sólo es un lenguaje funcional, sino que también es el primer lenguaje interpretado, con muchas características de tiempo de ejecución y poco chequeo estático. En esos aspectos ha influido en lenguajes dinámicos no funcionales como Python o Smalltalk. Lenguajes como Smalltalk o Objective-C también heredan de Lisp algunas características funcionales como la posibilidad de usar un bloque de código como un objeto primitivo que se crea en tiempo de ejecución y que se puede asignar o pasar como parámetro. Es lo que se denomina *clausura* en el paradigma de programación funcional.
- SIMULA es el primer lenguaje que define conceptos como clase u objeto. Es el origen de la programación orientada a objetos estática y fuertemente tipeada. Lenguajes como C++, Eiffel, o Java toman esta idea. Frente a esta tendencia se encuentra otra visión de la programación orientada a objetos de lenguajes como Smalltalk u Objective-C en la que se enfatiza más aspectos dinámicos como el paso de mensajes o la modificación de clases en tiempo de ejecución.

## Algunos lenguajes importantes y su fecha de creación

| 1950-1960    | 1970        | 1980              | 1990         | 2000        |
|--------------|-------------|-------------------|--------------|-------------|
| 1957 FORTRAN | 1970 Pascal | 1980 Smalltalk-80 | 1990 Haskell | 2000 C#     |
| 1958 ALGOL   | 1972 Prolog | 1983 Objective-C  | 1991 Python  | 2003 Scala  |
| 1960 Lisp    | 1972 C      | 1983 Ada          | 1993 Ruby    | 2003 Groovy |
| 1960 COBOL   | 1975 Scheme | 1986 C++          | 1995 Java    | 2009 Go     |
| 1962 APL     | 1975 Modula | 1986 Eiffel       | 1995 Racket  | 2014 Swift  |
| 1964 BASIC   |             | 1987 Perl         |              |             |
| 1967 SIMULA  |             |                   |              |             |

## Los creadores de los LPs

Si comprobamos la historia de los lenguajes de programación, podemos clasificar a sus creadores en tres grandes categorías:

- Investigadores trabajando en empresas (Backus/IBM-FORTRAN, Gosling/Sun-Java)
- Investigadores en universidades y departamentos de Informática (McCarthy/MIT-Lisp, Wirth/ETH-Pascal, Odersky/EHT-Scala)
- Desarrolladores open source que distribuyen su trabajo a la comunidad (Wall/Perl, Matsumoto/Ruby)

## Lenguajes de programación en la actualidad

El índice TIOBE es un indicador de la popularidad de los lenguajes de programación. El índice se actualiza una vez al mes. Las puntuaciones se basan en estadísticas no reveladas que incluyen el número de ingenieros en

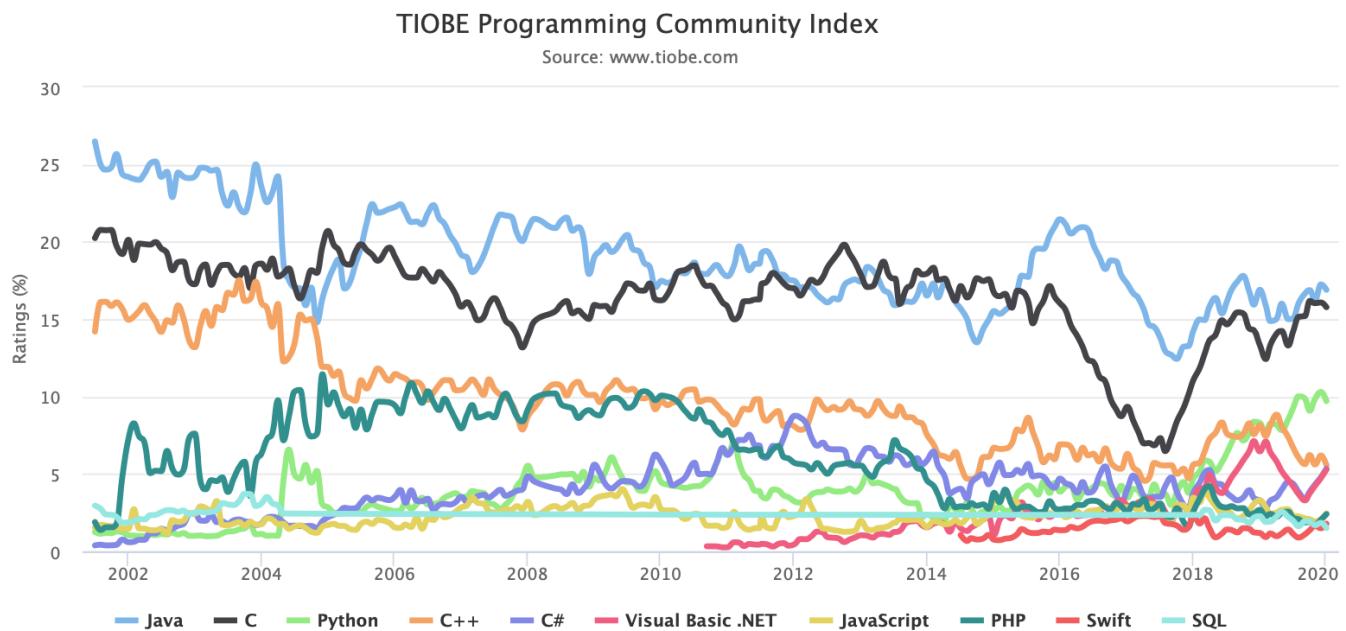
todo el mundo, cursos y aplicaciones desarrolladas. También se utilizan resultados obtenidos en los motores de búsqueda más usados.

El índice TIOBE no trata de medir el número de líneas escritas en los lenguajes de programación sino su *popularidad e importancia* en la comunidad.

| Jan 2020 | Jan 2019 | Change | Programming Language | Ratings | Change |
|----------|----------|--------|----------------------|---------|--------|
| 1        | 1        |        | Java                 | 16.896% | -0.01% |
| 2        | 2        |        | C                    | 15.773% | +2.44% |
| 3        | 3        |        | Python               | 9.704%  | +1.41% |
| 4        | 4        |        | C++                  | 5.574%  | -2.58% |
| 5        | 7        | ▲      | C#                   | 5.349%  | +2.07% |
| 6        | 5        | ▼      | Visual Basic .NET    | 5.287%  | -1.17% |
| 7        | 6        | ▼      | JavaScript           | 2.451%  | -0.85% |
| 8        | 8        |        | PHP                  | 2.405%  | -0.28% |
| 9        | 15       | ▲      | Swift                | 1.795%  | +0.61% |
| 10       | 9        | ▼      | SQL                  | 1.504%  | -0.77% |
| 11       | 18       | ▲      | Ruby                 | 1.063%  | -0.03% |
| 12       | 17       | ▲      | Delphi/Object Pascal | 0.997%  | -0.10% |
| 13       | 10       | ▼      | Objective-C          | 0.929%  | -0.85% |
| 14       | 16       | ▲      | Go                   | 0.900%  | -0.22% |
| 15       | 14       | ▼      | Assembly language    | 0.877%  | -0.32% |
| 16       | 20       | ▲      | Visual Basic         | 0.831%  | -0.20% |
| 17       | 25       | ▲      | D                    | 0.825%  | +0.25% |
| 18       | 12       | ▼      | R                    | 0.808%  | -0.52% |
| 19       | 13       | ▼      | Perl                 | 0.746%  | -0.48% |
| 20       | 11       | ▼      | MATLAB               | 0.737%  | -0.76% |

### *Lista TIOBE*

También es muy interesante comprobar la evolución de los 10 lenguajes más populares en los últimos 10 años.



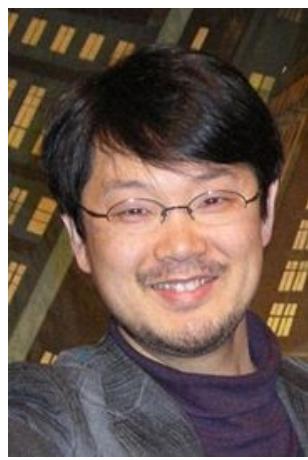
### Evolución TIOBE

#### La evolución no se detiene

Es interesante comprobar que cada vez es más fácil desarrollar nuevos lenguajes de programación. Las técnicas y herramientas de procesamiento de lenguajes se han popularizado cada vez más y son accesibles a mayor número de personas. Los lenguajes ya no sólo se crean en departamentos con un gran número de investigadores, sino también en comunidades open source formadas por voluntarios interesados y motivados

Ejemplos de nuevos lenguajes y sus creadores:

#### Ruby



*Yukihiro Matsumoto*

- [Ruby \(Wikipedia\)](#), un lenguaje de programación ideado en **1993** por el desarrollador japonés Yukihiro Matsumoto
- **Lenguaje multi-paradigma** interpretado y muy expresivo que actualmente se utiliza tanto para desarrollar **aplicaciones web como videojuegos**.
- Proyecto vivo, cada año aparecen nuevas versiones

## Scala



*Martin Odersky*

- [Scala \(Wikipedia\)](#), diseñado en 2003 por el profesor alemán Martin Odersky
- Respuesta a los problemas de los lenguajes tradicionales imperativos para manejar la concurrencia
- Está implementado sobre Java y corre en la Máquina Virtual Java

## Go



*Ken Thompson*

- [Go \(Wikipedia\)](#), el nuevo lenguaje de programación de Google lanzado en 2009
- Desarrollado, entre otros, por Ken Thompson, uno de los padres del UNIX
- Una mezcla de C y Python que intenta conseguir un lenguaje de programación de sistemas muy eficiente, expresivo y también multiparadigma.

## Swift



Chris Lattner

- [Swift \(Wikipedia\)](#), el nuevo lenguaje de programación de Apple lanzado en 2014
- [Proyecto open source](#) en el que se puede observar su [evolución y roadmap futuro](#)
- Desarrollado, entre otros, por [Chris Lattner](#), autor del sistema *LLVM Compiler Infrastructure*, conjunto de compilador, depurador, optimizador, etc. de código C, C++ y Objective-C.
- Lenguaje moderno, multiparadigma (programación orientada a objetos y funcional) fuertemente tipado y compilado.

## Elementos de los lenguajes de programación

Definición de la Encyclopedia of Computer Science

Un lenguaje de programación es un conjunto de caracteres, reglas para combinarlos y reglas para especificar sus efectos cuando sean ejecutados por un computador, que tienen las siguientes cuatro características:

1. No requiere ningún conocimiento de código máquina por parte del usuario.
2. Tiene independencia de la máquina.
3. Se traduce a lenguaje máquina.
4. Utiliza una notación que es más cercana al problema específico que se está resolviendo que al código máquina.

Definición de Abelson y Sussman

Vamos a estudiar la idea de un **proceso computacional**. Los procesos computacionales son seres abstractos que habitan los computadores. Cuando están en marcha, los procesos manipulan otras cosas abstractas denominadas **datos**. La evolución de un proceso está dirigida por un patrón de reglas denominado un programa. [...]

Y otra idea fundamental

Un lenguaje de programación potente es más que sólo una forma de pedir a un computador que realice tareas. El lenguaje también sirve como un marco dentro del que organizamos nuestras ideas acerca de los procesos. Así, cuando describimos un lenguaje, deberíamos prestar atención particular a los medios que proporciona el lenguaje para combinar ideas simples para formar ideas más complejas.

## Características de un LP

1. Define un proceso que se ejecuta en un computador
2. Es de alto nivel, cercano a los problemas que se quieren resolver (abstracción)
3. Permite construir nuevas abstracciones que se adapten al dominio que se programa

## Elementos de un LP

Para Abelson y Sussman, todos los lenguajes de programación permiten combinar ideas simples en ideas más complejas mediante los siguientes tres mecanismos

- **Expresiones primitivas** que representan las entidades más simples del lenguaje
- **Mecanismos de combinación** con los que se construyen elementos compuestos a partir de elementos más simples

- **Mecanismos de abstracción** con los que dar nombre a los elementos compuestos y manipularlos como unidades

## Sintaxis y semántica

*Sintaxis:* conjunto de reglas que definen qué expresiones de texto son correctas. Por ejemplo, en C todas las sentencias deben terminar en ';'.

*Semántica:* conjunto de reglas que define cuál será el resultado de la ejecución de un programa en el computador.

## Los lenguajes son para las personas

Los lenguajes de programación deben ser precisos, deben poder traducirse sin ambigüedad en lenguaje máquina para que sean ejecutados por computadores. Pero deben ser utilizados (leídos, comentados, probados, etc.) por personas.

La programación es una actividad colaborativa y debe basarse en la comunicación.

## Importancia del aprendizaje de técnicas de LPs

Es importante conocer cómo funciona "por dentro" un lenguaje de programación y sus características comparadas.

- Mejora el uso del lenguaje de programación
- Incrementa el vocabulario de los elementos de programación
- Permite una mejor elección del lenguaje de programación
- Mejora la habilidad para desarrollar programas efectivos y eficientes
- Facilita el aprendizaje de un nuevo lenguaje de programación
- Facilita el diseño de nuevos lenguajes de programación

## Abstracción

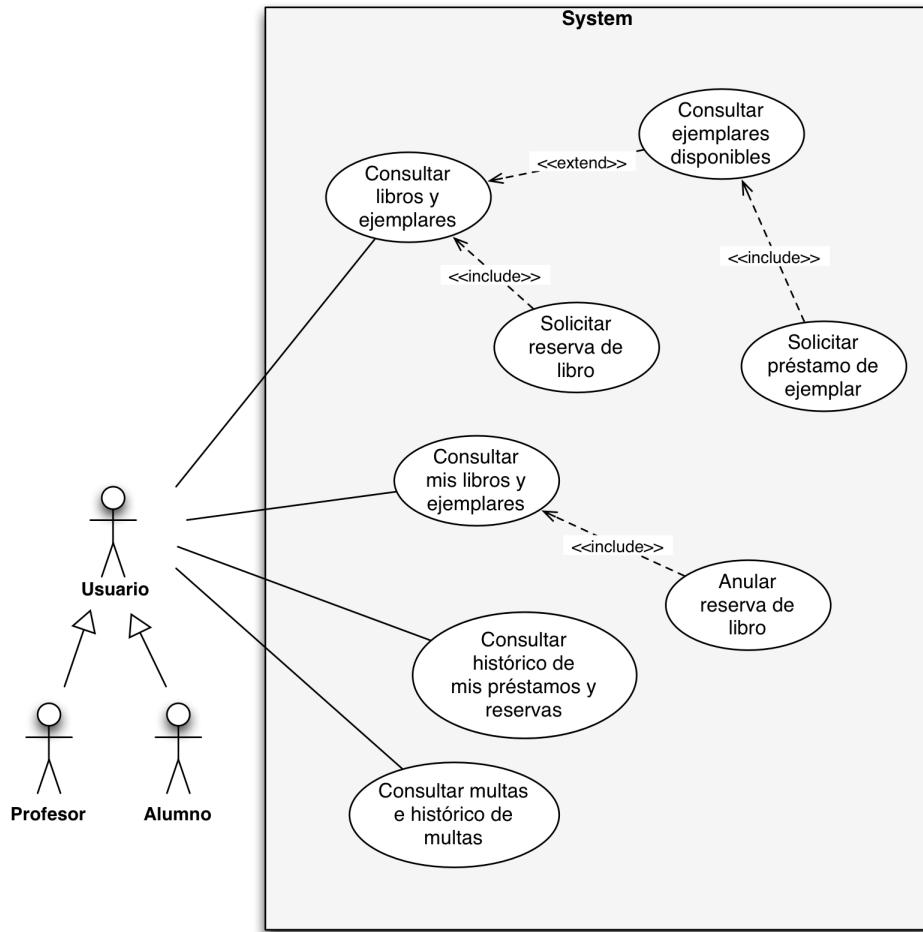
Una misión fundamental de los lenguajes de programación es proporcionar herramientas que sirvan para construir abstracciones. Por ejemplo, estamos construyendo una abstracción cuando damos un **nombre** a una **entidad del lenguaje** (una variable, una función, una clase, etc.).

Escoger un buen nombre para los elementos que vamos construyendo en nuestro programas es fundamental para conseguir un código legible y reutilizable.

## Modelar como una actividad fundamental

- Para escribir un programa que preste unos servicios es fundamental modelar el dominio sobre el que va a trabajar
- Es necesario definir distintas abstracciones (tanto APIs, como datos) que nos permitan tratar sus elementos y comunicarnos correctamente con los usuarios que van a utilizar el programa.
- Las abstracciones que vamos construyendo van apoyándose unas en otras y permiten hacer comprensible y comunicable un problema complejo
- Ejemplo: el **modelado del funcionamiento de una biblioteca** contiene abstracciones como "libros", "préstamo", "reserva", o "libros disponibles" que representan conceptos del dominio que deben ser

implementados en nuestra solución



### Casos de uso biblioteca

### Abstracciones computacionales

Existen abstracciones propias de la informática (*computer science*), que se utilizan en múltiples dominios. Por ejemplo, abstracciones de datos como:

- Listas
- Árboles
- Grafos
- Tablas hash

También existen abstracciones que nos permiten tratar con dispositivos y ordenadores externos:

- Fichero
- Raster gráfico
- Protocolo TCP/IP

### Construcción de abstracciones

Uno de los trabajos principales de un informático es la construcción de abstracciones que permitan ahorrar tiempo y esfuerzo a la hora de tratar con la complejidad del mundo real.

Cita de Joel Spolsky en su blog [Joel on Software](#)

TCP es lo que los científicos de computación llaman una abstracción: una simplificación de algo mucho más complicado que va por debajo de la cubierta. Resulta que una gran parte de la programación de computadores consiste en construir abstracciones. ¿Qué es una librería de cadenas? Es una forma de hacer creer que los computadores pueden manipular cadenas tan fácilmente como manipulan los números. ¿Qué es un sistema de ficheros? Es una forma de hacer creer que un disco duro no es en realidad un montón de platos magnéticos en rotación que pueden almacenar bits en ciertas posiciones, sino en su lugar, un sistema jerárquico de carpetas-dentro-de-carpetas que contiene ficheros individuales.

## Distintos aspectos de los lenguajes de programación

La programación es una disciplina compleja, que tiene que tener en cuenta múltiples aspectos de los lenguajes de programación y las API:

1. Programas como procesos *runtime* que se ejecutan en un computador. Tenemos que entender qué pasa cuando se crea un objeto, cuánto tiempo permanece en memoria, cuál es el ámbito de una variable, etc.

Herramientas: depuradores, analizadores de rendimiento.

2. Programas como declaraciones estáticas. Hay que considerar un programa desde el punto de vista de una declaración de nuevos tipos, nuevos métodos, tipos genéricos, herencia entre clases, etc.

Herramientas: entornos de programación con autocompletado de código, detección de errores sintácticos.

3. Programas como comunicación y actividad social. Tenemos que tener en cuenta que un programa va a ser usado por otras personas, leído, extendido, mantenido, modificado. Los programas siempre se van a modificar.

Herramientas: sistemas de control de versiones (Git, Mercurial, Github, Bitbucket), de gestión de incidencias (Jira) , tests que evitan errores de regresión, ...

## Paradigmas de programación

### ¿Qué es un paradigma de programación?

Un paradigma define un conjunto de características, patrones y estilos de programación basados en alguna idea fundamental. Por ejemplo el paradigma funcional se basa en la idea que una computación se puede especificar como un conjunto de funciones que transforman valores de entrada en valores de salida.

Es conveniente ver un paradigma como un estilo de programación que puede usarse en distintos lenguajes de programación y expresarse con distintas sintaxis. Por ejemplo, se puede escribir código que use programación lógica en Prolog (sería lo más natural), pero también en Java, usando algún API específica.

Normalmente todos los lenguajes tienen características de más de un paradigma. Por motivos prácticos los lenguajes más populares no se limitan de forma estricta o pura a un único paradigma de programación.

Por ejemplo, el Prolog es un lenguaje en su mayor parte lógico y declarativo, pero tiene operadores imperativos como el *corte*. A pesar de ello, es normal adscribir un lenguaje al paradigma en el que es más sencillo o natural escribir código usando sus construcciones.

Existen lenguajes que refuerzan y promueven la expresión de código en más de un paradigma de programación. Y lo hacen no por necesidad o accidente, sino con el intento explícito de fusionar más de un paradigma en una forma única de programar. Estos lenguajes se denominan lenguajes *multi-paradigma*.

Por ejemplo, Scala es lenguaje multi-paradigma en el que Martin Odersky, su creador, mezcla características de programación orientada a objetos con programación funcional.

Prolog o Lisp, aunque tienen características no lógicas o no funcionales, no pueden ser considerados multi-paradigma porque no fueron creados con la idea de integrar paradigmas variados en una forma coherente de expresión.

**Los paradigmas más importantes son:**

- **Paradigma funcional**
- **Paradigma lógico**
- **Paradigma imperativo o procedural**
- **Paradigma orientado a objetos**

## **Paradigma funcional**

Resumen de las características principales:

- **La computación se realiza mediante la evaluación de expresiones**
- **Definición de funciones**
- **Funciones como datos primitivos**
- Valores sin efectos laterales, **no existen referencias a celdas de memoria** en las que se guarda un estado modifiable
- **Programación declarativa** (en la programación funcional *pura*)

Lenguajes: **Lisp, Scheme, Haskell, Scala, Clojure.**

Ejemplo de código (Lisp):

```
(define (factorial x)
  (if (= x 0)
    1
    (* x (factorial (- x 1)))))

(factorial 8)
40320
(factorial 30)
265252859812191058636308480000000
```

## **Paradigma lógico**

Características:

- **Definición de reglas**
- **Unificación como elemento de computación**

- Programación declarativa

Lenguajes: Prolog, Mercury, Oz.

Ejemplo de código (Prolog):

```

padrede('juan', 'maria'). % juan es padre de maria
padrede('pablo', 'juan'). % pablo es padre de juan
padrede('pablo', 'marcela').
padrede('carlos', 'debora').

hijode(A,B) :- padrede(B,A).
abuelode(A,B) :- padrede(A,C), padrede(C,B).
hermanode(A,B) :- padrede(C,A) , padrede(C,B), A \== B.

familiarde(A,B) :- padrede(A,B).
familiarde(A,B) :- hijode(A,B).
familiarde(A,B) :- hermanode(A,B).

?- hermanode('juan', 'marcela').
yes
?- hermanode('carlos', 'juan').
no
?- abuelode('pablo', 'maria').
yes
?- abuelode('maria', 'pablo').
no

```

## Paradigma imperativo

Los lenguajes de programación que cumplen el paradigma imperativo se caracterizan por tener un estado implícito que es modificado mediante instrucciones o comandos del lenguaje. Como resultado, estos lenguajes tienen una noción de secuenciación de los comandos para permitir un control preciso y determinista del estado.

Características:

- Definición de procedimientos
- Definición de tipos de datos
- Chequeo de tipos en tiempo de compilación
- Cambio de estado de variables
- Pasos de ejecución de un proceso

Ejemplo (Pascal): <

```

type
  tDimension = 1..100;
  eMatriz(f,c: tDimension) = array [1..f,1..c] of real;

```

```
tRango = record
    f,c: tDimension value 1;
end;

tpMatriz = ^eMatriz;

procedure EscribirMatriz(var m: tpMatriz);
var filas,col : integer;
begin
    for filas := 1 to m^.f do begin
        for col := 1 to m^.c do
            write(m^[filas,col]:7:2);
        writeln(resultado);
        writeln(resultado)
    end;
end;
```

## Paradigma orientado a objetos

Características:

- Definición de clases y herencia
- Objetos como abstracción de datos y procedimientos
- Polimorfismo y chequeo de tipos en tiempo de ejecución

Ejemplo (Java):

```
public class Bicicleta {
    public int marcha;
    public int velocidad;

    public Bicicleta(int velocidadInicial, int marchaInicial) {
        marcha = marchaInicial;
        velocidad = velocidadInicial;
    }

    public void setMarcha(int nuevoValor) {
        marcha = nuevoValor;
    }

    public void frenar(int decremento) {
        velocidad -= decremento;
    }

    public void acelerar(int incremento) {
        velocidad += incremento;
    }
}
```

```
public class MountainBike extends Bicicleta {  
    public int alturaSillin;  
  
    public MountainBike(int alturaInicial,  
                        int velocidadInicial,  
                        int marchaInicial) {  
        super(velocidadInicial, marchaInicial);  
        alturaSillin = alturaInicial;  
    }  
  
    public void setAltura(int nuevoValor) {  
        alturaSillin = nuevoValor;  
    }  
}  
  
public class Excursion {  
    public static void main(String[] args) {  
        MountainBike miBicicleta = new MoutainBike(10,10,3);  
        miBicicleta.acelerar(10);  
        miBicicleta.setMarcha(4);  
        miBicicleta.frenar(10);  
    }  
}
```

## Compiladores e intérpretes

En el nivel de abstracción más bajo, la ejecución de un programa en un computador consiste en la ejecución de un conjunto de instrucciones del código máquina del procesador. Por ejemplo, la siguiente figura muestra un ejemplo de un programa en ensamblador para un antiguo procesador (el Z80, procesador de 8 bits del mítico [ZX Spectrum](#), uno de los primeros ordenadores personales en Europa):

```

; Exit:           Register H = High byte of element address
;                 Register L = Low byte of element address ;
; Registers used: AF,BC,DE,HL ;
; Time:           Approximately 1100 cycles ;
; Size:           Program 44 bytes
;                 Data    4 bytes ;
;

D2BYTE:
;SAVE RETURN ADDRESS
POP   HL
LD    (RETADR),HL

;GET SECOND SUBSCRIPT
POP   HL
LD    (SS2),HL
;GET SIZE OF FIRST SUBSCRIPT (ROW LENGTH), FIRST SUBSCRIPT
POP   DE          ;GET LENGTH OF ROW
POP   BC          ;GET FIRST SUBSCRIPT
;MULTIPLY FIRST SUBSCRIPT * ROW LENGTH USING SHIFT AND ADD
; ALGORITHM. PRODUCT IS IN HL
LD    HL,0          ;PRODUCT = 0
LD    A,15          ;COUNT = BIT LENGTH - 1
MLP:
SLA   E
RL    D          ;SHIFT LOW BYTE OF MULTIPLIER
JR    NC,MLP1      ;ROTATE HIGH BYTE OF MULTIPLIER
ADD   HL,BC        ;JUMP IF MSB OF MULTIPLIER = 0
MLP1: ADD   HL,HL      ;ADD MULTIPLICAND TO PARTIAL PRODUCT
DEC   A
JR    NZ,MLP      ;SHIFT PARTIAL PRODUCT
;CONTINUE THROUGH 15 BITS
;DO LAST ADD IF MSB OF MULTIPLIER IS 1
OR    D
JP    P,MLP2      ;SIGN FLAG = MSB OF MULTIPLIER
ADD   HL,BC        ;ADD IN MULTIPLICAND IF SIGN = 1
MLP2: ADD   DE,(SS2)  ;ADD IN SECOND SUBSCRIPT
ADD   HL,DE

;ADD BASE ADDRESS TO FORM FINAL ADDRESS
POP   DE          ;GET BASE ADDRESS OF ARRAY
ADD   HL,DE        ;ADD BASE TO INDEX
;RETURN TO CALLER
LD    DE,(RETADR)  ;RESTORE RETURN ADDRESS TO STACK
PUSH  DE
RET

;DATA

```

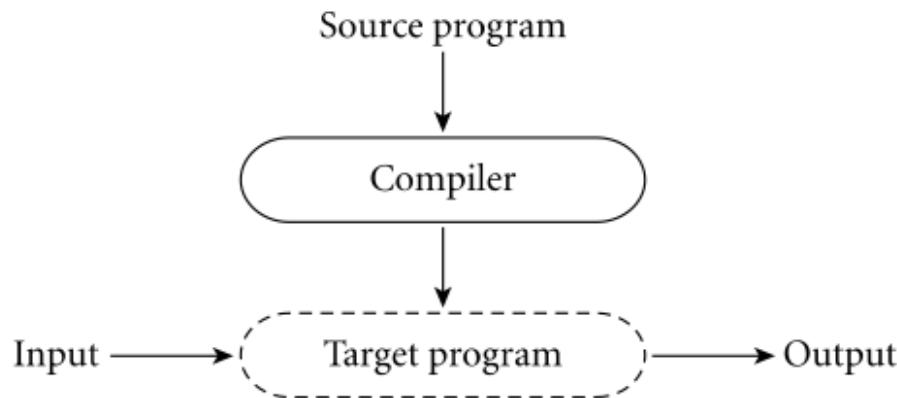
### Ensamblador Z80

Dependiendo del tipo de lenguaje de programación en el que esté escrito este programa, el código máquina que se estará ejecutando será:

- el resultado de la compilación del programa original (en el caso de un lenguaje compilado)
- el código de un programa (intérprete) que realiza la interpretación del programa original (en el caso de un lenguaje interpretado)

### Compilación

La siguiente figura (tomada, como las demás de este apartado del *Programming Language Pragmatics*) muestra el proceso de generación y ejecución de un programa compilado.

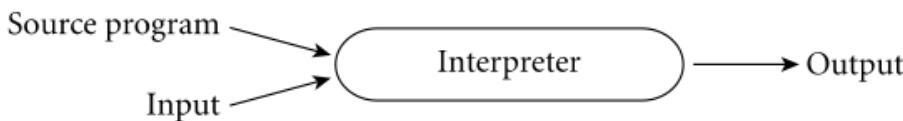


### Compilación

El proceso de compilación de un programa consiste en la traducción del código fuente original en el lenguaje de alto nivel al código máquina específico del procesador en el que va a ejecutarse el programa. El código máquina resultante sólo corre en el procesador para el que se ha generado. Por ejemplo, un programa C compilado para un procesador Intel no puede ejecutarse en un procesador ARM, como los [A7 de Apple](#).

- Ejemplos: C, C++
- Diferentes momentos en la vida de un programa: tiempo de compilación y tiempo de ejecución
- Mayor eficiencia

### Interpretación



### Interpretación

- Ejemplos: BASIC, Lisp, Scheme, Python, Ruby
- No hay diferencia entre el tiempo de compilación y el tiempo de ejecución
- Mayor flexibilidad: el código se puede construir y ejecutar "on the fly" (funciones lambda o closures)

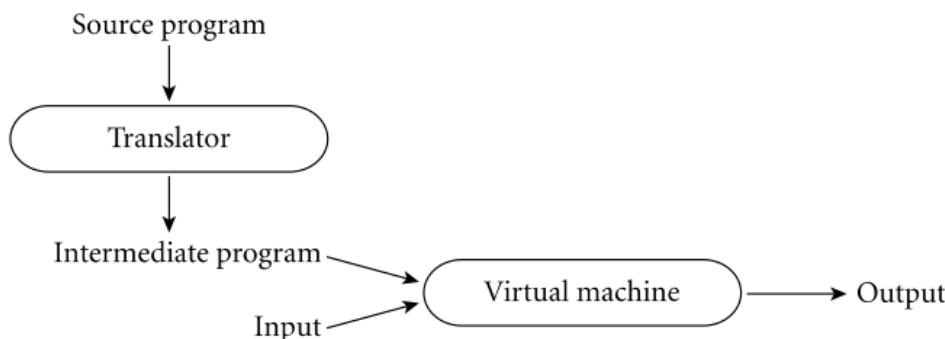
Los lenguajes interpretados suelen proporcionar un *shell* o intérprete. Se trata de un entorno interactivo en el que podemos definir y evaluar expresiones. Este entorno se denomina en los círculos de programación funcional un *REPL* (*Read, Eval, Print, Loop*) y ya se utilizó en los primeros años de implementación del Lisp. El uso del *REPL* promueve una programación interactiva en la que continuamente evaluamos y comprobamos el código que desarrollamos.

### Enfoques mixtos

Existen también enfoques mixtos, como el usado por el lenguaje de programación Java, en el que se realizan ambos procesos.

En una primera fase el compilador de Java (`javac`) realiza una traducción del código fuente original a un *código intermedio* binario independiente del procesador, denominado *bytecode*. Este código binario es multiplataforma.

El código intermedio es interpretado después por un intérprete (**java**) que ya sí que es dependiente de la plataforma. En la figura el intérprete se denomina *Virtual machine* (no confundir con el concepto de *máquina virtual* que permite emular un sistema operativo, por ejemplo VirtualBox).



### *Enfoque mixto (Java)*

- Ejemplos: Java, Scala

## Bibliografía

- Introducción y capítulo 1 del Structure and Interpretation of Computer Programs: *Building Abstractions with Procedures*
- Capítulo 1.2 Programming Language Pragmatics: *The Programming Language Spectrum*
- Capítulo 1.3 Programming Language Pragmatics, *Why Study Programming Languages*
- Capítulo 1.4 Programming Language Pragmatics, *Compilation and Interpretation*
- Raul Rojas, "Konrad Zuse's legacy the architecture of the Z1 and Z3", IEEE Annals of the History of Computing, Vol. 19, No. 2, 1997
- Charles Petzold, "Code", Microsoft Press, 2000 (Capítulo 18: "From Abaci to Chips")
- Jack Copeland, "The Modern History of Computing", The Stanford Encyclopedia of Philosophy (Fall 2008 Edition), URL = <http://plato.stanford.edu/archives/fall2008/entries/computing-history/>
- Georgi Dalakov, "History of Computers", URL = <http://history-computer.com>

Lenguajes y Paradigmas de Programación, curso 2019-20

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante  
Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez

## Tema 2: Programación funcional

### El paradigma de Programación Funcional

Pasado y presente del paradigma funcional

#### Definición y características

En una definición muy breve y concisa la programación funcional define un **programa** de la siguiente forma:

!!! Quote "Definición de programa funcional" En programación funcional un programa es un conjunto de funciones matemáticas que convierten unas entradas en unas salidas, sin ningún estado interno y ningún efecto lateral.

Hablaremos más adelante de la no existencia de estado interno (variables en las que se guardan y se modifican valores) y de la ausencia de efectos laterales. Avancemos que estas son también características de la **programación declarativa** (frente a la programación tradicional imperativa que es la que se utiliza en lenguajes como C o Java). En este sentido, la programación funcional es un tipo concreto de programación declarativa.

Las características principales del paradigma funcional son:

- Definiciones de funciones matemáticas puras, sin estado interno ni efectos laterales
- Valores inmutables
- Uso profuso de la recursión en la definición de las funciones
- Uso de listas como estructuras de datos fundamentales
- Funciones como tipos de datos primitivos: expresiones lambda y funciones de orden superior

Expicaremos estas propiedades a continuación.

## Orígenes históricos

En los años 30, junto con la máquina de Turing, se propusieron distintos modelos computacionales equivalentes que formalizaban el concepto de *algoritmo*. Uno de estos modelos fue el denominado *Cálculo lambda* propuesto por Alonzo Church en los años 30 y basado en la evaluación de expresiones matemáticas. En este formalismo los algoritmos se expresan mediante funciones matemáticas en las que puede ser usada la recursión. Una función matemática recibe parámetros de entrada y devuelve un valor. La evaluación de la función se realiza evaluando sus expresiones matemáticas mediante la sustitución de los parámetros formales por los valores reales que se utilizan en la invocación (el denominado **modelo de sustitución** que veremos más adelante).

El cálculo lambda es un formalismo matemático, basado en operaciones abstractas. Dos décadas después, cuando los primeros computadores electrónicos estaban empezando a utilizarse en grandes empresas y en universidades, este formalismo dio origen a algo mucho más tangible y práctico: un lenguaje de alto nivel, mucho más expresivo que el ensamblador, con el que expresar operaciones y funciones **que pueden ser definidas y evaluadas en el computador**, el lenguaje de programación Lisp.

## Historia y características del Lisp

- [Lisp](#) es el primer lenguaje de programación de alto nivel basado en el paradigma funcional.
- Creado en 1958 por John McCarthy.
- Lisp fue en su época un lenguaje revolucionario que introdujo nuevos conceptos de programación no existentes entonces: funciones como objetos primitivos, funciones de orden superior, polimorfismo, listas, recursión, símbolos, homogeneidad de datos y programas, bucle REPL (*Read-Eval-Print Loop*)
- La herencia del Lisp llega a lenguajes derivados de él (Scheme, Golden Common Lisp) y a nuevos lenguajes de paradigmas no estrictamente funcionales, como C#, Python, Ruby, Objective-C o Scala.

Lisp fue el primer lenguaje de programación interpretado, con muchas características dinámicas que se ejecutan en tiempo de ejecución (*run-time*). Entre estas características podemos destacar la gestión de la

memoria (creación y destrucción **automática** de memoria reservada para datos), la detección de excepciones y errores en tiempo de ejecución o la creación en tiempo de ejecución de funciones anónimas (expresiones *lambda*). Todas estas características se ejecutan mediante un *sistema de tiempo de ejecución (runtime system)* presente en la ejecución de los programas. A partir del Lisp muchos otros lenguajes han usado estas características de interpretación o de sistemas de tiempo de ejecución. Por ejemplo, lenguajes como BASIC, Python, Ruby o JavaScript son lenguajes interpretados. Y lenguajes como Java o C# tienen una avanzada plataforma de tiempo de ejecución con soporte para la gestión de la memoria dinámica (*recolección de basura, garbage collection*) o la [compilación just in time](#).

Lisp no es un lenguaje exclusivamente funcional. Lisp se diseñó con el objetivo de ser un lenguaje de alto nivel capaz de resolver problemas prácticos de Inteligencia Artificial, no con la idea de ser un lenguaje formal basado en un único modelo de computación. Por ello en Lisp (y en Scheme) existen primitivas que se salen del paradigma funcional puro y permiten programar de forma imperativa (no declarativa), usando mutación de estado y pasos de ejecución.

Sin embargo, durante la primera parte de la asignatura en la que estudiaremos la programación funcional, no utilizaremos las instrucciones imperativas de Scheme sino que escribiremos código exclusivamente funcional.

## Lenguajes de programación funcional

En los años 60 la programación funcional definida por el Lisp fue dominante en departamentos de investigación en Inteligencia Artificial (MIT por ejemplo). En los años 70, 80 y 90 se fue relegando cada vez más a los nichos académicos y de investigación; en la empresa se impusieron los lenguajes imperativos y orientados a objetos.

En la primera década del 2000 han aparecido lenguajes que evolucionan de Lisp y que resaltan sus aspectos funcionales, aunque actualizando su sintaxis. Destacamos entre ellos:

- [Clojure](#)
- [Erlang](#)

También hay una tendencia desde mediados de la década de 2000 de incluir aspectos funcionales como las expresiones *lambda* o las funciones de orden superior en lenguajes imperativos orientados a objetos, dando lugar a lenguajes *multi-paradigma*:

- [Ruby](#)
- [Python](#)
- [Groovy](#)
- [Scala](#)
- [Swift](#)

Por último, en la década del 2010 también se ha hecho popular un lenguaje **exclusivamente funcional** como [Haskell](#). Este lenguaje, a diferencia de Scheme y de otros lenguajes multi-paradigma, no tienen ningún elemento imperativo y consigue que todas sus expresiones sean puramente funcionales.

## Aplicaciones prácticas de la programación funcional

En la actualidad el paradigma funcional es un **paradigma de moda**, como se puede comprobar observando la cantidad de artículos, charlas y blogs en los que se habla de él, así como la cantidad de lenguajes que están

aplicando sus conceptos. Por ejemplo, solo como muestra, a continuación puedes encontrar algunos enlaces a charlas y artículos interesantes publicados recientemente sobre programación funcional:

- Lupo Montero - [Introducción a la programación funcional en JavaScript](#) (Blog)
- Andrés Marzal - [Por qué deberías aprender programación funcional ya mismo](#) (Charla en YouTube)
- Mary Rose Cook - [A practical introduction to functional programming](#) (Blog)
- Ben Christensen - [Functional Reactive Programming in the Netflix API](#) (Charla en InfoQ)

El auge reciente de estos lenguajes y del paradigma funcional se debe a varios factores, entre ellos que es un paradigma que facilita:

- la programación de sistemas concurrentes, con múltiples hilos de ejecución o con múltiples computadores ejecutando procesos conectados concurrentes.
- la definición y composición de múltiples operaciones sobre *streams* de forma muy concisa y compacta, aplicable a la programación de sistemas distribuidos en Internet.
- la programación interactiva y evolutiva.

### **Programación de sistemas concurrentes**

Veremos más adelante que una de las características principales de la programación funcional es que no se usa la *mutación* (no se modifican los valores asignados a variables ni parámetros). Esta propiedad lo hace un paradigma excelente para implementar programas concurrentes, en los que existen múltiples hilos de ejecución. La programación de sistemas concurrentes es muy complicada con el paradigma imperativo tradicional, en el que la modificación del estado de una variable compartida por más de un hilo puede provocar *condiciones de carrera* y errores difícilmente localizables y reproducibles.

Como dice [Bartosz Milewski](#), investigador y teórico de ciencia de computación, en su [respuesta en Quora](#) a la pregunta *¿por qué a los ingenieros de software les gusta la programación funcional?*:

!!! Quote "Bartosz Milewski: ¿Por qué es popular la programación funcional?" Porque es la única forma práctica de escribir programas concurrentes. Intentar escribir programas concurrentes en lenguajes imperativos, no sólo es difícil, sino que lleva a *bugs* que son muy difíciles de descubrir, reproducir y arreglar. En los lenguajes imperativos y, en particular, en los lenguajes orientados a objetos se ocultan las mutaciones y se comparten datos sin darse cuenta, por lo que son extremadamente propensos a los errores de concurrencia producidos por las condiciones de carrera.

### **Definición y composición de operaciones sobre streams**

El paradigma funcional ha originado un estilo de programación sobre *streams* de datos, en el que se concatenan operaciones como *filter* o *map* para definir de forma sencilla procesos y transformaciones asíncronas aplicables a los elementos del *stream*. Este estilo de programación ha hecho posible nuevas ideas de programación, como la programación *reactiva*, basada en eventos, o los *futuros* o *promesas* muy utilizados en lenguajes muy populares como JavaScript para realizar peticiones asíncronas a servicios web.

Por ejemplo, en el artículo [Exploring the virtues of microservices with Play and Akka](#) se explica con detalle las ventajas del uso de lenguajes y primitivas para trabajar con sistemas asíncronos basados en eventos en servicios como Tumblr o Netflix.

Otro ejemplo es el [uso de Scala en Tumblr](#) con el que se consigue crear código que no tiene estado compartido y que es fácilmente paralelizable entre los más de 800 servidores necesarios para atender picos de más de 40.000 peticiones por segundo:

!!! Quote "Uso de Scala en Tumblr" Scala promueve que no haya estado compartido. El estado mutable se evita usando sentencias en Scala. No se usan máquinas de estado de larga duración. El estado se saca de la base de datos, se usa, y se escribe de nuevo en la base de datos. La ventaja principal es que los desarrolladores no tienen que preocuparse sobre hilos o bloqueos.

### Programación evolutiva

En la metodología de programación denominada *programación evolutiva* o *iterativa* los programas complejos se construyen a base de ir definiendo y probando elementos computacionales cada vez más complicados. Los lenguajes de programación funcional encajan perfectamente en esta forma de construir programas.

Como Abelson y Sussman comentan en el libro *Structure and Implementation of Computer Programs* (SICP):

!!! Quote "Abelson y Sussman sobre la programación incremental" En general, los objetos computacionales pueden tener estructuras muy complejas, y sería extremadamente inconveniente tener que recordar y repetir sus detalles cada vez que queremos usarlas. En lugar de ello, se construyen programas complejos componiendo, paso a paso, objetos computacionales de creciente complejidad.

El intérprete hace esta construcción paso-a-paso de los programas particularmente conveniente porque las asociaciones nombre-objeto se pueden crear de forma incremental en interacciones sucesivas. Esta característica favorece el desarrollo y prueba incremental de programas, y es en gran medida responsable del hecho de que un programa Lisp consiste normalmente de un gran número de procedimientos relativamente simples.

No hay que confundir una metodología de programación con un paradigma de programación. Una metodología de programación proporciona sugerencias sobre cómo debemos diseñar, desarrollar y mantener una aplicación que va a ser usada por usuarios finales. La programación funcional se puede usar con múltiples metodologías de programación, debido a que los programas resultantes son muy claros, expresivos y fáciles de probar.

### Evaluación de expresiones y definición de funciones

En la asignatura usaremos Scheme como primer lenguaje en el que exploraremos la programación funcional.

En el seminario de Scheme que se imparte en prácticas se estudiará en más profundidad los conceptos más importantes del lenguaje: tipos de datos, operadores, estructuras de control, intérprete, etc.

### Evaluación de expresiones

Empezamos este apartado viendo cómo se definen y evalúan expresiones Scheme. Y después veremos cómo construir nuevas funciones.

Scheme es un lenguaje que viene del Lisp. Una de sus características principales es que las expresiones se construyen utilizando paréntesis.

Ejemplos de expresiones en Scheme, junto con el resultado de su ejecución:

```
2 ; => 2
(+ 2 3) ; => 5
(+) ; => 0
(+ 2 4 5 6) ; => 17
(+ (* 2 3) (- 3 1)) ; => 8
```

En programación funcional en lugar de decir "ejecutar una expresión" se dice "**evaluar una expresión**", para reforzar la idea de que se tratan de expresiones matemáticas que **siempre devuelven uno y sólo un resultado**.

Las expresiones se definen con una notación prefija: el primer elemento después del paréntesis de apertura es el **operador** de la expresión y el resto de elementos (hasta el paréntesis de cierre) son sus operandos.

- Por ejemplo, en la expresión **(+ 2 4 5 6)** el operador es el símbolo **+** que representa función *suma* y los operandos son los números 2, 4, 5 y 6.
- Puede haber expresiones que no tengan operandos, como el ejemplo **(+)**, cuya evaluación devuelve 0.

Una idea fundamental de Lisp y Scheme es que los paréntesis se evalúan de dentro a fuera. Por ejemplo, la expresión

```
(+ (* 2 3) (- 3 (/ 12 3)))
```

que devuelve 5, se evalúa así:

```
(+ (* 2 3) (- 3 (/ 12 3))) =>
(+ 6 (- 3 (/ 12 3))) =>
(+ 6 (- 3 4)) =>
(+ 6 -1) =>
5
```

La evaluación de cada expresión devuelve un valor que se utiliza para continuar calculando la expresión exterior. En el caso anterior

- primero se evalúa la expresión **(\* 2 3)** que devuelve 6,
- después se evalúa **(/ 12 3)** que devuelve 4,
- después se evalúa **(- 3 4)** que devuelve -1
- y por último se evalúa **(+ 6 -1)** que devuelve 5

Cuando se evalúa una expresión en el intérprete de Scheme el resultado aparece en la siguiente línea.

## Definición de funciones

En programación funcional las funciones son similares a las funciones matemáticas: reciben parámetros y devuelven siempre un único resultado de operar con esos parámetros.

Por ejemplo, podemos definir la función (**cuadrado x**) que devuelve el cuadrado de un número que pasamos como parámetro:

```
(define (cuadrado x)
  (* x x))
```

Después del nombre de la función se declaran sus argumentos. El número de argumentos de una función se denomina **aridad de la función**. Por ejemplo, la función **cuadrado** es una función de aridad 1, o *unaria*.

Después de declarar los parámetros, se define el cuerpo de la función. Es una expresión que se evaluará con el valor que se pase como parámetro. En el caso anterior la expresión es (**\* x x**) y multiplicará el parámetro por si mismo.

Hay que hacer notar que en Scheme no existe la palabra clave **return**, sino que las funciones siempre se definen con una única expresión cuya evaluación es el resultado que se devuelve.

Una vez definida la función **cuadrado** podemos usarla de la misma forma que las funciones primitivas de Scheme:

```
(cuadrado 10) ; => 100
(cuadrado (+ 10 (cuadrado (+ 2 4)))) ; => 2116
```

La evaluación de la última expresión se hace de la siguiente forma:

```
(cuadrado (+ 10 (cuadrado (+ 2 4)))) =>
(cuadrado (+ 10 (cuadrado 6))) =>
(cuadrado (+ 10 36)) =>
(cuadrado 46) =>
2116
```

## Definición de funciones auxiliares

Las funciones definidas se pueden utilizar a su vez para construir otras funciones.

Lo habitual en programación funcional es definir funciones muy pequeñas e ir construyendo funciones cada vez de mayor nivel usando las anteriores.

### Ejemplo: suma de cuadrados

Por ejemplo, supongamos que tenemos que definir una función que devuelva la suma del cuadrado de dos números. Podríamos definirla escribiendo la expresión completa, pero queda una definición poco legible.

```
; Definición poco legible de la suma de cuadrados

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

Podemos hacer una definición mucho más legible si usamos la función **cuadrado** definida anteriormente:

```
; Definición de suma de cuadrados más legible.
; Usamos la función auxiliar 'cuadrado'

(define (cuadrado x)
  (* x x))

(define (suma-cuadrados x y)
  (+ (cuadrado x)
     (cuadrado y)))
```

Esta segunda definición es mucho más expresiva. Leyendo el código queda muy claro qué es lo que queremos hacer.

#### Ejemplo: tiempo de impacto

Veamos otro ejemplo de uso de funciones auxiliares. Supongamos que estamos programando un juego de guerra de barcos y submarinos, en el que utilizamos las coordenadas del plano para situar todos los elementos de nuestra flota.

Supongamos que necesitamos calcular el tiempo que tarda un torpedo en llegar desde una posición (**x1**, **y1**) a otra (**x2**, **y2**). Suponemos que la velocidad del torpedo es otro parámetro **v**.

¿Cómo calcularíamos este tiempo de impacto?

La forma menos correcta de hacerlo es definir todo el cálculo en una única expresión. Como en programación funcional las funciones deben definirse con una única expresión debemos realizar todo el cálculo en forma de expresiones anidadas, unas dentro de otras. Esto construye una función que calcula bien el resultado. El problema que tiene es que es muy difícil de leer y entender para un compañero (o para nosotros mismos, cuando pasen unos meses):

```
;;
; Definición incorrecta: muy poco legible
;
; La función tiempo-impacto devuelve el tiempo que tarda
; en llegar un torpedo a la velocidad v desde la posición
; (x1, y1) a la posición (x2, y2)
```

```
;  
  
(define (tiempo-impacto x1 y1 x2 y2 v)  
  (/ (sqrt (+ (* (- x2 x1) (- x2 x1))  
             (* (- y2 y1) (- y2 y1))))  
      v))
```

La función anterior hace bien el cálculo pero es muy complicada de modificar y de entender.

La forma más correcta de definir la función sería usando varias funciones auxiliares. Fíjate que es muy importante también poner los nombres correctos a cada función, para entender qué hace. Scheme es un lenguaje débilmente tipado y no tenemos la ayuda de los tipos que nos dan más contexto de qué es cada parámetro y qué devuelve la función.

```
; Definición correcta, modular y legible de la función tiempo-impacto  
  
;  
; La función 'cuadrado' devuelve el cuadrado de un número  
;  
  
(define (cuadrado x)  
  (* x x))  
  
;  
; La función 'distancia' devuelve la distancia entre dos  
; coordenadas (x1, y1) y (x2, y2)  
;  
  
(define (distancia x1 y1 x2 y2)  
  (sqrt (+ (cuadrado (- x2 x1))  
            (cuadrado (- y2 y1)))))  
  
;  
; La función 'tiempo' devuelve el tiempo que  
; tarda en recorrer un móvil una distancia d a un velocidad v  
;  
  
(define (tiempo distancia velocidad)  
  (/ distancia velocidad))  
  
;  
; La función 'tiempo-impacto' devuelve el tiempo que tarda  
; en llegar un torpedo a la velocidad v desde la posición  
; (x1, y1) a la posición (x2, y2)  
;  
  
(define (tiempo-impacto x1 y1 x2 y2 velocidad)  
  (tiempo (distancia x1 y1 x2 y2) velocidad))
```

En esta segunda versión definimos más funciones, pero cada una es mucho más legible. Además las funciones como **cuadrado**, **distancia** o **tiempo** las vamos a poder reutilizar para otros cálculos.

## Funciones puras

A diferencia de lo que hemos visto en programación imperativa, en programación funcional no es posible definir funciones con estado local. Las funciones que se definen son funciones matemáticas puras, que cumplen las siguientes condiciones:

- No modifican los parámetros que se les pasa
- Devuelven un único resultado
- No tienen estado local ni el resultado depende de un estado exterior mutable

Esta última propiedad es muy importante y quiere decir que la función siempre devuelve el mismo valor cuando se le pasan los mismos parámetros.

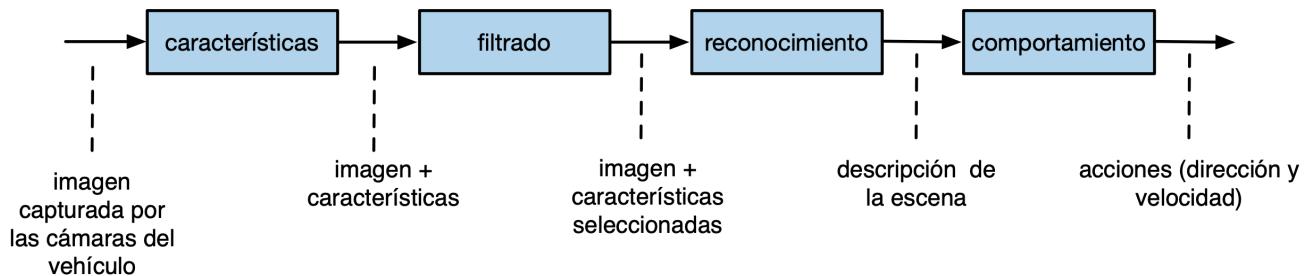
Las funciones puras son muy fáciles de entender porque no es necesario tener en cuenta ningún contexto a la hora de describir su funcionamiento. El valor devuelto únicamente depende de los parámetros de entrada.

Por ejemplo, funciones matemáticas como suma, resta, cuadrado, sin, cos, etc. cumplen esta propiedad.

## Composición de funciones

Una idea fundamental de la programación funcional es la composición de funciones que transforman unos datos de entrada en otros de salida. Es una idea muy actual, porque es la forma en la que están planteados muchos algoritmos de procesamiento de datos en inteligencia artificial.

Por ejemplo, podemos representar de la siguiente forma el algoritmo que maneja un vehículo autónomo:



Las cajas representan funciones que transforman los datos de entrada (imágenes tomadas por las cámaras del vehículo) en los datos de salida (acciones a realizar sobre la dirección y el motor del vehículo). Las funciones intermedias representan transformaciones que se realizan sobre los datos de entrada y obtienen los datos de salida.

En un lenguaje de programación funcional como Scheme el diagrama anterior se escribiría con el siguiente código:

```
(define (conduce-vehiculo imágenes)
  (obten-acciones
    (reconoce
      (filtra
        (obten-características imágenes)))))
```

Veremos más adelante que las expresiones en Scheme se evalúan de dentro a fuera y que tienen notación prefija. El resultado de cada función constituye la entrada de la siguiente.

En el caso de la función **conduce-vehiculo** primero se obtienen las características de las imágenes, después se filtran, después se reconoce la escena y, por último, se obtienen las acciones para conducir el vehículo.

## Programación declarativa vs. imperativa

Hemos dicho que la programación funcional es un estilo de programación declarativa, frente a la programación tradicional de los lenguajes denominados imperativos. Vamos a explicar esto un poco más.

### Programación declarativa

Empecemos con lo que conocemos todos: un **programa imperativo**. Se trata de un conjunto de instrucciones que se ejecutan una tras otra (pasos de ejecución) de forma secuencial. En la ejecución de estas instrucciones se van cambiando los valores de las variables y, dependiendo de estos valores, se modifica el flujo de control de la ejecución del programa.

Para entender el funcionamiento de un programa imperativo debemos imaginar toda la evolución del programa, los pasos que se ejecutan y cuál es el flujo de control en función de los cambios de los valores en las variables.

En la **programación declarativa**, sin embargo, utilizamos un paradigma totalmente distinto. Hablamos de *programación declarativa* para referirnos a lenguajes de programación (o sentencias de código) en los que se *declaran* los valores, objetivos o características de los elementos del programa y en cuya ejecución no existe mutación (modificación de valores de variables) ni secuencias de pasos de ejecución.

De esta forma, la ejecución de un programa declarativo tiene que ver más con algún modelo formal o matemático que con un programa tradicional imperativo. Define un conjunto de reglas y definiciones *de estilo matemático*.

La programación declarativa no es exclusiva de los lenguajes funcionales. Existen muchos lenguajes no funcionales con características declarativas. Por ejemplo Prolog, en el que un programa se define como un conjunto de reglas lógicas y su ejecución realiza una deducción lógica matemática que devuelve un resultado. En dicha ejecución no son relevantes los pasos internos que realiza el sistema sino las relaciones lógicas entre los datos y los resultados finales.

Un ejemplo claro de programación declarativa es una **hoja de cálculo**. Las celdas contiene valores o expresiones matemáticas que se actualizan automáticamente cuando cambiamos los valores de entrada. La relación entre valores y resultados es totalmente matemática y para su cálculo no tenemos que tener en cuenta pasos de ejecución. Evidente, por debajo de la hoja de cálculo existe un programa que realiza el su cálculo de la hoja, pero cuando estamos usándola no nos preocupa esa implementación. Podemos no preocuparnos de ella y usar únicamente el modelo matemático definido en la hoja.

Otro ejemplo muy actual de programación declarativa es SwiftUI, el nuevo API creado por Apple para definir las interfaces de usuario de las aplicaciones iOS.

| Thumbnail | Name                | Favorite |
|-----------|---------------------|----------|
|           | Turtle Rock         | ★ >      |
|           | Silver Salmon Creek | >        |
|           | Chilkoot Trail      | ★ >      |
|           | St. Mary Lake       | ★ >      |
|           | Twin Lake           | >        |
|           | Lake McDonald       | >        |
|           | Charley Rivers      | >        |
|           | Icy Bay             | >        |
|           | Rainbow Lake        | >        |
|           | Hidden Lake         | >        |
|           | Chincoteague        | >        |

```

List(landmarks) { landmark in
    HStack {
        Image(landmark.thumbnail)
        Text(landmark.name)
        Spacer()

        if landmark.isFavorite {
            Image(systemName: "star.fill")
                .foregroundColor(.yellow)
        }
    }
}

```

En el código de la imagen vemos una descripción de cómo está definida la aplicación: una lista de lugares (*landmarks*) apilada verticalmente. Para cada lugar se define su imagen, su texto, y una estrella si el lugar es favorito.

El código es declarativo porque no hay pasos de ejecución para definir la interfaz. No existe un bucle que va añadiendo elementos a la interfaz. Vemos una declaración de cómo la interfaz va estar definida. El compilador del lenguaje y el API son los responsables de construir esa declaración y mostrar la interfaz tal y como nosotros queremos.

### Declaración de funciones

La programación funcional utiliza un estilo de programación declarativo. Declaramos funciones en las que se transforman unos datos de entrada en unos datos de salida. Veremos que esta transformación se realiza mediante la evaluación de expresiones, sin definir valores intermedios, ni variables auxiliares, ni pasos de ejecución. Únicamente se van componiendo llamadas a funciones auxiliares que construyen el valor resultante.

Tal y como ya hemos visto, el siguiente ejemplo es una **declaración** en Scheme de una función que toma como entrada un número y devuelve su cuadrado:

```
(define (cuadrado x)
  (* x x))
```

En el cuerpo de la función **cuadrado** vemos que no se utiliza ninguna variable auxiliar, sino que únicamente se llama a la función **\*** (multiplicación) pasando el valor de **x**. El valor resultante es el que se devuelve.

Por ejemplo, si llamamos a la función pasándole el parámetro 4 devuelve el resultado de multiplicar 4 por si mismo, 16.

```
(cuadrado 4) ; => 16
```

## Programación imperativa

Repasemos un algunas características propias de la programación imperativa **que no existen en la programación funcional**. Son características a las que estamos muy habituados porque son propias de los lenguajes más populares y con los que hemos aprendido a programar (C, C++, Java, python, etc.)

- Pasos de ejecución
- Mutación
- Efectos laterales
- Estado local mutable en las funciones

Veremos que, aunque parece imposible, es posible programar sin utilizar estas características. Lo demuestran lenguajes de programación funcional como Haskell, Clojure o el propio Scheme.

### Pasos de ejecución

Una de las características básicas de la programación imperativa es la utilización de pasos de ejecución. Por ejemplo, en C podemos realizar los siguientes pasos de ejecución:

```
int a = cuadrado(8);
int b = doble(a);
int c = cuadrado(b);
return c
```

O, por ejemplo, si queremos filtrar y procesar una lista de pedidos en Swift podemos hacerlo en dos sentencias:

```
filtrados = filtra(pedidos);
procesados = procesa(filtrados);
return procesados;
```

Sin embargo, en programación funcional (por ejemplo, Scheme) no existen pasos de ejecución separados por sentencias. Como hemos visto antes, la forma típica de expresar las instrucciones anteriores es componer todas las operaciones en una única instrucción:

```
(cuadrado (doble (cuadrado 8))) ; => 16384
```

El segundo ejemplo lo podemos componer de la misma forma:

```
(procesa (filtra pedidos))
```

## Mutación

En los lenguajes imperativos es común modificar el valor de las variables en los pasos de ejecución:

```
int x = 10;  
int x = x + 1;
```

La expresión `x = x + 1` es una expresión de **asignación** que modifica el valor anterior de una variable por un nuevo valor. El *estado* de las variables (su valor) cambia con la ejecución de los pasos del programa.

A esta asignación que modifica un valor ya existente se le denomina *asignación destructiva* o **mutación**.

En programación imperativa también se puede modificar (mutar) el valor de componentes de estructuras de datos, como posiciones de un array, de una lista o de un diccionario.

En programación funcional, por contra, **las definiciones son inmutables**, y una vez asignado un valor a un identificador no se puede modificar éste. En programación funcional **no existe sentencia de asignación** que pueda modificar un valor ya definido. Se entienden las variables como variables matemáticas, no como referencias a una posiciones de memoria que puede ser modificada.

Por ejemplo, la forma especial `define` en Scheme crea un nuevo identificador y le da el valor definido de forma permanente. Si escribimos el siguiente código en un programa en Scheme R6RS:

```
#lang racket  
  
(define a 12)  
(define a 200)
```

tendremos el siguiente error:

```
module: identifier already defined in: a
```

!!! Note "Nota" En el intérprete REPL del DrRacket sí que podemos definir más de una vez la misma función o identificador. Se ha diseñado así para facilitar el uso del intérprete para la prueba de expresiones en Scheme.

En los lenguajes de programación imperativos es habitual introducir también sentencias declarativas. Por ejemplo, en el siguiente código Java las líneas 1 y 3 las podríamos considerar declarativas y las 2 y 4 imperativas:

```

1. int x = 1;
2. x = x+1;
3. int y = x+1;
4. y = x;

```

## Mutación y efectos laterales

En programación imperativa es habitual también trabajar con referencias y hacer que más de un identificador refiera el mismo valor. Esto produce la posibilidad de que la mutación del valor a través de uno de los identificadores produzca un **efecto lateral** (*side effect* en inglés) en el que el valor de un identificador cambia sin ejecutar ninguna expresión en la que se utilice explícitamente el propio identificador.

Por ejemplo, en la mayoría de lenguajes orientados a objetos los identificadores guardan referencias a objetos. De forma que si asignamos un objeto a más de un identificador, todos los identificadores están accediendo al mismo objeto. Si mutamos algún valor del objeto a través de un identificador provocamos un efecto lateral en los otros identificadores.

Por ejemplo, lo siguiente es un ejemplo de una mutación en programación imperativa, en la que se modifican los atributos de un objeto en Java:

```

Point2D p1 = new Point2D(3.0, 2.0); // creamos un punto 2D con coordX=3.0 y
coordY=2.0
p1.getCoordX(); // la coord x de p2 es 3.0
p1.setCoordX(10.0);
p1.getCoordX(); // la coord x de p1 es 10.0

```

Si el objeto está asignado a más de una variable tendremos el **efecto lateral** (*side effect*) en el que el dato guardado en una variable cambia después de una sentencia en la que no se ha usado esa variable:

```

Point2D p1 = new Point2D(3.0, 2.0); // la coord x de p1 es 3.0
p1.getCoordX(); // la coord x de p1 es 3.0
Point2D p2 = p1;
p2.setCoordX(10.0);
p1.getCoordX(); // la coord x de p1 es 10.0, sin que ninguna sentencia haya
modificado directamente p1

```

El mismo ejemplo anterior, en C:

```

typedef struct {
    float x;
    float y;
} TPunto;

TPunto p1 = {3.0, 2.0};
printf("Coordenada x: %f", p1.x); // 3.0

```

```
TPunto *p2 = &p1;
p2->x = 10.0;
printf("Coordenada x: %f", p1.x); // 10.0 Efecto lateral
```

Los efectos laterales son los responsables de muchos *bugs* y hay que ser muy consciente de su uso. Son especialmente complicados de depurar los *bugs* debidos a efectos laterales en programas concurrentes con múltiples hilos de ejecución, en los que varios hilos pueden acceder a las mismas referencias y [provocar condiciones de carrera](#).

Por otro lado, también existen situaciones en las que su utilización permite ganar mucha eficiencia porque podemos definir estructuras de datos en el que los valores son compartidos por varias referencias y modificando un único valor se actualizan de forma instantánea esas referencias.

En los lenguajes en los que no existe la mutación no se producen efectos laterales, ya que no es posible modificar el valor de una variable una vez establecido. Los programas que escribamos en estos lenguajes van a estar libres de este tipo de *bugs* y van a poder ser ejecutado sin problemas en hilos de ejecución concurrente.

Por otro lado, la ausencia de mutación hace que sean algo más costosas ciertas operaciones, como la construcción de estructuras de datos nuevas a partir de estructuras ya existentes. Veremos, por ejemplo, que la única forma de añadir un elemento al final de una lista será construir una lista nueva con todos los elementos de la lista original y el nuevo elemento. Esta operación tiene un coste lineal con el número de elementos de la lista. Sin embargo, en una lista en la que pudiéramos utilizar la mutación podríamos implementar esta operación con coste constante.

### Estado local mutable

Otra característica de la programación imperativa es lo que se denomina **estado local mutable** en funciones, procedimientos o métodos. Se trata la posibilidad de que una invocación a un método o una función modifique un cierto estado, de forma que la siguiente invocación devuelva un valor distinto. Es una característica básica de la programación orientada a objetos, donde los objetos guardan valores que se modifican con la invocaciones a sus métodos.

Por ejemplo, en Java, podemos definir un contador que incrementa su valor:

```
public class Contador {
    int c;

    public Contador(int valorInicial) {
        c = valorInicial;
    }

    public int valor() {
        c++;
        return c;
    }
}
```

Cada llamada al método `valor()` devolverá un valor distinto:

```
Contador cont = new Contador(10);
cont.valor(); // 11
cont.valor(); // 12
cont.valor(); // 13
```

También se pueden definir funciones con estado local mutable en C:

```
int function contador () {
    static int c = 0;

    c++;
    return c;
}
```

Cada llamada a la función `contador()` devolverá un valor distinto:

```
contador() ;; 1
contador() ;; 2
contador() ;; 3
```

Por el contrario, los lenguajes funcionales tienen la propiedad de **transparencia referencial**: es posible sustituir cualquier aparición de una expresión por su resultado sin que cambia el resultado final del programa. Dicho de otra forma, en programación funcional, **una función siempre devuelve el mismo valor cuando se le llama con los mismos parámetros**. Las funciones no modifican ningún estado, no acceden a ninguna variable ni objeto global y modifican su valor.

## Resumen

Un resumen de las características fundamentales de la programación declarativa frente a la programación imperativa. En los siguientes apartados explicaremos más estas características.

### Características de la programación declarativa

- Variable = nombre dado a un valor (declaración)
- En lugar de pasos de ejecución se utiliza la composición de funciones
- No existe asignación ni cambio de estado
- No existe mutación, se cumple la *transferencia referencial*: dentro de un mismo ámbito todas las ocurrencias de una variable y las llamadas a funciones devuelven el mismo valor

### Características de la programación imperativa

- Variable = nombre de una zona de memoria
- Asignación

- Referencias
- Pasos de ejecución

## Modelo de computación de sustitución

Un modelo computacional es un formalismo (conjunto de reglas) que definen el funcionamiento de un programa. En el caso de los lenguajes funcionales basados en la evaluación de expresiones, el modelo computacional define cuál será el resultado de evaluar una expresión.

El **modelo de sustitución** es un modelo muy sencillo que permite definir la semántica de la evaluación de expresiones en lenguajes funcionales como Scheme. Se basa en una versión simplificada de la regla de reducción del cálculo lambda.

Es un modelo basado en la reescritura de unos términos por otros. Aunque se trata de un modelo abstracto, sería posible escribir un intérprete que, basándose en este modelo, evalúe expresiones funcionales.

Supongamos un conjunto de definiciones en Scheme:

```
(define (doble x)
  (+ x x))

(define (cuadrado y)
  (* y y))

(define (f z)
  (+ (cuadrado (doble z)) 1))

(define a 2)
```

Supongamos que, una vez realizadas esas definiciones, se evalúa la siguiente expresión:

```
(f (+ a 1))
```

¿Cuál será su resultado? Si lo hacemos de forma intuitiva podemos pensar que 37. Si lo comprobamos en el intérprete de Scheme veremos que devuelve 37. ¿Hemos seguido algunas reglas específicas? ¿Qué reglas son las que sigue el intérprete? ¿Podríamos implementar nosotros un intérprete similar? Sí, usando las reglas del modelo de sustitución.

El modelo de sustitución define cuatro reglas sencillas para evaluar una expresión. Llamemos a la expresión *e*. Las reglas son las siguientes:

1. Si *e* es un valor primitivo (por ejemplo, un número), devolvemos ese mismo valor.
2. Si *e* es un identificador, devolvemos su valor asociado con un **define** (se lanzará un error si no existe ese valor).
3. Si *e* es una expresión del tipo *(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.

La regla 4 tiene dos variantes, dependiendo del orden de evaluación que utilizamos.

### Orden aplicativo

4. Si  $e$  es una expresión del tipo  $(f \ arg_1 \dots \ arg_n)$ , donde  $f$  es el nombre de una función definida con un **define**, tenemos que evaluar primero los argumentos  $\arg_1 \dots \arg_n$  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 del tipo  $(f \ arg_1 \dots \ arg_n)$ , donde  $f$  es el nombre de una función definida con un **define**, tenemos que **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.

En el orden aplicativo se realizan las evaluaciones antes de realizar las sustituciones, lo que define una evaluación de *dentro a fuera* de los paréntesis. Cuando se llega a una expresión primitiva se evalúa.

En el orden normal se realizan todas las sustituciones hasta que se tiene una larga expresión formada por expresiones primitivas; se evalúa entonces.

Ambas formas de evaluación darán el mismo resultado en programación funcional. Scheme utiliza el orden aplicativo.

### Ejemplo 1

Vamos a empezar con un ejemplo sencillo para comprobar cómo se evalúa una misma expresión utilizando ambos modelos de sustitución. Supongamos las siguientes definiciones:

```
(define (doble x)
  (+ x x))

(define (cuadrado y)
  (* y y))

(define a 2)
```

Queremos evaluar la siguiente expresión:

```
(doble (cuadrado a))
```

La evaluación utilizando el **modelo de sustitución aplicativo**, usando paso a paso las reglas anteriores, es la siguiente (en cada línea se indica entre paréntesis la regla usada):

|                         |   |
|-------------------------|---|
| (doble (cuadrado a)) => | ; Sustituimos $a$ por su valor (R2)       |
| (doble (cuadrado 2)) => | ; Sustituimos cuadrado por su cuerpo (R4) |

```
(doble (* 2 2)) => ; Evaluamos (* 2 2) (R3)
(doble 4) => ; Sustituimos doble por su cuerpo (R4)
(+ 4 4) => ; Evaluamos (+ 4 4) (R3)
8
```

Podemos comprobar que en el modelo aplicativo se intercalan las sustituciones de una función por su cuerpo (regla 4) y las evaluaciones de expresiones (regla 3).

Por el contrario, la evaluación usando el **modelo de sustitución normal** es:

```
(doble (cuadrado a)) => ; Sustituimos doble por su cuerpo (R4)
(+ (cuadrado a) (cuadrado a)) => ; Sustituimos cuadrado por su cuerpo (R4)
(+ (* a a) (* a a)) => ; Sustituimos a por su valor (R2)
(+ (* 2 2) (* 2 2)) => ; Evaluamos (* 2 2) (R3)
(+ 4 (* 2 2)) => ; Evaluamos (* 2 2) (R3)
(+ 4 4) => ; Evaluamos (+ 4 4) (R3)
8
```

Al usar este modelo de evaluación primero se realizan todas las sustituciones (regla 4) y después todas las evaluaciones (regla 3).

Las sustituciones se hacen de izquierda a derecha (de fuera a dentro de los paréntesis). Primero se sustituye **doble** por su cuerpo y después **cuadrado**.

## Ejemplo 2

Veamos la evaluación del ejemplo algo más complicado que hemos planteado al comienzo:

```
(define (doble x)
  (+ x x))

(define (cuadrado y)
  (* y y))

(define (f z)
  (+ (cuadrado (doble z)) 1))

(define a 2)
```

Expresión a evaluar:

```
(f (+ a 1))
```

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

```
(f (+ a 1)) ⇒ ; Para evaluar f, evaluamos primero su argumento (+  
a 1) (R4)  
; 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) ⇒ ; Sustituyos (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 programación funcional el resultado de evaluar una expresión es el mismo independientemente del tipo de orden. Pero si estamos fuera del paradigma funcional y las funciones tienen estado y cambian de valor entre distintas invocaciones sí que importan si escogemos un orden.

Por ejemplo, supongamos una función (`random x`) que devuelve un entero aleatorio entre 0 y x. Esta función no cumpliría el paradigma funcional, porque devuelve un valor distinto con el mismo parámetro de entrada.

Evaluamos las siguientes expresiones con orden aplicativo y normal, para comprobar que el resultado es distinto.

```
(define (zero x) (- x x))  
(zero (random 10))
```

Si evaluamos la última expresión en orden aplicativo:

```
(zero (random 10)) ⇒ ; Evaluamos (random 10) (R3)  
(zero 3) ⇒ ; Sustituimos (zero ...) (R4)
```

```
(- 3 3) ⇒ ; Evaluamos - (R3)
0
```

Si lo evaluamos en orden normal:

```
(zero (random 10)) ⇒ ; Sustituimos (zero ...) (R4)
(- (random 10) (random 10)) ⇒ ; Evaluamos (random 10) (R3)
(- 5 3) ⇒ ; Evaluamos - (R3)
2
```

## Scheme como lenguaje de programación funcional

Ya hemos visto cómo definir funciones y evaluar expresiones en Scheme. Vamos continuar con ejemplos concretos de otras características funcionales características funcionales de Scheme.

En concreto, veremos:

- Símbolos y primitiva **quote**
- Uso de listas
- Definición de funciones recursivas en Scheme

### Funciones y formas especiales

En el seminario de Scheme hemos visto un conjunto de primitivas que podemos utilizar en Scheme.

Podemos clasificar las primitivas en **funciones** y **formas especiales**. Las funciones se evalúan usando el modelo de sustitución aplicativo ya visto:

- Primero se evalúan los argumentos y después se sustituye la llamada a la función por su cuerpo y se vuelve a evaluar la expresión resultante.
- Las expresiones siempre se evalúan desde los paréntesis interiores a los exteriores.

Las *formas especiales* son expresiones primitivas de Scheme que tienen una forma de evaluarse propia, distinta de las funciones.

### Formas especiales en Scheme

Veamos la forma de evaluar las distintas formas especiales en Scheme. En estas formas especiales no se aplica el modelo de sustitución, al no ser invocaciones de funciones, sino que cada una se evalúa de una forma diferente.

#### **Forma especial **define****

##### **Sintaxis**

```
(define <identificador> <expresión>)
```

## Evaluación

1. Evaluar expresión
2. Asociar el valor resultante con el *identificador*

## Ejemplo

```
(define base 10) ; Asociamos a 'base' el valor 10
(define altura 12) ; Asociamos a 'altura' el valor 12
(define area (/ (* base altura) 2)) ; Asociamos a 'area' el valor 60
```

## Forma especial **define** para definir funciones

### Sintaxis

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

## Evaluación

La semana que viene veremos con más detalle la semántica, y explicaremos la forma especial **lambda** que es la que realmente crea la función. Hoy nos quedamos en la siguiente descripción de alto nivel de la semántica:

1. Crear la función con el *cuerpo*
2. Dar a la función el nombre *nombre-función*

## Ejemplo

```
(define (factorial x)
  (if (= x 0)
      1
      (* x (factorial (- x 1)))))
```

## Forma especial **if**

### Sintaxis

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

## Evaluación

1. Evaluar *condición*
2. Si el resultado es **#t** evaluar la *expresión-true*, en otro caso, evaluar la *expresión-false*

## Ejemplo

```
(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"

!!! Note "Nota" Al ser **if** una forma especial, no se evalúa utilizando el modelo de sustitución, sino usando las reglas propias de la forma especial.

Por ejemplo, veamos la siguiente expresión:

```
```racket
(if (> 3 0) (+ 2 3) (/ 1 0)) ; => 5
```

```

Si se evaluara con el modelo de sustitución se lanzaría un error de división por cero al intentar evaluar `(/ 1 0)`. Sin embargo, esa expresión no llega a evaluarse, porque la condición `(> 3 0)` es cierta y sólo se evalúa la suma `(+ 2 3)`.

## Forma especial **cond**

### Sintaxis

```
(cond
  (<exp-cond-1> <exp-consec-1>)
  (<exp-cond-2> <exp-consec-2>)
  ...
  (else <exp-consec-else>))
```

### Evaluación

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

## Ejemplo

```
(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"))

;; Se evalúan una a una las expresiones (> 3 4),
;; (< 2 1), (= 3 1) y (> 3 5). Como ninguna de ella
;; es cierta se devuelve la cadena "ninguna condición es cierta".
```

## Formas especiales **and** y **or**

Las expresiones lógicas **and** y **or** no son funciones, sino formas especiales. Lo podemos comprobar con el siguiente ejemplo:

```
(and #f (/ 3 0)) ; => #f
(or #t (/ 3 0)) ; => #t
```

Si **and** y **or** fueran funciones, seguirían la regla que hemos visto de evaluar primero los argumentos y después invocar a la función con los resultados. Esto produciría un error al evaluar la expresión **(/ 3 0)**, al ser una división por 0.

Sin embargo, vemos que las expresiones no dan error y devuelven un valor booleano. ¿Por qué? Porque **and** y **or** no son funciones, sino formas especiales que se evalúan de forma diferente a las funciones.

En concreto, **and** y **or** van evaluando los argumentos hasta que encuentran un valor que hace que ya no sea necesario evaluar el resto.

## Sintaxis

```
(and exp1 ... expn)
(or exp1 ... expn)
```

## Evaluación **and**

- Se evalúa la expresión 1. Si el resultado es **#f**, se devuelve **#f**, en otro caso, se evalúa la siguiente expresión.
- Se repite hasta la última expresión, cuyo resultado se devuelve.

## Ejemplos **and**

```
(and #f (/ 3 0)) ; => #f
(and #t (> 2 1) (< 5 10)) ; => #t
(and #t (> 2 1) (< 5 10) (+ 2 3)) ; => 5
```

La regla de evaluación de **and** hace que sea posible que devuelva resultados no booleanos, como el último ejemplo. Sin embargo, no es recomendable usarlo de esta forma y en la asignatura no lo vamos a hacer nunca.

## Evaluación or

- Se evalúa la expresión 1. Si el resultado es distinto de #f se devuelve ese resultado. Si el resultado es #f se evalúa la siguiente expresión.
- Se repite hasta la última expresión, cuyo resultado se devuelve.

## Ejemplos or

```
(or #t (/ 3 0)) ; => #t
(or #f (< 2 10) (> 5 10)) ; => #t
(or (+ 2 3) (> 5 10)) ; => 5
```

Al igual que and, la regla de evaluación de or hace que sea posible que devuelva resultados no booleanos, como el último ejemplo. Tampoco es recomendable usarlo de esta forma.

Forma especial quote y símbolos

## Sintaxis

```
(quote <identificador>)
```

## Evaluación

- Se devuelve el identificador sin evaluar (un símbolo).
- Se abrevia en con el carácter '.

## Ejemplos

```
(quote x) ; el símbolo x
'hola ; el símbolo hola
```

A diferencia de los lenguajes imperativos, Scheme trata a los *identificadores* (nombres que se les da a las variables) como datos del lenguaje de tipo **symbol**. En el paradigma funcional a los identificadores se les denomina *símbolos*.

Los símbolos son distintos de las cadenas. Una cadena es un tipo de dato **compuesto** y se guardan en memoria todos y cada uno de los caracteres que la forman. Sin embargo, los símbolos son tipos atómicos, que se representan en memoria con un único valor determinado por el *código hash* del identificador.

Ejemplos de funciones Scheme con símbolos:

```
(define x 12)
(symbol? 'x) ; => #t
(symbol? x) ; => #f ¿Por qué?
(symbol? 'hola-que<>)
```

```
(symbol->string 'hola-que>)
'mañana
'lápiz ; aunque sea posible, no vamos a usar acentos en los símbolos
; pero sí en los comentarios
(symbol? "hola") ; #f
(symbol? #f) ; #f
(symbol? (car '(hola cómo estás))) ; #t
(equal? 'hola 'hola)
(equal? 'hola "hola")
```

Como hemos visto anteriormente, un símbolo puede asociarse o ligarse (*bind*) a un valor (cualquier dato de *primera clase*) con la forma especial **define**.

```
(define pi 3.14159)
```

!!! Note "Nota" No es correcto escribir (**define 'pi 3.14156**) porque la forma especial **define** debe recibir un identificador *sin quote*.

Cuando escribimos un símbolo en el prompt de Scheme el intérprete lo evalúa y devuelve su valor:

```
> pi
3.14159
```

Los nombres de las funciones (**equal?**, **sin**, **+**, ...) son también símbolos y Scheme también los evalúa (en un par de semanas hablaremos de las funciones como objetos primitivos en Scheme):

```
> sin
#<procedure:sin>
> +
#<procedure:+>
> (define (cuadrado x) (* x x))
> cuadrado
#<procedure:cuadrado>
```

Los símbolos son tipos primitivos del lenguaje: pueden pasarse como parámetros o ligarse a variables.

```
> (define x 'hola)
> x
hola
```

Forma especial **quote** con expresiones

## Sintaxis

```
(quote <expresión>)
```

## Evaluación

Si **quote** recibe una expresión correcta de Scheme (una expresión entre paréntesis) se devuelve la lista o pareja pareja definida por la expresión (sin evaluar sus elementos).

## Ejemplos

```
'(1 2 3) ; => (1 2 3) Una lista
'(+ 1 2 3 4) ; La lista formada por el símbolo + y los números 1 2 3 4
(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

Otra de las características fundamentales del paradigma funcional es la utilización de listas. Ya hemos visto en el seminario de Scheme las funciones más importantes para trabajar con ellas. Vamos a repasarlas de nuevo en este apartado, antes de ver algún ejemplo de cómo usar la recursión con listas.

Ya hemos visto en dicho seminario que Scheme es un lenguaje débilmente tipado. Una variable o parámetro no se declara de un tipo y puede contener cualquier valor. Sucede igual con las listas: una lista en Scheme puede contener cualquier valor, incluyendo otras listas.

### Diferencia entre la función **list** y la forma especial **quote**

En el seminario de Scheme explicamos que podemos crear listas de forma dinámica, llamando a la función **list** y pasándole un número variable de parámetros que son los elementos que se incluirán en la lista:

```
(list 1 2 3 4 5) ; => (1 2 3 4)
(list 'a 'b 'c) ; => (a b c)
(list 1 'a 2 'b 3 'c #t) ; => (1 a 2 b 3 c #t)
(list 1 (+ 1 1) (* 2 (+ 1 2))) ; => (1 2 6)
```

Las expresiones interiores se evalúan y se llama a la función **list** con los valores resultantes.

Otro ejemplo:

```
(define a 1)
(define b 2)
(define c 3)
(list a b c) ; => (1 2 3)
```

Como hemos visto cuando hemos hablado de **quote**, esta forma especial también puede construir una lista. Pero lo hace sin evaluar sus elementos.

Por ejemplo:

```
'(1 2 3 4) ; => (1 2 3 4)
(define a 1)
(define b 2)
(define c 3)
'(a b c) ; => (a b c)
'(1 (+ 1 1) (* 2 (+ 1 2))) ; => (1 (+ 1 1) (* 2 (+ 1 2)))
```

La última lista tiene 3 elementos:

- El número 1
- La lista `(+ 1 1)`
- La lista `(* 2 (+ 1 2))`

Es posible definir una lista vacía (sin elementos) realizando una llamada sin argumentos a la función **list** o utilizando el símbolo `():

```
(list) ; => ()
`() ; => ()
```

La diferencia entre creación de listas con la función **list** y con la forma especial **quote** se puede comprobar en los ejemplos.

La evaluación de la función **list** funciona como cualquier función, primero se evalúan los argumentos y después se invoca a la función con los argumentos evaluados. Por ejemplo, en la siguiente invocación se obtiene una lista con cuatro elementos resultantes de las invocaciones de las funciones dentro del paréntesis:

```
(list 1 (/ 2 3) (+ 2 3)) ; => (1 2/3 5)
```

Sin embargo, usamos **quote** obtenemos una lista con sublistas con símbolos en sus primeras posiciones:

```
'(1 (/ 2 3) (+ 2 3)) ; => (1 (/ 2 3) (+ 2 3))
```

### Selección de elementos de una lista: **car** y **cdr**

En el seminario vimos también cómo obtener los elementos de una lista.

- Primer elemento: función **car**

- Resto de elementos: función **cdr** (los devuelve en forma de lista)

Ejemplos:

```
(define lista1 '(1 2 3 4))
(car lista1) ; => 1
(cdr lista1) ; => (2 3 4)
(define lista2 '((1 2) 3 4))
(car lista2) => (1 2)
(cdr lista2) => (3 4)
```

### Composición de listas: **cons** y **append**

Por último, en el seminario vimos también cómo crear nuevas listas a partir de ya existentes con las funciones **cons** y **append**.

La función **cons** crea una lista nueva resultante de añadir un elemento al comienzo de la lista. Esta función es la forma habitual de construir nuevas listas a partir de una lista ya existente y un nuevo elemento.

```
(cons 1 '(1 2 3 4)) ; => (1 1 2 3 4)
(cons 'hola '(como estás)) ; => (hola como estás)
(cons '(1 2) '(1 2 3 4)) ; => ((1 2) 1 2 3 4)
```

La función **append** se usa para 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

Otra característica fundamental de la programación funcional es la no existencia de bucles. Un bucle implica la utilización de pasos de ejecución en el programa y esto es característico de la programación imperativa.

En programación funcional las iteraciones se realizan con recursión.

### Función (**suma-hasta x**)

Por ejemplo, podemos definir la función (**suma-hasta x**) que devuelve la suma de los números hasta el parámetro **x** cuyo valor pasamos en la invocación de la función.

Por ejemplo, (**suma-hasta 5**) devolverá **0+1+2+3+4+5 = 15**.

La definición de la función es la siguiente:

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

En una definición recursiva siempre tenemos un **caso general** y un **caso base**. 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.

El **caso base** es el caso en el que **x** vale 0. En este caso devolvemos el propio 0, no hay que realizar ningún cálculo.

El **caso general** es en el que se realiza la llamada recursiva. Esta llamada devuelve un valor que se utiliza para cálculo final evaluando la expresión del caso general con valores concretos.

En programación funcional, al no existir efectos laterales, lo único que importa cuando realizamos una recursión es el valor devuelto por la llamada recursiva. Ese valor devuelto se combina con el resto de la expresión del caso general para construir el valor resultante.

!!! Important "Importante" Para entender la recursión no es conveniente utilizar el depurador, ni hacer trazas, ni *entrar en la recursión*, sino que hay que suponer que **la llamada recursiva se ejecuta y devuelve el valor que debería. ¡Debemos confiar en la recursión!**.

El caso general del ejemplo anterior indica lo siguiente:

Para calcular la suma hasta **x**:

Llamamos a la recursión para que calcule la suma hasta **x-1**  
(confiamos en que la implementación funciona bien y esta llamada nos devolverá el resultado hasta **x-1**) y a ese resultado le sumamos el propio número **x**.

Siempre es aconsejable usar un ejemplo concreto para probar el caso general. Por ejemplo, el caso general de la suma hasta 5 se calculará de la siguiente forma:

```
(+ (suma-hasta (- 5 1)) 5) ; =>
(+ (suma-hasta 4) 5) ; => confiamos en la recursión:
;           (suma-hasta 4) = 4+3+2+1 = 10 =>
(+ 10 5) ; =>
15
```

La evaluación de esta función calculará la llamada recursiva (**suma-hasta 4**). Ahí es donde debemos **confiar en que la recursión hace bien su trabajo** y que esa llamada devuelve el valor resultante de  $4+3+2+1$ , o sea, 10. Una vez obtenido ese valor hay que terminar el cálculo sumándole el propio número 5.

Otra característica necesaria del caso general en una definición recursiva, que también vemos en este ejemplo, es que **la llamada recursiva debe trabajar sobre un caso más sencillo que la llamada general**. De esta

forma la recursión va descomponiendo el problema hasta llegar al caso base y construye la solución a partir de ahí.

En nuestro caso, la llamada recursiva para calcular la suma hasta 5 se hace calculando la suma hasta 4 (un caso más sencillo).

### Diseño de la función (`suma-hasta x`)

¿Cómo hemos diseñado esta función? ¿Cómo hemos llegado a la solución?

Debemos empezar teniendo claro qué es lo que queremos calcular. Lo mejor es utilizar un ejemplo.

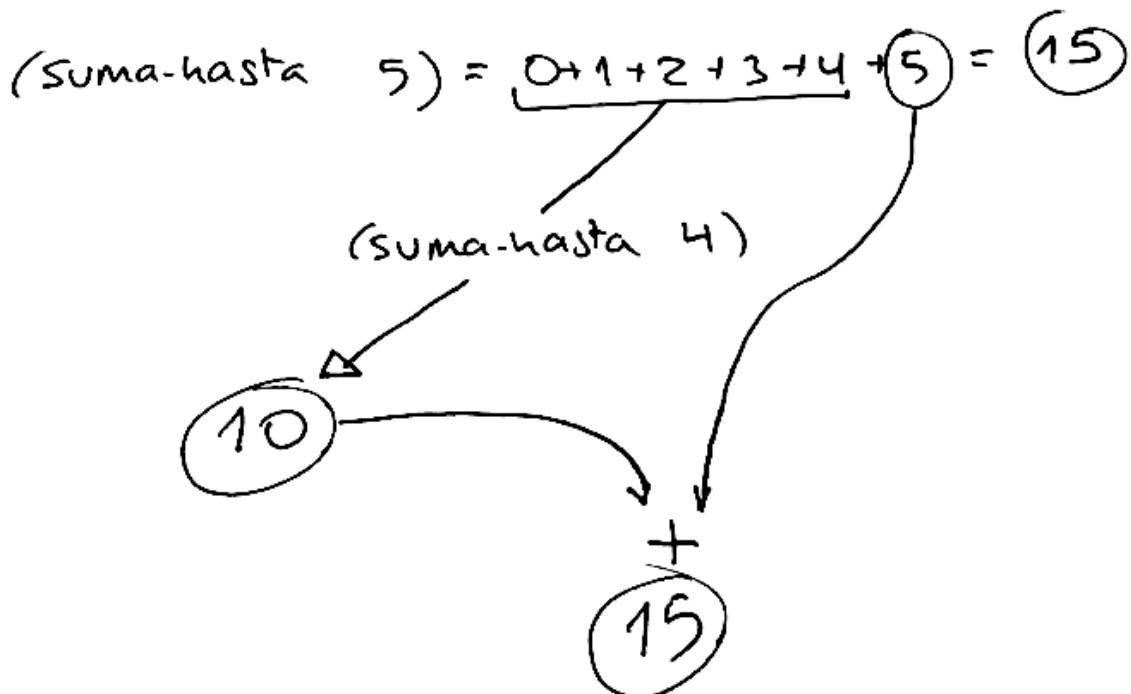
Por ejemplo, (`suma-hasta 5`) devolverá  $0+1+2+3+4+5 = 15$ .

Una vez que tenemos esta expresión de un ejemplo concreto debemos diseñar el caso general de la recursión. Para ello tenemos que encontrar una expresión para el cálculo de (`suma-hasta 5`) que **use una llamada recursiva** a un problema más pequeño.

O, lo que es lo mismo, ¿podemos obtener el resultado 15 con lo que nos devuelve una llamada recursiva que obtenga la suma hasta un número más pequeño y haciendo algo más?

Pues sí: para calcular la suma hasta 5, esto es, para obtener 15, podemos llamar a la recursión para calcular la suma hasta 4 (devuelve 10) y a este resultado sumarle el propio 5.

Lo podemos expresar con el siguiente dibujo:



Generalizamos este ejemplo y lo expresamos en Scheme de la siguiente forma:

```
(define (suma-hasta x)
  (+ (suma-hasta (- x 1)) x))
```

Nos falta el caso base de la recursión. Debemos preguntarnos **¿cuál es el caso más sencillo del problema, que podemos calcular sin hacer ninguna llamada recursiva?**. En este caso podría ser el caso en el que **x** es 0, en el que devolveríamos 0.

Podemos ya escribirlo todo en Scheme:

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

Una aclaración sobre el caso general. En la implementación anterior la llamada recursiva a **suma-hasta** se realiza en el primer argumento de la suma:

```
(+ (suma-hasta (- x 1)) x)
```

La expresión anterior es totalmente equivalente a la siguiente en la que la llamada recursiva aparece como segundo argumento

```
(+ x (suma-hasta (- x 1)))
```

Ambas expresiones son equivalentes porque en programación funcional no importa el orden en el que se evalúan los argumentos. Da lo mismo evaluarlos de derecha a izquierda que de izquierda a derecha. La transparencia referencial garantiza que el resultado es el mismo.

### Función (**alfabeto-hasta char**)

Vamos con otro ejemplo. Queremos diseñar una función (**alfabeto-hasta char**) que devuelva una cadena que empieza en la letra **a** y termina en el carácter que le pasamos como parámetro.

Por ejemplo:

```
(alfabeto-hasta #\h) ; => "abcdefghijklmnpqrstuvwxyz"
(alfabeto-hasta #\z) ; => "abcdefghijklmnopqrstuvwxyz"
```

Pensamos en el caso general: ¿cómo podríamos invocar a la propia función **alfabeto-hasta** para que (confiando en la recursión) nos haga gran parte del trabajo (construya casi toda la cadena con el alfabeto)?

Podríamos hacer que la llamada recursiva devolviera el alfabeto hasta el carácter previo al que nos pasan como parámetro y después nosotros añadir ese carácter a la cadena que devuelve la recursión.

Veamos un ejemplo concreto:

```
(alfabeto-hasta #\h) = (alfabeto-hasta #\g) + \#h
```

La llamada recursiva `(alfabeto-hasta #\g)` devolvería la cadena "abcdefg" (confiando en la recursión) y sólo faltaría añadir la última letra.

Para implementar esta idea en Scheme lo único que necesitamos es usar la función `string-append` para concatenar cadenas y una función auxiliar `(anterior char)` que devuelve el carácter anterior a uno dado.

```
(define (anterior char)
  (integer->char (- (char->integer char) 1)))
```

El caso general quedaría como sigue:

```
(define (alfabeto-hasta char)
  (string-append (alfabeto-hasta (anterior char)) (string char)))
```

Faltaría el caso base. ¿Cuál es el caso más sencillo posible que nos pueden pedir? El caso del alfabeto hasta la `#\a`. En ese caso basta con devolver la cadena "a".

La función completa quedaría así:

```
(define (alfabeto-hasta char)
  (if (equal? char #\a)
      "a"
      (string-append (alfabeto-hasta (anterior char)) (string char))))
```

## Recursión y listas

La utilización de la recursión es muy útil para trabajar con estructuras secuenciales, como listas. Vamos a empezar viendo unos sencillos ejemplos y más adelante veremos algunos más complicadas.

### Función recursiva `suma-lista`

Veamos un primer ejemplo, la función `(suma-lista lista-nums)` que recibe como parámetro una lista de números y devuelve la suma de todos ellos.

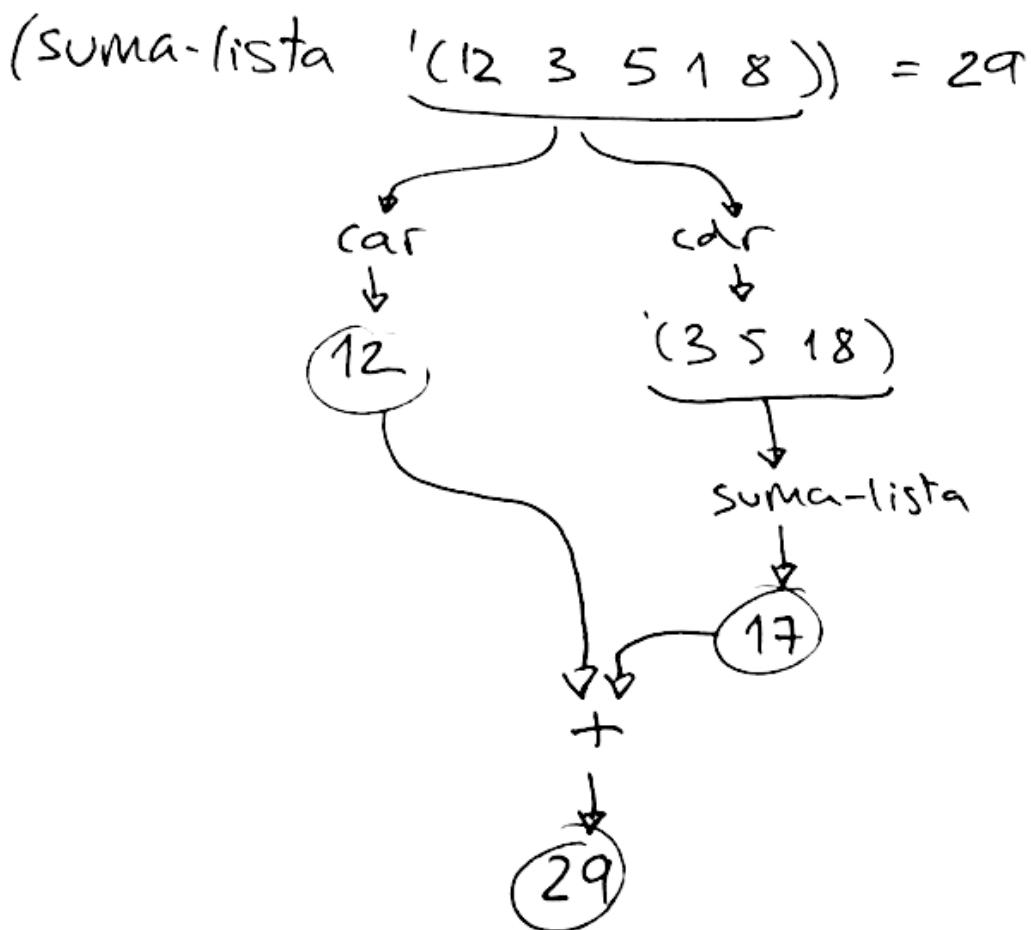
Siempre debemos empezar escribiendo un ejemplo de la función, para entenderla bien:

```
(suma-lista '(12 3 5 1 8)) ; => 29
```

Para diseñar una implementación recursiva de la función tenemos que pensar en cómo descomponer el ejemplo en una llamada recursiva a un problema más pequeño y en cómo tratar el valor devuelto por la

recursión para obtener el valor esperado.

Por ejemplo, en este caso podemos pensar que para sumar la lista de números (12 3 5 1 8) podemos obtener un problema más sencillo (una lista más pequeña) haciendo el `cdr` de la lista de números y llamando a la recursión con el resultado. La llamada recursiva devolverá la suma de esos números (confiamos en la recursión) y a ese valor basta con sumarle el primer número de la lista. Lo podemos representar en el siguiente dibujo:



Podemos generalizar este ejemplo y expresarlo en Scheme de la siguiente forma:

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

Falta el caso base. ¿Cuál es la lista más sencilla con la que podemos calcular la suma de sus elementos sin llamar a la recursión?. Podría ser una lista sin elementos, y devolvemos 0. O una lista con un único elemento, y devolvemos el propio elemento. Escogemos como caso base el primero de ellos.

Con todo junto, la recursión quedaría como sigue:

```
(define (suma-lista lista)
  (if (null? lista)
```

```

0
(+ (car lista) (suma-lista (cdr lista))))

```

## Función recursiva **longitud**

Veamos cómo definir la función recursiva que devuelve la longitud de una lista, el número de elementos que contiene.

Comencemos como siempre con un ejemplo:

```
(longitud '(a b c d e)) ; => 5
```

Suponiendo que la función **longitud** funciona correctamente, ¿cómo podríamos formular el caso general de la recursión? ¿cómo podríamos llamar a la recursión con un problema más pequeño y cómo podemos aprovechar el resultado de esta llamada para obtener el resultado final?

En este caso es bastante sencillo. Si a la lista le quitamos un elemento, cuando llamemos a la recursión nos va a devolver la longitud original menos uno. En este caso:

```
(longitud (cdr '(a b c d e))) ; =>
(longitud '(b c d e)) => (confiamos en la recursión) 4
```

De esta forma, para conseguir la longitud de la lista inicial, sólo habría que sumarle 1 a lo que nos devuelve la llamada recursiva.

Si expresamos en Scheme este caso general:

```
; Sólo se define el caso general, falta el caso base
(define (longitud lista)
  (+ (longitud (cdr lista)) 1))
```

Para definir el caso base debemos preguntarnos cuál es el caso más simple que le podemos pasar a la función. Si en cada llamada recursiva vamos reduciendo la longitud de la lista, el caso base recibirá la lista vacía. ¿Cuál es la longitud de una lista vacía? Una lista vacía no tiene elementos, por lo que es 0.

De esta forma completamos la definición de la función:

```
(define (longitud lista)
  (if (null? lista)
    0
    (+ (longitud (cdr lista)) 1)))
```

En Scheme existe la función **length** que hace lo mismo. Devuelve la longitud de una lista:

```
(length '(a b c d e)) ; => 5
```

## Cómo comprobar si una lista tiene un único elemento

En el caso base de algunas funciones recursivas es necesario comprobar que la lista que se pasa como parámetro tiene un único elemento. Por ejemplo, en el caso base de la función recursiva que comprueba si una lista está ordenada.

Al estar definida la función `length` en Scheme la primera idea que se nos puede ocurrir es comprobar si la longitud de la lista es 1. Sin embargo es una mala idea.

```
; Ejemplo de función recursiva con un caso
; base en el que se comprueba si la lista tienen
; un único elemento
; ¡¡MALA IDEA, NO HACERLO ASÍ!!
(define (foo lista)
  (if (= (length lista) 1)
      ; devuelve caso base
      ; caso general
    ))
```

El problema de la implementación anterior es que el coste de la función `length` es lineal. Tal y como hemos visto en el apartado anterior, para calcular la longitud de la lista es necesario recorrer todos sus elementos. Además, la función recursiva hace esa comprobación en cada llamada recursiva. El coste resultante de la función `foo`, por tanto, es cuadrático.

¿Cómo mejorar el coste? Hay que tener en cuenta que la comprobación anterior está haciendo cosas de más. Realmente no queremos saber la longitud de la lista sino únicamente si esa longitud es mayor que uno. Esta comprobación sí que puede hacerse en tiempo constante. Lo único que debemos hacer es comprobar si el `cdr` de la lista es la lista vacía. Si lo es, ya sabemos que la lista original tenía un único elemento.

Por tanto, la versión correcto del código anterior sería la siguiente:

```
; Versión correcta para comprobar si una lista tiene
; un único elemento
(define (foo lista)
  (if (null? (cdr lista))
      ; devuelve caso base
      ; caso general
    ))
```

El coste de la comprobación (`null? (cdr lista)`) es constante. No depende de la longitud de la lista.

## Función recursiva veces

Como último ejemplo vamos a definir la función

```
(veces lista id)
```

que cuenta el número de veces que aparece un identificador en una lista.

Por ejemplo,

```
(veces '(a b c a d a) 'a) ; => 3
```

¿Cómo planteamos el caso general? Llamaremos a la recursión con el resto de la lista. Esta llamada nos devolverá el número de veces que aparece el identificador en este resto de la lista. Y sumaremos al valor devuelto 1 si el primer elemento de la lista coincide con el identificador.

En Scheme hay que definir este caso general en una única expresión:

```
(if (equal? (car lista) id)
    (+ 1 (veces (cdr lista) id))
    (veces (cdr lista) id))
```

Como caso base, si la lista es vacía devolvemos 0.

La versión completa:

```
(define (veces lista id)
  (cond
    ((null? lista) 0)
    ((equal? (car lista) id) (+ 1 (veces (cdr lista) id)))
    (else (veces (cdr lista) id)))

(veces '(a b a a b b) 'a) ; => 3
```

## Tipos de datos compuestos en Scheme

El tipo de dato pareja

### Función de construcción de parejas **cons**

Ya hemos visto en el seminario de Scheme que el tipo de dato compuesto más simple es la pareja: una entidad formada por dos elementos. Se utiliza la función **cons** para construirla:

```
(cons 1 2) ; => (1 . 2)
(define c (cons 1 2))
```

Dibujamos la pareja anterior y la variable **c** que la referencia de la siguiente forma:



### *Tipo compuesto pareja*

La instrucción **cons** construye un dato compuesto a partir de otros dos datos (que llamaremos izquierdo y derecho). La expresión **(1 . 2)** es la forma que el intérprete tiene de imprimir las parejas.

### **Construcción de parejas con quote**

Al igual que las listas, es posible construir parejas con la forma especial **quote**, definiendo la pareja entre paréntesis y separando su parte izquierda y derecha con un punto:

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

Utilizaremos a veces **cons** y otras veces **quote** para definir parejas. Pero hay que tener en cuenta que, al igual que con las listas, **quote** no evalúa sus parámetros, por lo que no lo deberemos utilizar por ejemplo dentro de una función en la que queremos construir una pareja con los resultados de evaluar expresiones.

Por ejemplo:

```
(define a 1)
(define b 2)
(cons a b) ; => (1 . 2)
'(a . b) ; => (a . b)
```

### **Funciones de acceso car y cdr**

Una vez construida una pareja, podemos obtener el elemento correspondiente a su parte izquierda con la función **car** y su parte derecha con la función **cdr**:

```
(define c (cons 1 2))
(car c) ; => 1
(cdr c) ; => 2
```

### **Definición declarativa**

Las funciones **cons**, **car** y **cdr** quedan perfectamente definidas con las siguientes ecuaciones algebraicas:

```
(car (cons x y)) = x
(cdr (cons x y)) = y
```

!!! Note "¿De dónde vienen los nombres **car** y **cdr**?" Inicialmente los nombres eran CAR y CDR (en mayúsculas). La historia se remonta al año 1959, en los orígenes del Lisp y tiene que ver con el nombre que se les daba a ciertos registros de la memoria del IBM 709.

Podemos leer la explicación completa en  
 [The origin of CAR and CDR in LISP]([http://www.iwriteiam.nl/HaCAR\\_CDR.html](http://www.iwriteiam.nl/HaCAR_CDR.html)).

## Función pair?

La función **pair?** nos dice si un objeto es atómico o es una pareja:

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

## Las parejas pueden contener cualquier tipo de dato

Ya hemos comprobado que Scheme es un lenguaje *débilmente tipado*. Las funciones pueden devolver y recibir distintos tipos de datos.

Por ejemplo, podríamos definir la siguiente función **suma** que sume tanto números como cadenas:

```
(define (suma x y)
  (cond
    ((and (number? x) (number? y)) (+ x y))
    ((and (string? x) (string? y)) (string-append x y))
    (else 'error)))
```

En la función anterior los parámetros **x** e **y** pueden ser números o cadenas (o incluso de cualquier otro tipo). Y el valor devuelto por la función será un número, una cadena o el símbolo '**error**'.

Sucede lo mismo con el contenido de las parejas. Es posible guardar en las parejas cualquier tipo de dato y combinar distintos tipos. Por ejemplo:

```
(define c (cons 'hola #f))
(car c) ; => 'hola
(cdr c) ; => #f
```

## Las parejas son objetos inmutables

Recordemos que en los paradigmas de programación declarativa y funcional no existe el *estado mutable*. Una vez declarado un valor, no se puede modificar. Esto debe suceder también con las parejas: una vez creada una pareja no se puede modificar su contenido.

En Lisp y Scheme estándar las parejas sí que pueden ser mutadas. Pero durante toda esta primera parte de la asignatura no lo contemplaremos, para no salirnos del paradigma funcional.

En Swift y otros lenguajes de programación es posible definir **estructuras de datos inmutables** que no pueden ser modificadas una vez creadas. Lo veremos también más adelante.

## Las parejas son objetos de primera clase

En un lenguaje de programación un elemento es de primera clase cuando puede:

- Asignarse a variables
- Pasarse como argumento
- Devolverse por una función
- Guardarse en una estructura de datos mayor

Las parejas son objetos de primera clase.

Una pareja puede asignarse a una variable:

```
(define p1 (cons 1 2))
(define p2 (cons #f "hola"))
```

Una pareja puede pasarse como argumento y devolverse en una función:

```
(define (suma-parejas p1 p2)
  (cons (+ (car p1) (car p2))
        (+ (cdr p1) (cdr p2)))))

(suma-parejas '(1 . 5) '(4 . 12)) ; ⇒ (5 . 17)
```

Una vez definida esta función `suma-parejas` podríamos ampliar la función `suma` que vimos previamente con este nuevo tipo de datos:

```
(define (suma x y)
  (cond
    ((and (number? x) (number? y)) (+ x y))
    ((and (string? x) (string? y)) (string-append x y))
    ((and (pair? x) (pair? y)) (suma-parejas p1 p2))
    (else 'error)))
```

Y, por último, las parejas *pueden formar parte de otras parejas*.

Es lo que se denomina la **propiedad de clausura de la función cons**: el resultado de un **cons** puede usarse como parámetro de nuevas llamadas a **cons**.

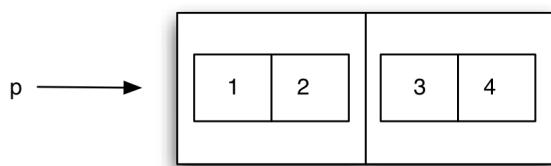
Ejemplo:

```
(define p1 (cons 1 2))
(define p2 (cons 3 4))
(define p (cons p1 p2))
```

Expresión equivalente:

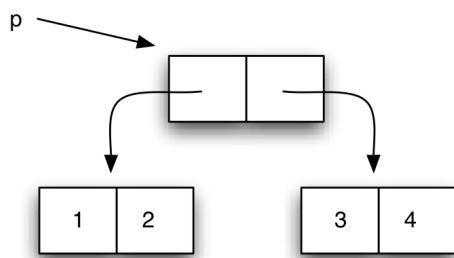
```
(define p (cons (cons 1 2)
                 (cons 3 4)))
```

Podríamos representar esta estructura así:



*Propiedad de clausura: las parejas pueden contener parejas*

Pero se haría muy complicado representar muchos niveles de anidamiento. Por eso utilizamos la siguiente representación:



Llamamos a estos diagramas *diagramas caja-y-puntero* (*box-and-pointer* en inglés).

Diagramas *caja-y-puntero*

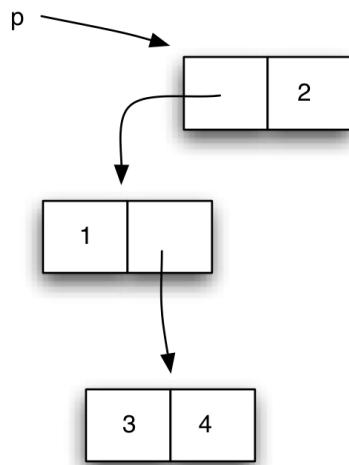
Al escribir expresiones complicadas con **cons** anidados es conveniente para mejorar su legibilidad utilizar el siguiente formato:

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

Para entender la construcción de estas estructuras es importante recordar que las expresiones se evalúan *de dentro a afuera*.

¿Qué figura representaría la estructura anterior?

Solución:

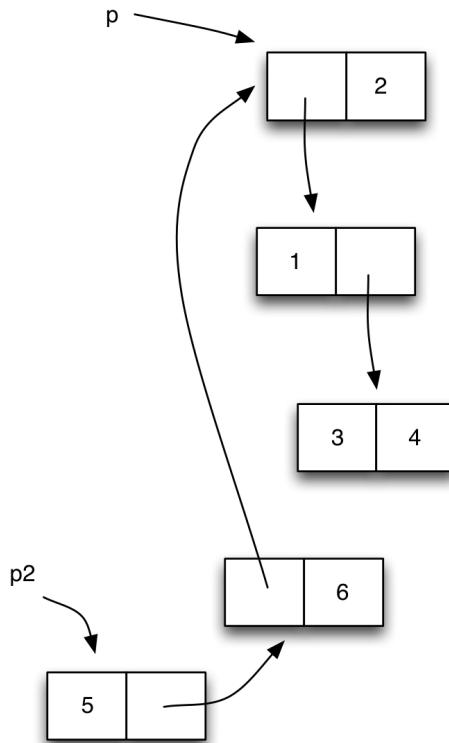


Es importante tener en cuenta que cada caja del diagrama representa una pareja creada en la memoria del intérprete con la instrucción `cons` y que el resultado de evaluar una variable en la que se ha guardado una pareja devuelve la pareja recién creada. Por ejemplo, si el intérprete evalúa `p` después de haber hecho la sentencia anterior devuelve la pareja contenida en `p`, no se crea una pareja nueva.

Por ejemplo, si después de haber evaluado la sentencia anterior evaluamos la siguiente:

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

El diagrama caja y puntero resultante sería el siguiente:



Vemos que en la pareja que se crea con `(cons p 6)` se guarda en la parte izquierda **la misma pareja que hay en p**. Lo representamos con una flecha que apunta a la misma pareja que `p`.

!!! Note "Nota"

El funcionamiento de la evaluación de variables que contienen parejas es similar al de las variables que contienen objetos en lenguajes orientados a objetos como Java. Cuando se evalúa una variable que contiene una pareja se devuelve la propia pareja, no una copia.

En programación funcional, como el contenido de las parejas es inmutable, no hay problemas de \*efectos laterales\* por el hecho de que una pareja esté compartida.

Es conveniente que pruebes a crear distintas estructuras de parejas con parejas y a dibujar su diagrama caja y puntero. Y también a recuperar un determinado dato (pareja o dato atómico) una vez creada la estructura.

La siguiente función `print-pareja` puede ser útil a la hora de mostrar por pantalla los elementos de una pareja

```
(define (print-pareja pareja)
  (if (pair? pareja)
      (begin
          (display "(")
          (print-dato (car pareja))
          (display " . ")
          (print-dato (cdr pareja))
          (display ")")))
      (display "#"))
```

```
(display ""))
(define (print-dato dato)
  (if (pair? dato)
      (print-pareja dato)
      (display dato)))
```

!!! Warning "¡Cuidado!" La función anterior contiene pasos de ejecución con sentencias como `begin` y llamadas a `display` dentro del código de la función. Estas sentencias son propias de la programación imperativa. **No hacerlo en programación funcional.**

## Funciones c????r

Al trabajar con estructuras de parejas anidadas es muy habitual realizar llamadas del tipo:

```
(cdr (cdr (car p))) ; => 4
```

Es equivalente a la función `cadar` de Scheme:

```
(cddar p) ; => 4
```

El nombre de la función se obtiene concatenando a la letra "c", las letras "a" o "d" según hagamos un car o un cdr y terminando con la letra "r".

Hay definidas  $2^4$  funciones de este tipo: `caaaar`, `caaadr`, ..., `cddddr`.

## Listas en Scheme

### Implementación de listas en Scheme

Recordemos que Scheme permite manejar listas como un tipo de datos básico. Hemos visto funciones para crear, añadir y recorrer listas.

Como repaso, podemos ver las siguientes expresiones. Fijaros que las funciones `car`, `cdr` y `cons` son exactamente las mismas funciones que las vistos anteriormente.

¿Por qué? ¿Qué relación hay entre las parejas y las listas?

Hagamos algunas pruebas, probando si los resultados son listas o parejas usando las funciones `list?` y `pair?`.

Por ejemplo, una pareja formada por dos números es una pareja, pero no es una lista:

```
(define p1 (cons 1 2))
(pair? p1) ; => #t
(list? p1) ; => #f
```

Y una lista vacía es una lista, pero no es una pareja:

```
(list? '()) ; => #t
(pair? '()) ; => #f
```

¿Una lista es una pareja? Pues sí:

```
(define lista '(1 2 3))
(list? lista) ; => #t
(pair? lista) ; => #t
```

Por último, una pareja con una lista vacía como segundo elemento es una pareja y una lista:

```
(define p1 (cons 1 '()))
(pair? p1) ; => #t
(list? p1) ; => #t
```

Con estos ejemplos ya tenemos pistas para deducir la relación entre listas y parejas en Scheme (y Lisp). Vamos a explicarlo.

### Definición de listas con parejas

Una lista es (definición recursiva):

- Una pareja que contiene en su parte izquierda el primer elemento de la lista y en su parte derecha el resto de la lista
- Un símbolo especial '`()`' que denota la lista vacía

Por ejemplo, una lista muy sencilla con un solo elemento, `(1)`, se define con la siguiente pareja:

```
(cons 1 '())
```

La pareja cumple las condiciones anteriores:

- La parte izquierda de la pareja es el primer elemento de la lista (el número 1)
- La parte derecha es el resto de la lista (la lista vacía)

|   |                 |
|---|-----------------|
| 1 | <code>()</code> |
|---|-----------------|

### La lista (1)

El objeto es al mismo tiempo una pareja y una lista. La función `list?` permite comprobar si un objeto es una lista:

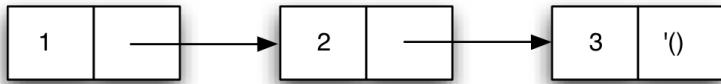
```
(define l (cons 1 '()))
(pair? l)
(list? l)
```

Por ejemplo, la lista '(1 2 3 4) se construye con la siguiente secuencia de parejas:

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

La primera pareja cumple las condiciones de ser una lista:

- Su primer elemento es el 1
- Su parte derecha es la lista '(2 3 4)



### Parejas formando una lista

Al comprobar la implementación de las listas en Scheme, entendemos por qué las funciones `car` y `cdr` nos devuelven el primer elemento y el resto de la lista.

### Lista vacía

La lista vacía es una lista:

```
(list? '()) ; => #t
```

Y no es un símbolo ni una pareja:

```
(symbol? '()) ; => #f
(pair? '()) ; => #f
```

Para saber si un objeto es la lista vacía, podemos utilizar la función `null?`:

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

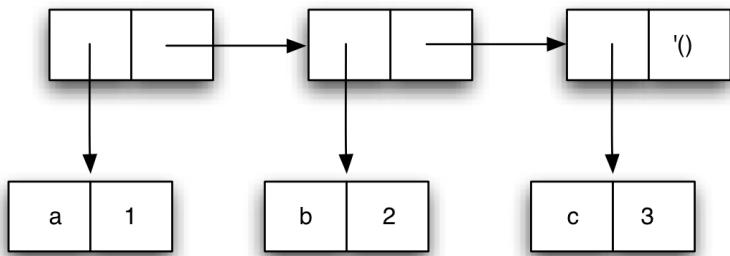
## Listas con elementos compuestos

Las listas pueden contener cualquier tipo de elementos, incluyendo otras parejas.

La siguiente estructura se denomina *lista de asociación*. Son listas cuyos elementos son parejas (*clave, valor*):

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

¿Cuál sería el diagrama *box and pointer* de la estructura anterior?



La expresión equivalente utilizando conses es:

```
(cons (cons 'a 1)
      (cons (cons 'b 2)
            (cons (cons 'c 3)
                  '()))))
```

## Listas de listas

Hemos visto que podemos construir listas que contienen otras listas:

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

La lista anterior también se puede definir con quote:

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

La lista resultante contiene tres elementos: el primero y el último son elementos atómicos (números) y el segundo es otra lista.

Si preguntamos por la longitud de la lista Scheme nos dirá que es una lista de 3 elementos:

```
(length lista) ; => 3
```

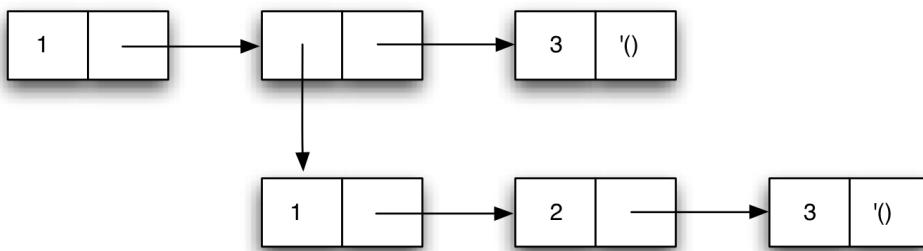
Y el segundo elemento de la lista es otra lista:

```
(car (cdr lista)) ; => (1 2 3)
```

¿Cómo implementa Scheme esta lista usando parejas?

Al ser una lista de tres elementos lo hará con tres parejas enlazadas que terminan en una lista vacía en la parte derecha de la última pareja. En las partes izquierdas de esas tres parejas tendremos los elementos de la lista propiamente dichos: un 1 y un 3 en la primera y última pareja y una lista en la segunda pareja.

El diagrama *box and pointer*:



*Lista que contiene otra lista como segundo elemento*

### Impresión de listas y parejas por el intérprete de Scheme

El intérprete de Scheme siempre intenta mostrar una lista cuando encuentra una pareja cuyo siguiente elemento es otra pareja.

Por ejemplo, si tenemos la siguiente estructura:

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

Cuando se evalúe **p** el intérprete imprimirá por pantalla lo siguiente:

```
(1 2 . 3)
```

¿Por qué? Porque el intérprete va construyendo la salida conforme recorre la pareja **p**. Como encuentra una pareja cuya parte derecha es otra pareja, lo interpreta como el comienzo de una lista, y por eso escribe **(1 2** en lugar de **(1 . 2**. Pero inmediatamente después se encuentra con el **3** en lugar de una lista vacía. En ese

momento el intérprete "se da cuenta" de que no tenemos una lista y termina la expresión escribiendo el . 3 y el paréntesis final.

Si queremos comprobar la estructura de parejas podemos utilizar la función `print-pareja` definida anteriormente, que imprimiría lo siguiente:

```
(print-pareja p) ; => (1 . (2 . 3))
```

## Funciones de alto nivel sobre listas

Es importante conocer cómo se implementan las listas usando parejas y su representación con diagramas caja y puntero para definir funciones de alto nivel. Algunas de estas funciones ya las conocemos, otras las vemos por primera vez en los siguientes ejemplos:

```
(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)
```

En los siguientes apartados veremos cómo están implementadas.

### Funciones recursivas que construyen listas

Para terminar el apartado sobre las listas en Scheme vamos a ver ejemplos adicionales de funciones recursivas que trabajan con listas. Veremos alguna función que recibe una lista y, como antes, usa la recursión para recorrerla. Pero veremos también funciones que usan la recursión para **construir nuevas listas**.

Algunas de las funciones que presentamos son implementaciones de las ya existentes en Scheme. Para no solapar con las definiciones de Scheme pondremos el prefijo `mi-` en todas ellas.

Vamos a ver las siguientes funciones:

- `mi-list-ref`: implementación de la función `list-ref`
- `mi-list-tail`: implementación de la función `list-tail`
- `mi-append`: implementación de la función `append`
- `mi-reverse`: implementación de la función `reverse`
- `cuadrados-hasta`: devuelve la lista de cuadrados hasta uno dado
- `filtra-pares`: devuelve la lista de los números pares de la lista que se recibe
- `primo?`: comprueba si un número es o no primo

### Función `mi-list-ref`

La función (`mi-list-ref n lista`) devuelve el elemento `n` de una lista (empezando a contar por 0):

```
(define lista '(a b c d e f g))
(mi-list-ref lista 2) ; => c
```

Veamos con el ejemplo anterior cómo hacer la formulación recursiva.

Hemos visto que, en general, cuando queremos resolver un problema de forma recursiva tenemos que hacer una llamada recursiva a un problema más sencillo, **confiar en que la llamada nos devuelva el resultado correcto** y usar ese resultado para resolver el problema original.

En este caso nuestro problema es obtener el número que está en la posición 2 de la lista (a b c d e f g). Suponemos que la función que nos devuelve una posición de la lista ya la tenemos implementada y que la llamada recursiva nos va a devolver el resultado correcto. ¿Cómo podemos simplificar el problema original? Veamos la solución para este caso concreto:

Para devolver el elemento 2 de la lista (a b c d e f g):  
 Hacemos el cdr de la lista (obtenemos (b c d e f g))  
 y devolvemos su elemento 1. Será el valor c (empezamos  
 a contar por 0).

Generalizamos el ejemplo anterior, para cualquier n y cualquier lista:

Para devolver el elemento que está en la posición `n` de una lista,  
 devuelvo el elemento n-1 de su cdr.

Y, por último, formulamos el caso base de la recursión, el problema más sencillo que se puede resolver directamente, sin hacer una llamada recursiva:

Para devolver el elemento que está en la posición 0 de una lista,  
 devuelvo el `car` de la lista.

La implementación de todo esto en Scheme sería la siguiente:

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

### Función mi-list-tail

La función (mi-list-tail lista n) devuelve la lista resultante de quitar n elementos de la cabeza de la lista original:

```
(mi-list-tail '(1 2 3 4 5 6 7) 2) ; => (3 4 5 6 7)
```

Piensa en cómo se implementaría de forma recursiva. Esta vez vamos a mostrar directamente la implementación, sin dar explicaciones de cómo se ha llegado a ella:

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

### Función mi-append

Veamos ahora cómo podríamos implementar de forma recursiva la función **append** que une dos listas. La llamaremos **(mi-append lista1 lista2)**.

Por ejemplo:

```
(mi-append '(a b c) '(d e f)) ; => (a b c d e f)
```

Para resolver el problema de forma recursiva, debemos confiar en la recursión para que resuelva un problema más sencillo y después terminar de arreglar el resultado devuelto por la recursión.

En este caso, podemos pasarle a la recursión un problema más sencillo quitando el primer elemento de la primera lista (con la función **cdr**) y llamando a la recursión para que concatene esta lista más pequeña con la segunda. Confiamos en que la recursión funciona correctamente y nos devuelve la concatenación de ambas listas

```
(mi-append (cdr '(a b c)) '(d e f)) => (b c d e f)
```

Y añadiremos el primer elemento a la lista resultante usando un **cons**:

```
(mi-append '(a b c) '(d e f)) =
(cons 'a (mi-append '(b c) '(d e f))) =
(cons 'a '(b c d e f)) =
(a b c d e f)
```

En general:

```
(define (mi-append lista1 lista2)
  (cons (car lista1) (mi-append (cdr lista1) lista2)))
```

El caso base, el caso en el que la función puede devolver un valor directamente sin llamar a la recursión, es aquel en el que **lista1** es **null?**. En ese caso devolvemos **lista2**:

```
(mi-append '() '(a b c)) => '(a b c)
```

La formulación recursiva completa queda como sigue:

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

### Función mi-reverse

Veamos cómo implementar de forma recursiva la función **mi-reverse** que invierte una lista

```
(mi-reverse '(1 2 3 4 5 6)) ; => (6 5 4 3 2 1)
```

La idea es sencilla: llamamos a la recursión para hacer la inversa del **cdr** de la lista y añadimos el primer elemento a la lista resultante que devuelve ya invertida la llamada recursiva.

Podemos definir una función auxiliar (**añade-al-final dato lista**) que añade un dato al final de una lista usando **append**:

Veamos directamente su implementación, usando **mi-append** para añadir un elemento al final de la lista:

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

La función **mi-reverse** quedaría entonces como sigue:

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

### Función cuadrados-hasta

La función (**cuadrados-hasta x**) devuelve una lista con los cuadrados de los números hasta **x**:

Para construir una lista de los cuadrados hasta  $x$ :  
construyo la lista de los cuadrados hasta  $x-1$  y le añado el cuadrado de  $x$

El caso base de la recursión es el caso en el que  $x$  es 1, entonces devolvemos una lista formada por el 1.

En Scheme:

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

Ejemplo:

```
(cuadrados-hasta 10) ; => (100 81 64 49 36 25 16 9 4 1)
```

## Función **filtrapares**

Es muy habitual recorrer una lista y comprobar condiciones de sus elementos, construyendo una lista con los que cumplan una determinada condición.

Por ejemplo, la siguiente función **filtrapares** construye una lista con los números pares de la lista que le pasamos como parámetro:

```
(define (filtrapares lista)
  (cond
    ((null? lista) '())
    ((even? (car lista)) (cons (car lista)
                                (filtrapares (cdr lista))))
    (else (filtrapares (cdr lista)))))
```

En el caso general, llamamos de forma recursiva a la función para que filtre el **cdr** de la lista. Y le añadimos el primer elemento si es par.

Cada vez llamaremos a la recursión con una lista más pequeña, por lo que en el caso base tendremos que comprobar si la lista que recibimos. En ese caso devolvemos la lista vacía.

Ejemplo:

```
(filtrapares '(1 2 3 4 5 6)) ; => (2 4 6)
```

## Función primo?

El uso de listas es uno de los elementos fundamentales de la programación funcional.

Como ejemplo, vamos a ver cómo trabajar con listas para construir una función que calcula si un número es primo. La forma de hacerlo será calcular la lista de divisores del número y comprobar si su longitud es dos. En ese caso será primo.

Por ejemplo:

```
(divisores 8) ; => (1 2 4 8) longitud = 4, no primo
(divisores 9) ; => (1 3 9) longitud = 3, no primo
(divisores 11) ; => (1 11) longitud = 2, primo
```

Podemos definir entonces la función (**primo? x**) de la siguiente forma:

```
(define (primo? x)
  (= 2
    (length (divisores x))))
```

¿Cómo implementamos la función (**divisores x**) que nos devuelve la lista de los divisores de un número **x**. Vamos a construirla de la siguiente forma:

1. Creamos una lista de todos los números del 1 a x
2. Filtramos la lista para dejar los divisores de x

La función (**lista-hasta x**) devuelve una lista de números 1..x:

```
(define (lista-hasta x)
  (if (= x 0)
    '()
    (cons x (lista-hasta (- x 1)))))
```

Ejemplos:

```
(lista-hasta 2) ; => (1 2)
(lista-hasta 10) ; => (1 2 3 4 5 6 7 8 9 10)
```

Definimos la función (**divisor? x y**) que nos diga si x es divisor de y:

```
(define (divisor? x y)
  (= 0 (mod y x)))
```

Ejemplos:

```
(divisor 2 10) ; => #t
(divisor 3 10) ; => #f
```

Una vez que hemos definido La función **divisor?** podemos utilizarla para definir la función recursiva (**filtra-divisores lista x**) que devuelve una lista con los números de **lista** que son divisores de **x**:

```
(define (filtra-divisores lista x)
  (cond
    ((null? lista) '())
    ((divisor? (car lista) x) (cons (car lista)
                                      (filtra-divisores (cdr lista) x)))
    (else (filtra-divisores (cdr lista) x))))
```

Ya podemos implementar la función que devuelve los divisores de un número **x** generando los números hasta **x** y filtrando los divisores de ese número. Por ejemplo, para calcular los divisores de 10:

```
(filtra-divisores (1 2 3 4 5 6 7 8 9 10) 10) ; => (1 2 5 10)
```

Se puede implementar de una forma muy sencilla:

```
(define (divisores x)
  (filtra-divisores (lista-hasta x) x))
```

Y una vez definida esta función, ya puede funcionar correctamente la función **primo?**.

## Funciones con número variable de argumentos

Hemos visto algunas funciones primitivas de Scheme, como **+** o **max** que admiten un número variable de argumentos. ¿Podemos hacerlo también en funciones definidas por nosotros?

La respuesta es sí, utilizando lo que se denomina notación *dotted-tail* (punto-cola) para definir los parámetros de la función. En esta notación se coloca 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.

Por ejemplo, si tenemos la definición

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

podemos llamar a la función anterior con dos o más argumentos:

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

En la llamada, los parámetros **x** e **y** tomarán los valores 1 y 2. El parámetro **lista-args** tomará como valor una lista con los argumentos restantes **(3 4 5 6)**.

También es posible permitir que todos los argumentos sean opcionales no poniendo ningún argumento antes del punto::

```
(define (funcion-cualquier-numero-args . lista-args)
  <cuerpo>)
```

Si hacemos la llamada

```
(funcion-cualquier-numero-args 1 2 3 4 5 6)
```

el parámetro **lista-args** tomará como valor la lista **(1 2 3 4 5 6)**.

Veamos un sencillo ejemplo.

Podemos implementar una función **mi-suma** que tome al menos dos argumentos y después un número variable de argumentos y devuelva la suma de todos ellos. Es muy sencillo: recogemos todos los argumentos en la lista de argumentos variables y llamamos a la función **suma-lista** que suma una lista de números:

```
(define (mi-suma x y . lista-nums)
  (if (null? lista-nums)
    (+ x y)
    (+ x (+ y (suma-lista lista-nums)))))
```

## Función **apply**

Con la función **(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.

Por ejemplo, podemos definir la función **(suma-cuadrados x y)** de aridad 2 de la siguiente forma:

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

Para invocar a **suma-cuadrados** hay que pasarle 2 parámetros:

```
(suma-cuadrados 10 5) ; => 100
```

Podemos usar la función **apply** para aplicar esta función a los dos elementos de una lista:

```
(apply suma-cuadrados '(10 5)) ; => 100
```

Usando **apply** podemos definir funciones recursivas con número variable de argumentos.

Por ejemplo la función **suma-parejas** que suma un número variable de parejas:

```
(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)
```

Hay que hacer notar en que la llamada recursiva es necesario usar **apply** porque **(cdr parejas)** es una lista. No podemos invocar a **suma-parejas** pasando una lista como parámetro, sino que hay que pasarle todos los argumentos por separado (recibe un número variable de argumentos). Eso lo conseguimos hacer con **apply**.

## Funciones como tipos de datos de primera clase

Hemos visto que la característica fundamental de la programación funcional es la definición de funciones. Hemos visto también que no producen efectos laterales y no tienen estado. Una función toma unos datos como entrada y produce un resultado como salida.

Una de las características fundamentales de la programación funcional es considerar a las funciones como *objetos de primera clase*. Recordemos que un tipo de primera clase es aquel que:

1. Puede ser asignado a una variable
2. Puede ser pasado como argumento a una función
3. Puede ser devuelto como resultado de una invocación a una función
4. Puede ser parte de un tipo mayor

Vamos a ver que las funciones son ejemplos de todos los casos anteriores: vamos a poder crear funciones sin nombre y asignarlas a variables, pasárlas como parámetro de otras funciones, devolverlas como resultado de invocar a otra función y guardarlas en tipos de datos compuestos como listas.

La posibilidad de usar funciones como objetos de primera clase es una característica fundamental de los lenguajes funcionales. Es una característica de muchos lenguajes multi-paradigma con características

funcionales como [JavaScript](#), [Python](#), [Swift](#) o a partir de la versión 8 de Java, [Java 8](#), (donde se denominan *expresiones lambda*).

## Forma especial **lambda**

Vamos a empezar explicando la forma especial **lambda** de Scheme, que nos permite crear funciones anónimas en tiempo de ejecución.

De la misma forma que podemos usar cadenas o enteros sin darles un nombre, en Scheme es posible usar una función sin darle un nombre mediante esta forma especial.

### Sintaxis de la forma especial **lambda**

La sintaxis de la forma especial **lambda** es:

```
(lambda (<arg1> ... <argn>)
      <cuerpo>)
```

El cuerpo del lambda define un *bloque de código* y sus argumentos son los parámetros necesarios para ejecutar ese bloque de código. Llamamos a la función resultante una *función anónima*.

Algunos ejemplos:

Una función anónima que suma dos parejas:

```
(lambda (p1 p2)
      (cons (+ (car p1) (car p2))
            (+ (cdr p1) (cdr p2))))
```

Una función anónima que devuelve el mayor de dos números:

```
(lambda (a b)
      (if (> a b)
          a
          b))
```

### Semántica de la forma especial **lambda**

La invocación a la forma especial **lambda** construye una función anónima en tiempo de ejecución.

Por ejemplo, si ejecutamos una expresión lambda en el intérprete veremos que devuelve un procedimiento:

```
(lambda (x) (* x x)) ; => #<procedure>
```

El procedimiento construido es un bloque de código que devuelve el cuadrado de un número.

¿Qué podemos hacer con este procedimiento?

Podemos asignarlo a un identificador. Por ejemplo, en la siguiente expresión, primero se evalúa la *expresión lambda* y el procedimiento resultante se asocia al identificador **f**.

```
(define f (lambda (x) (* x x)))
```

El ejemplo anterior funciona de una forma idéntica al siguiente:

```
(define x (+ 2 3))
```

En ambos casos se evalúa la expresión derecha y el resultado se guarda en un identificador. En el primer caso la expresión que se evalúa devuelve un procedimiento, que se guarda en la variable **f** y en el segundo un número, que se guarda en la variable **x**.

Si escribimos los identificadores **f** y **x** en el intérprete Scheme los evalúa y muestra los valores guardados:

```
f ; => #<procedure:f>
x ; => 5
```

En el primer caso se devuelve un procedimiento y en el segundo un número. Fíjate que Scheme trata a los procedimientos y a los números de la misma forma; son lo que se denominan datos de primera clase.

Una vez asignado un procedimiento a un identificador, lo podemos utilizar como de la misma forma que invocamos habitualmente a una función:

```
(f 3) ; => 9
```

No es necesario un identificador para invocar a una función; podemos crear la función con una expresión lambda e invocar a la función anónima recién creada:

```
((lambda (x) (* x x)) 3) ; => 9
```

La llamada a **lambda** crea un procedimiento y el paréntesis a su izquierda lo invoca con el parámetro 3:

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

Es importante remarcar que con **lambda** estamos creando una función en *tiempo de ejecución*. Es código que creamos para su posterior invocación.

Cada lenguaje de programación tiene su sintaxis propia de expresiones lambda. Por ejemplo, las siguientes expresiones crean una función que devuelve el cuadrado de un número:

### Java 8

```
Integer x -> {x*x}
```

### Scala

```
(x:Int) => {x*x}
```

### Objective C

```
^int (int x)
{
    x*x
};
```

### Swift

```
{ (x: Int) -> Int in return x*x }
```

### Identificadores y funciones

Tras conocer **lambda** ya podemos explicarnos por qué cuando escribimos en el intérprete de Scheme el nombre de cualquier función, se evalúa a un *procedure*:

```
+ ; => <procedure:+>
append ; => #<procedure:append>
```

El identificador se evalúa y devuelve el *objeto función* al que está ligado. En Scheme los nombres de las funciones son realmente símbolos a los que están ligados *objetos de tipo función*.

Podemos comprobar también de esta manera que **and** y **or** no son funciones. Si escribimos **and** o **or** e intentamos evaluar cualquiera de los dos símbolos, veremos que Scheme devuelve un error:

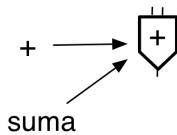
```
and
; and: bad syntax in: and
```

```
or
; or: bad syntax in: or
```

Podemos asignar funciones ya existentes a nuevos identificadores usando **define**, como en el ejemplo siguiente:

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

Es muy importante darse cuenta que la expresión (**define suma +**) se evalúa de forma idéntica a (**define y x**). Primero se evalúa el identificador **+**, que devuelve el *objeto función* **suma**, que se asigna a la variable **suma**. El resultado final es que tanto **+** como **suma** tienen como valor el mismo procedimiento:



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

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

siempre se convierte internamente en:

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

Por ejemplo

```
(define (cuadrado x)
  (* x x))
```

es equivalente a:

```
(define cuadrado
  (lambda (x) (* x x)))
```

**Predicado procedure?**

Podemos comprobar si algo es una función utilizando el predicado de Scheme **procedure?**.

Por ejemplo:

```
(procedure? (lambda (x) (* x x))) ; => #t
(define suma +)
(procedure? suma) ; => #t
(procedure? '+) ; => #f
```

Hemos visto que las funciones pueden asignarse a variables. También cumplen las otras condiciones necesarias para ser consideradas objetos de primera clase.

## Funciones argumentos de otras funciones

Hemos visto ya un ejemplo de cómo pasar una función como parámetro de otra. Veamos algún otro.

Por ejemplo, podemos definir la función **aplica** que recibe una función en el parámetro **func** y dos valores en los parámetros **x** e **y** y devuelve el resultado de invocar a la función que pasamos como parámetro con **x** e **y**. La función que se pase como parámetro debe tener dos argumentos

Para realizar la invocación a la función que se pasa como parámetro basta con usar **func** como su nombre. La función se ha ligado al nombre **func** en el momento de la invocación a **aplica**, de la misma forma que los argumentos se ligan a los parámetros **x** e **y**:

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

Algunos ejemplos de invocación, usando funciones primitivas, funciones definidas y expresiones lambda:

```
(aplica + 2 3) ; => 5
(aplica * 4 5) ; => 10
(aplica string-append "hola" "adios") ; => "holaadios"

(define (string-append-con-guion s1 s2)
  (string-append s1 "-" s2))

(aplica string-append-con-guion "hola" "adios") ; => "hola-adios"

(aplica (lambda (x y) (sqrt (+ (* x x) (* y y)))) 3 4) ; => 5
```

Otro ejemplo, la función **aplica-2** que toma dos funciones **f** y **g** y un argumento **x** y devuelve el resultado de aplicar **f** a lo que devuelve la invocación de **g** con **x**:

```
(define (aplica-2 f g x)
  (f (g x)))
```

```
(define (suma-5 x)
  (+ x 5))
(define (doble x)
  (+ x x))
(aplica-2 suma-5 doble 3) ; => 11
```

## Generalización

La posibilidad de pasar funciones como parámetros de otras es una poderosa herramienta de abstracción. Nos va a permitir diseñar funciones más genéricas.

Veamos un ejemplo. Supongamos que queremos calcular el sumatorio de **a** hasta **b**:

```
(define (sum-x a b)
  (if (> a b)
    0
    (+ a (sum-x (+ a 1) b))))
(sum-x 1 10) ; => 55
```

Supongamos ahora que queremos calcular el sumatorio de **a** hasta **b** sumando los números al cuadrado:

```
(define (sum-cuadrado-x a b)
  (if (> a b)
    0
    (+ (* a a) (sum-cuadrado-x (+ a 1) b))))
(sum-cuadrado-x 1 10) ; => 385
```

Y el sumatorio de **a** hasta **b** sumando los cubos:

```
(define (sum-cubo-x a b)
  (if (> a b)
    0
    (+ (* a a a) (sum-cubo-x (+ a 1) b))))
(sum-cubo-x 1 10) ; => 3025
```

Vemos que el código de las tres funciones anteriores es muy similar, cada función la podemos obtener haciendo un *copy-paste* de otra previa. Lo único que cambia es la función a aplicar a cada número de la serie.

Siempre que hagamos *copy-paste* al programar tenemos que empezar a sospechar que no estamos generalizando suficientemente el código. Un *copy-paste* arrastra también *bugs* y obliga a realizar múltiples modificaciones del código cuando en el futuro tengamos que cambiar cosas.

La posibilidad de pasar una función como parámetro viene a nuestra ayuda para generalizar el código anterior. En este caso, lo único que cambia en las tres funciones anteriores es la función a aplicar a los números de la serie. En primer caso no se hace nada, en el segundo se eleva al cuadrado y en el tercer caso al cubo.

Podemos tomar esa función como un parámetro adicional y definir una función genérica `sum-f-x` que generaliza las tres funciones anteriores. Tendríamos 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))))
```

Las funciones anteriores son casos particulares de esta función que las generaliza. Por ejemplo, para calcular el sumatorio desde 1 hasta 10 de `x` al cubo:

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

(sum-f-x cubo 1 10) ; => 3025
```

También podemos utilizar una expresión lambda en la invocación a `sum-f` que construya la función que queremos aplicar a cada número. Por ejemplo, podemos sumar la expresión  $(n/(n-1))$  para todos los números del 2 al 100:

```
(sum-f-x (lambda (n) (/ n (- n 1))) 2 100)
```

Veremos más adelante muchos más ejemplos de funciones pasadas como parámetros y de la generalidad que permite este patrón, cuando estudiemos las funciones de orden superior.

## Funciones que devuelven funciones

Cualquier objeto de primera clase puede ser devuelto por una función; enteros, booleanos, parejas, etc. son objetos primitivos y podemos definir funciones que los devuelven.

En el paradigma funcional lo mismo sucede con las funciones. Podemos definir una función que cuando se invoque construya otra función y la devuelva como resultado.

Esta es una de las características más importantes que diferencia los lenguajes de programación funcionales de otros que no lo son. En lenguajes como C, C++ o Java (antes de Java 8) no es posible hacer esto.

Para devolver una función en Scheme debemos usar la forma especial `lambda` en el cuerpo de una función. Así, cuando se invoca a esta función se evalúa `lambda` y se devuelve la función resultante. Es una función que creamos en tiempo de ejecución, durante la evaluación de la función principal.

La función que se devuelve se denomina **clausura** ([Wikipedia](#)). Y decimos que la función que ha construido la clausura es una **función constructora**.

## Función sumador

Vamos a empezar con un ejemplo muy sencillo. Definimos una función constructora que crea en su ejecución una función que suma **k** a un número:

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

El cuerpo de la función (**construye-sumador k**) está formado por una expresión lambda. Cuando se invoca a **construye-sumador** se evalúa esta expresión lambda y se devuelve el procedimiento creado.

En este caso se construye otra función de 1 argumento que suma **k** al argumento.

Por ejemplo, podemos invocar a **construye-sumador** pasando 10 como parámetro:

```
(construye-sumador 10) ; ⇒ #<procedure>
```

Como hemos dicho, se devuelve un procedimiento, una función. Esta función devuelta debe invocarse con un argumento y devolverá el resultado de sumar 10 a ese argumento:

```
(define f (construye-sumador 10))
(f 3) ; ⇒ 13
```

También podemos invocar directamente a la función que devuelve la función constructora, sin guardarla en una variable:

```
((construye-sumador 10) 3) ; ⇒ 13
```

Dependiendo del parámetro que le pasemos a la función constructora obtendremos una función sumadora que sume un número u otro. Por ejemplo para obtener una función sumadora que suma 100:

```
(define g (construye-sumador 100))
(g 3) ; ⇒ 103
```

¿Cómo funciona la clausura? ¿Por qué la invocación a (**g 3**) devuelve 103?

Aquí hay que apartarse bastante del modelo de evaluación de sustitución que hemos visto y utilizar un nuevo modelo en el que se tiene en cuenta los ámbitos de las variables.

No vamos a explicar en detalle este modelo, pero sí dar unas breves pinceladas.

Recordemos la definición de **construye-sumador**:

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

Y supongamos que realizamos las siguientes invocaciones:

```
(define g (construye-sumador 100))
(g 3) ; ⇒ 103
```

Podemos explicar lo que sucede en la evaluación de estas funciones de la siguiente forma:

- Cuando invocamos a **construye-sumador** con un valor concreto para **k** (por ejemplo 100), queda vinculado el valor de 100 al parámetro **k** en el ámbito local de la función.
- En este ámbito local la expresión lambda crea una función. Esta función creada en el ámbito local **captura** este ámbito local, con sus variables y sus valores (en este caso la variable **k** y su valor 100).
- Cuando se invoca a la función desde fuera (cuando llamamos a **g** en el ejemplo) se ejecuta el cuerpo de la función **(+ x k)** con **x** valiendo el parámetro (3) y el valor de **k** se obtiene del ámbito capturado (100).

El hecho de que función creada en el ámbito local capture este ámbito es lo que hace que se denomine una **clausura** (del inglés **closure**). La función *se cierra* sobre el ámbito capturado y puede utilizar sus variables.

### Función **composicion**

Otro ejemplo de una función que devuelve otra función es la función siguiente (**composicion f g**) que recibe dos funciones de un argumento y devuelve otra función que realiza la composición de ambas:

```
(define (composicion f g)
  (lambda (x)
    (f (g x))))
```

La función devuelta invoca primero a **g** y el resultado se lo pasa a **f**. Veamos un ejemplo. Supongamos que tenemos definidas la función **cuadrado** y **doble** que calculan el cuadrado y el doble de un número respectivamente. Podremos entonces llamar a **composicion** con esas dos funciones para construir otra función que primero calcule el cuadrado y después el doble de una número:

```
(define h (composicion doble cuadrado))
```

La variable `h` contiene la función devuelta por `composicion`. Una función de un argumento que devuelve el doble del cuadrado de un número:

```
(h 4) ; => 32
```

## Función construye-segura

Vamos a ver un último ejemplo en el que definimos una función constructora que extiende funciones ya existentes.

Recordemos la función `lista-hasta`:

```
(define (lista-hasta x)
  (if (= x 0)
      '()
      (cons x (lista-hasta (- x 1)))))
```

Un problema de la función anterior es que si le pasamos un número negativo entra en un bucle infinito.

Definimos la función `(construye-segura condicion f)` que recibe dos funciones: un predicado y otra función, ambos de 1 argumento. Devuelve otra función en la que sólo se llamará a `f` si el argumento cumple la `condicion`.

```
(define (construye-segura condicion f)
  (lambda (x)
    (if (condicion x)
        (f x)
        'error)))
```

La función construye una función anónima de un argumento `x` (igual que `f`) en cuyo cuerpo se comprueba si el argumento cumple la condición y sólo en ese caso se llama a `f`.

Podemos entonces construir una función segura a partir de la función `lista-hasta` en la que se devuelva `error` si el argumento es un número negativo:

```
(define lista-hasta-segura
  (construye-segura (lambda (x) (>= x 0)) lista-hasta))
(lista-hasta-segura 8) ; => (8 7 6 5 4 3 2 1)
(lista-hasta-segura -1) ; => error
```

Podríamos usar `construye-segura` con cualquier función de 1 argumento que queramos hacer segura. Por ejemplo, la función `sqrt`:

```
(define sqrt-segura (construye-segura (lambda (x) (>= x 0) sqrt))
(sqrt-segura 100) ; => 10
(sqrt-segura -100) ; => error
```

La potencia de las funciones constructoras viene del hecho de que es posible crear nuevas funciones en tiempo de ejecución. No es necesario conocer las condiciones y las características de estas nuevas funciones a priori, cuando estamos compilando nuestro programa. Sino que pueden depender de datos obtenidos del usuario o de otros módulos del programa en tiempo de ejecución.

Por ejemplo, la condición de **construye-segura** podría contener valores obtenidos en tiempo de ejecución, de forma que solo se llamaría a la función que queremos hacer segura si el número está en un rango definido esos valores:

```
(construye-segura (lambda (x) (and (>= x limite-inf)
(<= x limite-sup))) f))
```

## Funciones en estructuras de datos

La última característica de los tipos de primera clase es que pueden formar parte de tipos de datos compuestos, como listas.

Para construir una lista de funciones debemos llamar a **list** con las funciones:

```
(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>)
```

También podemos definir las funciones con expresiones lambda. Por ejemplo, podemos añadir a la lista una función que suma 5 a un número:

```
(define lista2 (cons (lambda (x) (+ x 5)) lista))
lista2
; => (#<procedure> #<procedure:cuadrado> #<procedure:suma-1> #<procedure:doble>)
```

Una vez creada una lista con funciones, ¿cómo podemos invocar a alguna de ellas?. Debemos tratarlas de la misma forma que tratamos cualquier otro dato guardado en la lista, las recuperamos con las funciones **car** o **list-ref** y las invocamos.

Por ejemplo, para invocar a la primera función de **lista2**:

```
((car lista2) 10) ; => 15
```

O a la tercera:

```
((list-ref lista2 2) 10) ; => 11
```

## Funciones que trabajan con listas de funciones

Veamos un ejemplo de una función (`aplica-funcs lista-funcs x`) que recibe una lista de funciones en el parámetro `lista-funcs` y las aplica todas **de derecha a izquierda** al número que pasamos en el parámetro `x`.

Por ejemplo, supongamos la lista anterior, que contiene las funciones `cuadrado`, `cubo` y `suma-1`:

```
(define lista (list cuadrado cubo suma-1))
```

la llamada a (`aplica-funcs lista 5`) debería devolver el resultado de aplicar primero `suma-1` a 5, después `cubo` al resultado y después `cuadrado`:

```
(cuadrado (cubo (suma-1 5))) ; => 46656
```

Para implementar `aplica-funcs` tenemos que usar una recursión. Si vemos el ejemplo, podemos comprobar que es sencillo definir el caso general:

```
(aplica-funcs (cuadrado cubo suma-1) 5) =
(cuadrado (aplica-funcs (cubo suma-1) 5)) =
(cuadrado 216) = 46656
```

El caso general de la recursión de la función `aplica-funcs` se define entonces como:

```
(define (aplica-funcs lista-funcs x)
  ; falta el caso base
  ((car lista-funcs) (aplica-funcs (cdr lista-funcs) x)))
```

El caso base sería en el que la lista de funciones es vacía, en cuyo caso se devuelve el propio número:

```
(if (null? lista-funcs) ; la lista de funciones está vacía
    x ; devolvemos el propio número
    ...)
```

La implementación completa es:

```
(define (aplica-funcs lista-funcs x)
  (if (null? lista-funcs)
      x
      ((car lista-funcs)
       (aplica-funcs (cdr lista-funcs) x))))
```

Un ejemplo de uso:

```
(define lista-funcs (list (lambda (x) (* x x))
                           (lambda (x) (* x x x))
                           (lambda (x) (+ x 1))))
(aplica-funcs lista-funcs 5) ; => 46656
```

## Funciones de orden superior

Llamamos funciones de orden superior (*higher order functions* en inglés) a las funciones que toman otras como parámetro o devuelven otra función. Permiten generalizar soluciones con un alto grado de abstracción.

Ya hemos visto algunas funciones de orden superior, unas construidas por nosotros y otras propias de Scheme, como **apply**.

Además de **apply**, los lenguajes de programación funcional como Scheme, Scala o Java 8 tienen ya predefinidas algunas otras funciones de orden superior que trabajan con listas. Estas funciones permiten definir operaciones sobre las listas de una forma muy concisa y compacta. Son muy usadas, porque también se pueden utilizar sobre *streams* de datos obtenidos en operaciones de entrada/salida (por ejemplo, datos JSON resultantes de una petición HTTP).

Vamos a ver las funciones más importantes, su uso y su implementación.

- **map**
- **filter**
- **exists?**
- **for-all?**
- **foldr** y **foldl**

Después de explicar estas funciones terminaremos con un ejemplo de su aplicación en el que comprobaremos cómo la utilización de funciones de orden superior es una excelente herramienta de la programación funcional que permite hacer código muy conciso y expresivo.

La combinación de funciones de nivel superior con listas es una de las características más potentes de la programación funcional.

### Función **map**

Comenzamos con la función `map`. La palabra `map` viene del inglés `mapping` o transformación. Se trata de una función que **transforma** una lista aplicando a todos sus elementos una función de transformación que se pasa como parámetro.

En concreto, la función recibe otra función y una lista:

```
(map transforma lista) -> lista
```

Y devuelve la lista resultante de aplicar la función a todos los elementos de la lista.

La función de transformación recibe como argumentos elementos de la lista y devuelve el resultado de transformar ese elemento.

```
(transforma elemento) -> elemento
```

Por ejemplo:

```
(map cuadrado '(1 2 3 4 5)) ; => (1 4 9 16 25)
```

La lista resultante es el resultado de construir una lista nueva aplicando la función `cuadrado` a todos los elementos de la lista original.

La función de transformación debe ser compatible con los elementos de la lista original. Por ejemplo, si la lista es una lista de parejas, la función de transformación debe recibir una pareja. Veamos un ejemplo de este caso, en el que a partir de una lista de parejas obtenemos una lista con las sumas de cada pareja:

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

(map suma-pareja '((2 . 4) (3 . 6) (5 . 3))) ; => (6 9 8)
```

También podríamos hacerlo con una expresión lambda:

```
(map (lambda (pareja)
            (+ (car pareja) (cdr pareja))) '((2 . 4) (3 . 6) (5 . 3)))
; => (6 9 8)
```

Un último ejemplo, en el que usamos `map` para transformar una lista de símbolos en una lista con sus longitudes:

```
(map (lambda (s)
    (string-length (symbol->string s))) '(Esta es una lista de símbolos))
; => (4 2 3 5 2 8)
```

### Implementación de map

¿Cómo se podría implementar `map` de forma recursiva? Definimos la función `mi-map`. La implementación es la siguiente:

```
(define (mi-map f lista)
  (if (null? lista)
      '()
      (cons (f (car lista))
            (mi-map f (cdr lista)))))
```

### Función map con más de una lista

La función `map` puede recibir un número variable de listas, todas ellas de la misma longitud:

```
(map transforma lista_1 ... lista_n) -> lista
```

En este caso la función de transforma debe recibir tantos argumentos como listas recibe `map`:

```
(transforma dato_1 ... dato_n) -> dato
```

La función `map` aplica `transforma` a los elementos cogidos de las n listas y construye así la lista resultante.

Ejemplos:

```
(map + '(1 2 3) '(10 20 30)) ; => (11 22 33)
(map cons '(1 2 3) '(10 20 30)) ; => ((1 . 10) (2 . 20) (3 . 30))
(map > '(12 3 40) '(20 0 10)) ; => (#f #t #t)

(define (mayor a b) (if (> a b) a b))
(define (mayor-de-tres a b c)
  (mayor a (mayor b c)))

(map mayor-de-tres '(10 2 20 -1 34)
      '(2 3 12 89 0)
      '(100 -10 23 45 8))
; => (100 3 23 89 34)
```

!!! Tip "Consejo" La función `map` recibe una lista de  $n$  elementos y devuelve otra de  $n$  elementos transformados.

## Función `filter`

Veamos otra función de orden superior que trabaja sobre listas.

La función (`filter predicado lista`) toma como parámetro un predicado y una lista y devuelve como resultado los elementos de la lista que cumplen el predicado.

```
(filter predicado lista) -> lista
```

La función (`predicado elem`) que usa `filter` recibe elementos de la lista y devuelve `#t` o `#f`.

```
(predicado elem) -> boolean
```

Un ejemplo de uso:

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

Otro ejemplo: supongamos que queremos filtrar una lista de parejas de números, devolviendo aquellas que parejas que cumplen que su parte izquierda es mayor o igual que la derecha. Lo podríamos hacer con la siguiente expresión:

```
(filter (lambda (pareja)
    (>= (car pareja) (cdr pareja)))
    '((10 . 4) (2 . 4) (8 . 8) (10 . 20)))
; => ((10 . 4) (8 . 8))
```

Y un último ejemplo: filtramos todos los símbolos con longitud menor de 4.

```
(filter (lambda (s)
    (>= (string-length (symbol->string s)) 4))
    '(Esta es una lista de símbolos))
; => (Esta lista símbolos)
```

!!! Tip "Consejo" La función `filter` recibe una lista de  $n$  elementos y devuelve otra de con  $n$  o menos elementos originales filtrados por una condición.

## Implementación de `filter`

Podemos implementar la función **filter** de forma recursiva:

```
(define (mi-filter pred lista)
  (cond
    ((null? lista) '())
    ((pred (car lista)) (cons (car lista)
                               (mi-filter pred (cdr lista))))
    (else (mi-filter pred (cdr lista)))))
```

### Función **exists?**

La función de orden superior **exists?** recibe un predicado y una lista y comprueba si algún elemento de la lista cumple ese predicado.

```
(exists? predicado lista) -> boolean
```

Igual que en **filter** el **predicado** recibe elementos de la lista y devuelve **#t** o **#f**.

```
(predicado elem) -> boolean
```

La función **exists?** no está definida con este nombre en Racket, aunque sí en Scheme. En Racket se llama **ormap**.

Ejemplo de uso:

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

La implementación recursiva de **exists?** es la siguiente:

```
(define (exists? predicado lista)
  (if (null? lista)
      #f
      (or (predicado (car lista))
          (exists? predicado (cdr lista)))))
```

### Función **for-all?**

La función de orden superior **for-all?** recibe un predicado y una lista y comprueba que todos los elementos de la lista cumplen ese predicado.

La función tampoco está definida con este nombre en Racket, aunque sí en Scheme. En Racket existe una función equivalente que se llama **andmap**.

Ejemplo de uso:

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

La implementación recursiva de **for-all?** es la siguiente:

```
(define (for-all? predicado lista)
  (or (null? lista)
      (and (predicado (car lista))
            (for-all? predicado (cdr lista)))))
```

La llamada recursiva comprueba que todos los elementos del resto de la lista cumplen el predicado y también lo debe cumplir el primer elemento. Una lista vacía cumple siempre devuelve **#t** (al no tener elementos, podemos decir que todos sus elementos cumplen el predicado).

## Función **foldr**

Veamos ahora la función (**foldr combina base lista**) que permite recorrer una lista aplicando una función binaria de forma acumulativa a sus elementos y devolviendo un único valor como resultado.

```
(foldr combina base lista) -> valor
```

El nombre **fold** significa *plegado*, indicando que la lista a la que se aplica se va "plegando" y al final se devuelve un único resultado. El plegado lo realiza 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**).

```
(combina dato resultado) -> resultado
```

La función **combina** se aplica a los elementos de la lista **de derecha a izquierda**, empezando por el último elemento de la lista y el valor inicial **base** y aplicándose sucesivamente a los resultados que se van obteniendo.

Veamos un ejemplo. Supongamos que la función de plegado es una función que suma el dato que viene de la lista con el valor acumulado:

```
(define (suma dato resultado)
  (+ dato resultado))
```

Llamamos a los parámetros **dato** y **resultado** para remarcar que el primer parámetro se va a coger de la lista y el segundo del resultado calculado.

Veamos qué pasa cuando hacemos un **foldr** con esta función suma y la lista '(1 2 3) y con el número 0 como base:

```
(foldr suma 0 '(1 2 3)) ; => 6
```

La función **suma** se va a ir aplicando a todos los elementos de la lista de **derecha a izquierda**, empezando por el valor base (0) y el último elemento de la lista (3) y cogiendo el resultado obtenido y utilizándolo como nuevo parámetro **resultado** en la siguiente llamada.

En concreto, la secuencia de llamadas a la función **suma** serán las siguientes:

```
(suma 3 0) ; => 3
(suma 2 3) ; => 5
(suma 1 5) ; => 6
```

Otro ejemplo de uso:

```
(foldr string-append "****" '("hola" "que" "tal")) ; => "holaquetal****"
```

En este caso la secuencia de llamadas a **string-append** que se van a producir son:

```
(string-append "tal" "****") ; => "tal****"
(string-append "que" "tal****") ; => "quetal****"
(string-append "hola" "quetal****") ; => "holaquetal****"
```

Otros ejemplos:

```
(foldr (lambda (x y) (* x y)) 1 '(1 2 3 4 5 6 7 8)) ; => 40320
(foldr cons '() '(1 2 3 4)) ; => (1 2 3 4)
```

Un último ejemplo:

```
(define (suma-parejas lista-parejas)
  (foldr (lambda (pareja resultado)
            (+ (car pareja) (cdr pareja) resultado)) 0 lista-parejas))

(suma-parejas (list (cons 3 6) (cons 2 9) (cons -1 8) (cons 9 3))) ; => 39
```

### Implementación de foldr

Podemos implementar de forma recursiva la función **foldr**:

```
(define (mi-foldr func base lista)
  (if (null? lista)
      base
      (func (car lista) (mi-foldr func base (cdr lista)))))
```

### Función foldl

La función (**foldl combina base lista**) (*fold left*) es similar a **foldr** con la diferencia de que la secuencia de aplicaciones de la función de plegado se hace **de izquierda a derecha** en lugar de derecha a izquierda.

El perfil de la función de plegado es el mismo que en **foldr**:

```
(func dato resultado) -> resultado
```

Por ejemplo, si la función de combinación es **string-append**:

```
(foldl string-append "****" '("hola" "que" "tal"))
; => "talquehola****"
```

La secuencia de llamadas a **string-append** es:

```
(string-append "hola" "****") ; => "hola****"
(string-append "que" "hola****") ; => "quehola****"
(string-append "tal" "quehola****") ; => "talquehola****"
```

Otro ejemplo:

```
(foldl cons '() '(1 2 3 4)) ; => (4 3 2 1)
```

La implementación de `foldl` la veremos cuando hablamos de recursión por la cola (*tail recursion*) en el próximo tema.

!!! Tip "Consejo" Las funciones `foldr` o `foldl` reciben una lista de datos y devuelven un único resultado.

### Uso de `and` y `or` con FOS

Hemos visto que las primitivas `and` y `or` no son funciones, sino formas especiales. Debido a esto, no podemos usarlas como funciones que se pasan a otra función de orden superior.

Por ejemplo, la siguiente expresión es incorrecta:

```
(foldr and #t '(#t #f #f))
; and: bad syntax in: and
```

Para comprobar expresiones booleanas en una lista podemos usar `foldr` con una expresión lambda:

```
(foldr (lambda (dato result)
             (and dato result)) #t '(#t #f #f))
; => #f
```

O, mejor aún, es posible usar `for-all?` o `exists?` (o las funciones equivalentes de Racket `andmap` o `ormap`).

Por ejemplo, para comprobar si algún booleano de una lista es `#t` podríamos hacer:

```
(exists? (lambda (x) x) '(#f #f #t #f)) ; => #t
(ormap (lambda (x) x) '(#f #f #t #f)) ; => #t
```

### Funciones recursivas con FOS y expresiones lambda

El uso de funciones de orden superior (FOS) y expresiones lambda proporciona muchísima expresividad en un lenguaje de programación. Es posible escribir código muy conciso y construir funciones recursivas que trabajan sobre listas sin usar la recursividad de forma explícita.

#### Función `(suma-n n lista)`

Supongamos que queremos definir una función `(suma-n n lista)` que devuelve la lista resultante el resultado de sumar un número `n` a todos los elementos de una lista.

Podemos hacerlo de forma recursiva:

```
(define (suma-n n lista)
  (if (null? lista)
      '()
```

```
(cons (+ (car lista) n)
      (suma-n n (cdr lista))))
```

Funciona de la siguiente manera:

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

### Implementación con map

Pero si utilizamos funciones de orden superior, podemos implementar la misma función de una forma mucho más concisa y expresiva.

Lo podemos hacer utilizando la función de orden superior `map` y una expresión lambda que sume el número `n` a los elementos de la lista:

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

Vemos que utilizamos el parámetro `n` en el cuerpo de la expresión lambda. De esta forma la función que se aplica a los elementos de la lista es una función que suma este número a cada elemento. La variable `x` en el parámetro de la expresión lambda es la que va tomando el valor de los elementos de la lista.

```
(suma-n 10 '(1 2 3 4) 10) =>
(map #<procedure-que-suma-10-a-x> (1 2 3 4)) = (11 12 13 14)
```

### Composición de funciones de orden superior

Dado que muchas de 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.

Por ejemplo, podemos 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.

Lo podríamos hacer reutilizando el código del ejemplo anterior, y añadiendo una llamada a `foldr` para que haga la suma:

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

Funcionaría de la siguiente forma:

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

Otro ejemplo. Supongamos que tenemos una lista de parejas de números y queremos contar aquellas parejas cuya suma de ambos números es mayor que un umbral (por ejemplo, 10).

```
(define lista-parejas (list (cons 1 2)
                               (cons 3 8)
                               (cons 2 3)
                               (cons 9 6)))
(cuenta-mayores-que 10 lista-parejas) ; => 2
```

Se podría implementar de una forma muy concisa componiendo una llamada a `map` para realizar la suma de cada pareja junto con una llamada a `filter` que compruebe que el resultado sea mayor de `n`. Y al final llamamos a `length` para contar la longitud de la lista resultante:

```
(define (cuenta-mayores-que n lista-parejas)
  (length
    (filter (lambda (x)
              (> x n)) (map (lambda (pareja)
                                (+ (car pareja) (cdr pareja))) lista-parejas))))
```

#### Función (`contienen-letra carácter lista-pal`)

Veamos otro ejemplo. Supongamos que queremos definir la función (`contienen-letra carácter lista-pal`) que devuelve las palabras de una lista que contienen un determinado carácter.

Por ejemplo:

```
(contienen-letra #\a '("En" "un" "lugar" "de" "la" "Mancha"))
; => ("lugar" "la" "Mancha")
```

Podemos implementar `contienen-letra` usando la función de orden superior `filter`, con una expresión lambda que se aplicará a cada una de las palabras de la lista para comprobar si la palabra contiene el carácter:

```
(define (contienen-letra carácter lista-pal)
  (filter (lambda (pal)
            (letra-en-pal? carácter pal)) lista-pal))
```

El parámetro `pal` de la expresión lambda irá cogiendo el valor de todas las palabras de `lista-pal` y la función (`letra-en-pal? carácter pal`) comprobará si la cadena contiene el carácter.

La función (`letra-en-pal? carácter pal`) es una función auxiliar que tenemos que implementar.

Por ejemplo:

```
(letra-en-pal? #\a "Hola") ; => #t
(letra-en-pal? #\a "Pepe") ; => #f
```

La podemos implementar de una forma muy elegante obteniendo una lista de caracteres a partir de la cadena y usando la función de orden superior `exists?`:

```
(define (letra-en-pal? caracter palabra)
  (exists? (lambda (c)
              (equal? c caracter)) (string->list palabra)))
```

### Función divisores

Un último ejemplo en el que implementamos la función `(divisores n)` utilizando una función de orden superior.

Suponemos que tenemos definidas las funciones `(numeros-hasta n)` y `(divisor? x n)`:

```
(define (numeros-hasta n)
  (if (= 0 n)
      '()
      (cons n (numeros-hasta (- n 1)))))

(define (divisor? x n)
  (= 0 (mod n x)))
```

Entonces la función `(divisores n)` se implementaría de la siguiente forma:

```
(define (divisores n)
  (filter (lambda (x)
              (divisor? x n)) (numeros-hasta n)))
```

## Bibliografía - SICP

En este tema hemos explicado conceptos de los siguientes capítulos del libro *Structure and Interpretation of Computer Programs*:

- [1.1.1 - Expressions](#)
- [1.1.2 - Naming and environment](#)
- [1.1.3 - Evaluating combinations](#)
- [1.1.4 - Compound procedures](#)
- [1.1.5 - The Substitution Model for Procedure Application](#)

- [1.1.6 - Conditional Expressions and Predicates](#)
  - [1.3 - Formulating Abstractions with Higher-Order Procedures](#)
  - [2.2 - Hierarchical Data and the Closure Property \(Introducción de la sección\)](#)
  - [2.2.1 - Representing Sequences](#)
  - [2.3.1 - Quotation](#)
- 

Lenguajes y Paradigmas de Programación, curso 2019-20

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante  
Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez

## Tema 3: Procedimientos recursivos

---

Ya hemos visto muchos ejemplos de funciones recursivas. Una función es recursiva cuando se llama a si misma. Una vez que uno se acostumbra a su uso, se comprueba que la recursión es una forma mucho más natural que la iteración de expresar un gran número de funciones y procedimientos.

La formulación matemática de la recursión es sencilla de entender, pero su implementación en un lenguaje de programación no lo es tanto. El primer lenguaje de programación que permitió el uso de expresiones recursivas fue el Lisp. En el momento de su creación existía ya el Fortran, que no permitía que una función se llamase a si misma.

Ya hemos visto la utilidad de la recursión en muchos ejemplos para recorrer listas, para filtrarlas, etc. En este tema veremos algunos aspectos negativos de la recursión: su coste espacial y temporal. Veremos que hay soluciones a estos problemas, cambiando el estilo de la recursión y generando *procesos iterativos* o usando un enfoque automático llamado *memoization* en el que se guardan los resultados de cada llamada recursiva. Por último, veremos un último ejemplo curioso e interesante de la recursión para realizar figuras fractales con gráficos de tortuga.

### El coste de la recursión

Hasta ahora hemos estudiado el diseño de funciones recursivas. Vamos a tratar por primera vez su coste. Veremos que hay casos en los que es prohibitivo utilizar la recursión tal y como la hemos visto. Y veremos también que existen soluciones para esos casos.

### La pila de la recursión

Vamos a estudiar el comportamiento de la evaluación de una llamada a una función recursiva. Supongamos la función *mi-length*:

```
(define (mi-length items)
  (if (null? items)
      0
      (+ 1 (mi-length (cdr items)))))
```

Examinamos cómo se evalúan las llamadas recursivas:

```
(mi-length '(a b c d))
(+ 1 (mi-length '(b c d)))
(+ 1 (+ 1 (mi-length '(c d))))
(+ 1 (+ 1 (+ 1 (mi-length '(d))))))
(+ 1 (+ 1 (+ 1 (+ 1 (mi-length '())))))
(+ 1 (+ 1 (+ 1 (+ 1 0)))))
(+ 1 (+ 1 (+ 1 1)))
(+ 1 (+ 1 2))
(+ 1 3)
4
```

Cada llamada a la recursión deja una función **en espera de ser evaluada** cuando la recursión devuelva un valor (en el caso anterior las funciones suma). Estas llamadas en espera, junto con sus argumentos, se almacenan en la *pila de la recursión*.

Cuando la recursión devuelve un valor, los valores se recuperan de la pila, se realiza la llamada y se devuelve el valor a la anterior llamada en espera.

Si la recursión está mal hecha y nunca termina se genera un *stack overflow* porque la memoria que se almacena en la pila sobrepasa la memoria reservada para el intérprete DrRacket.

### Coste espacial de la recursión

El coste espacial de un programa es una función que relaciona la memoria consumida por una llamada para resolver un problema con alguna variable que determina el tamaño del problema a resolver.

En el caso de la función *mi-length* el tamaño del problema viene dado por la longitud de la lista. El coste espacial de *mi-length* es  $O(n)$ , siendo  $n$  la longitud de la lista.

### El coste depende del número de llamadas a la recursión

Veamos con un ejemplo que el coste de las llamadas recursivas puede dispararse. Supongamos la famosa [secuencia de Fibonacci](#): 0,1,1,2,3,5,8,13,...

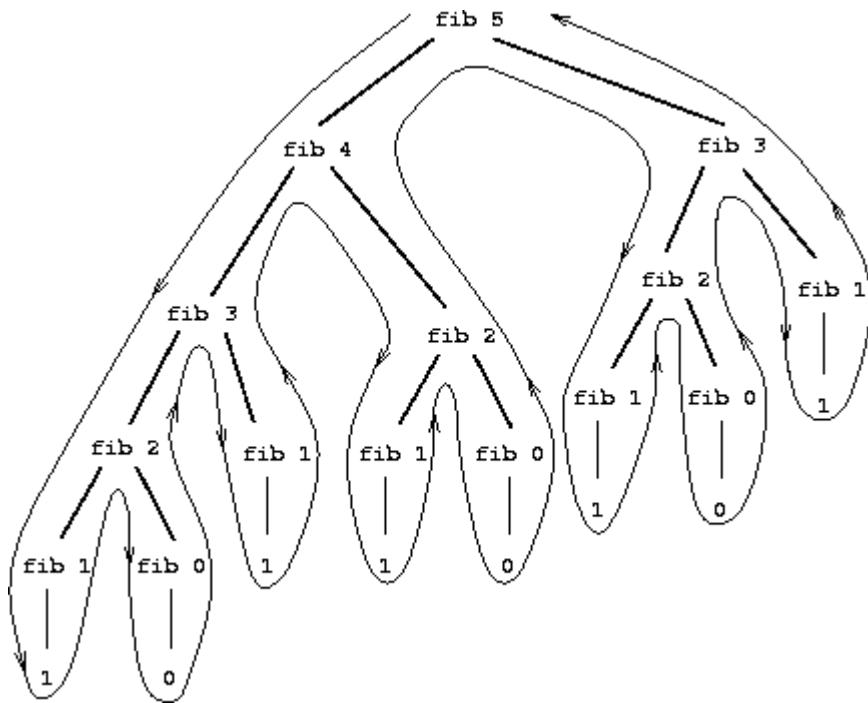
Formulación matemática de la secuencia de Fibonacci:

$$\begin{aligned} \text{Fibonacci}(n) &= \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2) \\ \text{Fibonacci}(0) &= 0 \\ \text{Fibonacci}(1) &= 1 \end{aligned}$$

Formulación recursiva en Scheme:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                  (fib (- n 2))))))
```

Evaluación de una llamada a Fibonacci:



Cada llamada a la recursión produce otras dos llamadas, por lo que el número de llamadas finales es  $2^n$  siendo  $n$  el número que se pasa a la función.

El coste espacial y temporal es exponencial,  $O(2^n)$ . Esto hace inviable utilizar esta implementación para realizar el cálculo de la función. Puedes comprobarlo intentando evaluar en el intérprete ([fib 35](#)).

## Soluciones al coste de la recursión: procesos iterativos

Diferenciamos entre procedimientos y procesos: un **procedimiento** es un algoritmo y un **proceso** es la ejecución de ese algoritmo.

Es posible definir *procedimientos recursivos* que generen *procesos iterativos* (como los bucles en programación imperativa) en los que **no se dejen llamadas recursivas en espera ni se incremente la pila de la recursión**. Para ello construimos la recursión de forma que en cada llamada se haga un cálculo parcial y en el caso base se pueda devolver directamente el resultado obtenido.

Este estilo de recursión se denomina *recursión por la cola* ([tail recursion](#), en inglés).

Se puede realizar una implementación eficiente de la ejecución del proceso, eliminando la pila de la recursión.

### Factorial iterativo

Empezamos a explicar la recursión por la cola con un ejemplo muy sencillo: la versión iterativa de la típica función [factorial](#). Le pondremos de nombre a la función [factorial-iter](#):

```

(define (factorial n)
  (fact-iter n n))

(define (fact-iter n result)
  (if (= n 1)
    
```

```
result
(fact-iter (- n 1) (* result (- n 1)) ))
```

La función (`fact-iter n result`) es la que define el proceso iterativo. Su argumento `n` es el valor del que hay que calcular el factorial y el argumento `result` es un parámetro adicional en el que se van guardando los resultados intermedios.

En cada llamada recursiva, `n` se va haciendo cada vez más pequeño y en `result` se va acumulando el cálculo del factorial. Al final de la recursión el factorial debe estar calculado en `result` y se devuelve.

Veamos la secuencia de llamadas:

```
(factorial 4)
(factorial-iter 4 4)
(factorial-iter 3 4*3=12)
(factorial-iter 2 12*2=24)
(factorial-iter 1 24*1=24)
24
```

Antes de realizar cada llamada recursiva se realiza el cálculo del resultado parcial, que se guarda en el parámetro `result`. Después se realiza la llamada con el nuevo valor calculado de `n` y de `result`.

Al final, cuando `n` vale `1` se devuelve el valor calculado de `result`. Este valor es el resultado completo de la recursión, ya que no hay que hacer ninguna operación más con él. A diferencia de los procesos recursivos, en los que se quedan llamadas en espera en la pila de la recursión, en los procesos iterativos no hay ninguna llamada en espera. El resultado devuelto por el caso base es directamente la solución de la recursión, no queda nada por hacer con este resultado.

Es importante el valor inicial de `resultado`. La función `factorial` se encarga de inicializar este parámetro. En este caso es el mismo valor del número `n` a calcular el factorial.

La secuencia de llamadas recursivas acumula en la variable `result` el valor del factorial:

```
4 * 3 * 2 * 1 = 24
```

## Versión iterativa de mi-length

Veamos un segundo ejemplo. ¿Cómo sería la versión iterativa de `mi-length`, la función que calcula la longitud de una lista?.

Tenemos que añadir un parámetro adicional en el que iremos acumulando el resultado parcial. En este caso, cada vez que llamemos a la recursión eliminando un elemento de la lista, incrementaremos en 1 el valor del resultado. Para que funcione bien este enfoque, debemos inicializar este resultado a 0.

La solución es la siguiente:

```
(define (mi-length lista)
  (mi-length-iter lista 0))

(define (mi-length-iter lista result)
  (if (null? lista)
      result
      (mi-length-iter (cdr lista) (+ result 1))))
```

Fijaros que, al igual que en la versión iterativa de factorial, no hay ninguna llamada a ninguna función que recoja el resultado de la llamada recursiva y haga algo con él. Directamente el resultado de la llamada recursiva es el resultado final de la recursión.

### Función **suma-lista** usando recursión por la cola

Veamos otro ejemplo. Supongamos que queremos calcular usando recursión por la cola la suma de los números de una lista.

Deberíamos añadir un parámetro adicional en el que vamos acumulando esa suma. Inicializaremos a 0 ese parámetro e iremos en cada llamada recursiva acumulando el primer elemento de la lista:

```
(define (suma-lista lista)
  (suma-lista-iter lista 0))

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

## Procesos iterativos

Un resumen de las características de los procesos iterativos resultantes de hacer una recursión por la cola:

- La recursión resultante es menos elegante.
- Se necesita una parámetro adicional en el que se van acumulando los resultados parciales.
- La última llamada a la recursión devuelve el valor acumulado.
- El proceso resultante de la recursión es iterativo en el sentido de que no deja llamadas en espera ni incurre en coste espacial.

## Fibonacci iterativo

Cualquier programa recursivo se puede transformar en otro que genera un proceso iterativo.

En general, las versiones iterativas son menos intuitivas y más difíciles de entender y depurar.

Veamos, por ejemplo, la formulación iterativa de Fibonacci:

```
(define (fib n)
  (fib-iter 1 0 n))

(define (fib-iter a b count)
  (if (= count 0)
      b
      (fib-iter (+ a b) a (- count 1))))
```

La secuencia de llamadas recursivas sería la siguiente:

```
(fib 6)
(fib-iter 1 0 6)
(fib-iter 1+0=1 1 5)
(fib-iter 1+1=2 1 4)
(fib-iter 2+1=3 2 3)
(fib-iter 3+2=5 3 2)
(fib-iter 5+3=8 5 1)
(fib-iter 8+5=13 8 0)
8
```

En la llamada recursiva **n**, el parámetro **a** guarda el valor de fibonacci **n+1** y el parámetro **b** guarda el valor de fibonacci **n**, que es el que se devuelve. Conseguimos **n** llamadas inicializando **count** a **n** y decrementando el parámetro en 1 cada vez.

## Triángulo de Pascal

El [triángulo de Pascal](#) es el siguiente triángulo de números.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
...
...
```

Si numeramos las filas y columnas empezando a contar por 0, la expresión general del valor en una fila y columna determinada se puede obtener con la siguiente definición recursiva:

```
Pascal (n, 0) = 1
Pascal (n, n) = 1
Pascal (fila, columna) =
  Pascal (fila-1,columna-1) + Pascal (fila-1, columna)
```

La función sólo está definida para valores de `columna` menores o iguales que `fila`.

En Scheme es fácil escribir una función recursiva que implemente la definición anterior:

```
(define (pascal fila col)
  (cond ((= col 0) 1)
        ((= col fila) 1)
        (else (+ (pascal (- fila 1) (- col 1))
                  (pascal (- fila 1) col)))))

(pascal 4 2)
; => 6
(pascal 8 4)
; => 70
(pascal 27 13)
; => 20058300
```

Hay que llamar a la función con un valor de `col` menor o igual que `fila`. En el caso en que se pase un valor `col` mayor que `fila` la recursión no termina y se entra en un bucle infinito.

La función tiene una formulación sencilla y funciona correctamente. Sin embargo, el coste de esta recursión es también exponencial, igual que pasaba en el caso de la secuencia de fibonacci. Por ejemplo, la última expresión (`pascal 27 13`) tarda un buen rato en devolver el resultado. Sería imposible calcular el valor de números de Pascal un poco más grandes, como (`pascal 40 20`).

Veamos cómo se puede conseguir una versión iterativa.

La idea es definir una función iterativa `pascal-fila` a la que le pasamos el número de fila `n` y nos devuelve la lista de `n+1` números que constituyen la fila `n` del triángulo de Pascal:

```
fila 0 = (1)
fila 1 = (1 1)
fila 2 = (1 2 1)
fila 3 = (1 3 3 1)
fila 4 = (1 4 6 4 1)
...
...
```

Esta función necesitará un parámetro adicional (`lista-fila`) que se inicializa con la lista `(1)` y en el que se va guardando cada fila sucesiva. Esta fila va creciendo hasta que llegamos a la fila que tenemos que devolver. Hay que hacer la iteración `n` veces, por lo que vamos decrementando el parámetro `n` hasta que se llega a 0.

Para implementar esta función usamos otra llamada (`pascal-sig-fila lista-fila`) que recibe una fila del triángulo y devuelve la siguiente.

Por ejemplo:

```
(pascal-sig-fila '(1 3 3 1))
; => (1 4 6 4 1)
```

Esta función la implementamos con una función recursiva auxiliar (esta es recursiva pura) llamada ([pascal-suma-dos-a-dos lista-fila](#)) que es la que se encarga de realizar el cálculo de la nueva fila.

El código completo es el siguiente:

```
(define (pascal fila col)
  (list-ref (pascal-fila '(1) fila) col))

(define (pascal-fila lista-fila n)
  (if (= 0 n)
      lista-fila
      (pascal-fila (pascal-sig-fila lista-fila) (- n 1)))))

(define (pascal-sig-fila lista-fila)
  (append '(1)
          (pascal-suma-dos-a-dos lista-fila)
          '(1)))

(define (pascal-suma-dos-a-dos lista-fila)
  (if (null? (cdr lista-fila))
      '()
      (cons (+ (car lista-fila) (car (cdr lista-fila)))
            (pascal-suma-dos-a-dos (cdr lista-fila)))))
```

Con esta implementación ya no se tiene un coste exponencial y se puede calcular el valor de números como Pascal(40, 20):

```
(pascal 40 20)
; => 137846528820
```

## Soluciones al coste de la recursión: memoization

Una alternativa que mantiene la elegancia de los procesos recursivos y la eficiencia de los iterativos es la [memoization](#). Si miramos la traza de ([fibonacci 4](#)) podemos ver que el coste está producido por la repetición de llamadas; por ejemplo ([fibonacci 3](#)) se evalúa 2 veces.

En programación funcional la llamada a ([fibonacci 3](#)) siempre va a devolver el mismo valor.

La idea de la [memoization](#) es guardar el valor devuelto por la cada llamada en alguna estructura (una lista de asociación, por ejemplo) y no volver a realizar la llamada a la recursión las siguientes veces.

### Fibonacci con memoization

Para implementar la [memoization](#) necesitamos odos métodos imperativos [put](#) y [get](#) que implementan un diccionario *clave-valor*.

- La función (`put key value dic`) asocia un valor a una clave, la guarda en el diccionario (con mutación) y devuelve el valor.
- La función (`get key dic`) devuelve el valor del diccionario asociado a una clave. En el caso en que no exista ningún valor se devuelve `#f`.

Ejemplos:

```
(define mi-dic (crea-diccionario))
(put 1 10 mi-dic) ; => 10
(get 1 mi-dic) ; => 10
(get 2 mi-dic) ; => #f
```

Estos métodos son imperativos porque modifican (mutan) la estructura de datos que pasamos como parámetro. Para implementarlos tenemos que salirnos del paradigma funcional, usando las funciones de Racket que permiten mutar las parejas.

No es importante la implementación, la dejamos aquí como referencia y para poder probar la *memoization*.

```
(define (crea-diccionario)
  (mcons '*diccionario* '()))

(define (busca key dic)
  (cond
    ((null? dic) #f)
    ((equal? key (mcar (mcar dic)))
     (mcar dic))
    (else (busca key (mcdr dic)))))

(define (get key dic)
  (define record (busca key (mcdr dic)))
  (if (not record)
      #f
      (mcdr record)))

(define (put key value dic)
  (define record (busca key (mcdr dic)))
  (if (not record)
      (set-mcdr! dic
                  (mcons (mcons key value)
                         (mcdr dic)))
      (set-mcdr! record value))
  value))
```

La función `fib-memo` realiza el cálculo de la serie de Fibonacci utilizando exactamente la misma definición recursiva original, pero añadiendo la técnica de *memoization*: lo primero que hacemos para calcular el número de fibonacci `n`, antes de llamar a la recursión, es comprobar si está ya guardado en la lista de asociación. En el caso en que esté, lo devolvemos. Sólo cuando el número no está calculado llamamos a la recursión para calcularlo.

La implementación se muestra a continuación. Vemos que para devolver el número de fibonacci `n` se comprueba si ya está guardado en la lista. Sólo en el caso en que no esté guardado se llama a la recursión para calcularlo y guardarlo. La función `put` que guarda el nuevo valor calculado también lo devuelve.

```
(define (fib-memo n dic)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        ((not (equal? (get n dic) #f))
         (get n dic))
        (else (put n (+ (fib-memo (- n 1) dic)
                          (fib-memo (- n 2) dic))) dic))))
```

Podemos comprobar la diferencia de tiempos de ejecución entre esta versión y la anterior. El coste de la función *memoizada* es  $O(n)$ . Frente al coste  $O(2^n)$  de la versión inicial que la hacía imposible de utilizar.

```
(fib-memo 200 lista)
⇒ 280571172992510140037611932413038677189525
```

## Recursión y gráficos de tortuga

Vamos a terminar el apartado sobre procedimientos recursivos con un último ejemplo algo distinto de los vistos hasta ahora. Usaremos la recursión para dibujar figuras fractales usando los denominados *gráficos de tortuga*. Para dibujar las figuras tendremos que utilizar un estilo de programación no funcional, dibujando los distintos trazos de las figuras con pasos de ejecución secuenciales. Para ello usaremos una primitiva imperativa de Scheme: la forma especial `begin` que permite realizar un grupo de pasos de ejecución de forma secuencial.

!!! Warning "Aviso" Ten cuidado con la forma especial `begin`, es una forma especial imperativa. No debes usarla en la implementación de ninguna función cuando estemos usando el paradigma funcional.

### Gráficos de tortuga en Racket

Se pueden utilizar los [gráficos de tortuga](#) en Racket cargando la librería (`graphics turtles`):

```
#lang racket
(require graphics/turtles)
```

Los comandos más importantes de esta librería son:

- (`turtles #t`): abre una ventana y coloca la tortuga en el centro, mirando hacia el eje X (derecha)
- (`clear`): borra la ventana y coloca la tortuga en el centro
- (`draw d`): avanza la tortuga dibujando  $d$  píxeles
- (`move d`): mueve la tortuga  $d$  píxeles hacia adelante (sin dibujar)
- (`turn g`): gira la tortuga  $g$  grados (positivos: en el sentido contrario a las agujas del reloj)

Prueba a realizar algunas figuras con los comandos de tortuga, antes de escribir el algoritmo en Scheme del triángulo de Sierpinski.

Por ejemplo, podemos definir una función que dibuja un triángulo rectángulo con catetos de longitud `x`:

```
(define (hipot x)
  (* x (sqrt 2)))

(define (triangulo-rectangulo x)
  (begin
    (draw x)
    (turn 90)
    (draw x)
    (turn 135)
    (draw (hipot x))
    (turn 135)))

(triangulo-rectangulo 100)
```

La función (`hipot x`) devuelve la longitud de la hipotenusa de un triángulo rectángulo con dos lados de longitud `x`. O sea, la expresión:

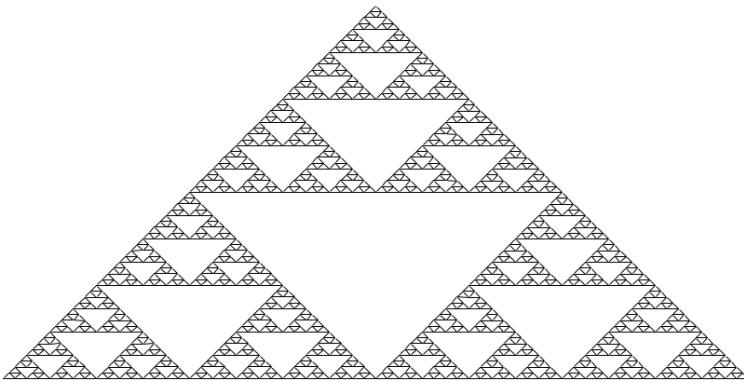
$$\text{hipot}(x) = \sqrt{x^2 + x^2} = x\sqrt{2}$$

Como puedes comprobar, el código es imperativo. La forma especial `begin` permite realizar una serie de pasos de ejecución que modifican el estado (posición y orientación) de la *tortuga* y dibujan los trazos de la figura.

El siguiente código es una variante del anterior que dibuja un triángulo rectángulo de base `w` y lados `w/2`. Va a ser la figura base del triángulo de Sierpinski.

```
(define (triangle w)
  (begin
    (draw w)
    (turn 135)
    (draw (hipot (/ w 2))))
    (turn 90)
    (draw (hipot (/ w 2))))
    (turn 135)))
```

## Triángulo de Sierpinski



### Triángulo de Sierpinski

- ¿Ves alguna recursión en la figura?
- ¿Cuál podría ser el parámetro de la función que la dibujara?
- ¿Se te ocurre un algoritmo recursivo que la dibuje?

La figura es *autosimilar* (una característica de las figuras fractales). Una parte de la figura es idéntica a la figura total, pero reducida de escala. Esto nos da una pista de que es posible dibujar la figura con un algoritmo recursivo.

Para intentar encontrar una forma de enfocar el problema, vamos a pensarlo de la siguiente forma: supongamos que tenemos un triángulo de Sierpinski de anchura  $h$  y altura  $h/2$  con su esquina inferior izquierda en la posición  $0,0$ . ¿Cómo podríamos construir **el siguiente** triángulo de Sierpinski?.

Podríamos construir un triángulo de Sierpinski más grande dibujando 3 veces el mismo triángulo, pero en distintas posiciones:

1. Triángulo 1 en la posición  $(0,0)$
2. Triángulo 2 en la posición  $(h/2, h/2)$
3. Triángulo 3 en la posición  $(h, 0)$

El algoritmo recursivo se basa en la misma idea, pero *hacia atrás*. Debemos intentar dibujar un triángulo de altura  $h$  situado en la posición  $x, y$  basándonos en 3 llamadas recursivas a triángulos más pequeños. En el caso base, cuando  $h$  sea menor que un umbral, dibujaremos un triángulo de lado  $h$  y altura  $h/2$ :

O sea, que para dibujar un triángulo de Sierpinski de base  $h$  y altura  $h/2$  debemos:

- Dibujar tres triángulos de Sierpinsky de la mitad del tamaño del original ( $h/2$ ) situadas en las posiciones  $(x,y)$ ,  $(x+h/4, y+h/4)$  y  $(x+h/2, y)$
- En el caso base de la recursión, en el que  $h$  es menor que una constante, se dibuja un triángulo de base  $h$  y altura  $h/2$ .

Una versión del algoritmo en *pseudocódigo*:

```
Sierpinsky (x, y, h):
    if (h > MIN) {
        Sierpinsky (x, y, h/2)
        Sierpinsky (x+h/4, y+h/4, h/2)
```

```

    Sierpinsky (x+h/2, y, h/2)
} else dibujaTriangulo (x, y, h)

```

## Sierpinski en Racket

La siguiente es una versión imperativa del algoritmo que dibuja el triángulo de Sierpinski. No es funcional porque se realizan *pasos de ejecución*, usando la forma especial `begin` o múltiples instrucciones en una misma función (por ejemplo la función `triangle`).

```

#lang racket
(require graphics/turtles)

(turtles #t)

(define (hipot x)
  (* x (sqrt 2)))

(define (triangle w)
  (begin
    (draw w)
    (turn 135)
    (draw (hipot (/ w 2)))
    (turn 90)
    (draw (hipot (/ w 2)))
    (turn 135)))

(define (sierpinski w)
  (if (> w 20)
      (begin
        (sierpinski (/ w 2))
        (move (/ w 4)) (turn 90) (move (/ w 4)) (turn -90)
        (sierpinski (/ w 2))
        (turn -90) (move (/ w 4)) (turn 90) (move (/ w 4))
        (sierpinski (/ w 2))
        (turn 180) (move (/ w 2)) (turn -180)) ; volvemos a la posición original
      (triangle w)))

```

La llamada a

```
(sierpinski 40)
```

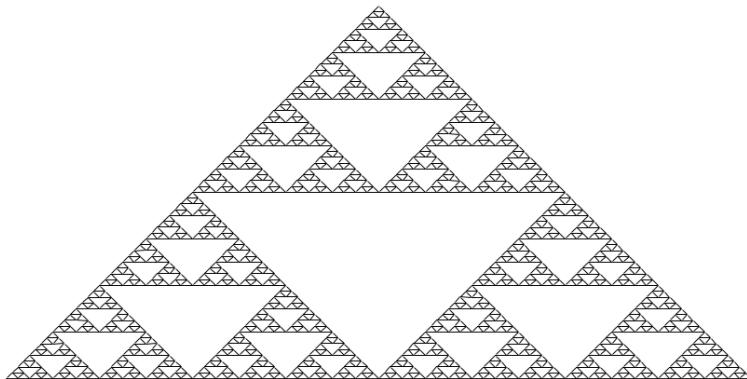
produce la siguiente figura:



La llamada a

```
(sierpinski 700)
```

Produce la figura que vimos al principio del apartado:



Para ocupar la venta completa debemos desplazar la tortuga hacia atrás antes de invocar a `sierpinski`:

```
(clear)
(move -350)
(sierpinski 700)
```

## Recursión mutua

En la recursión mutua definimos una función en base a una segunda, que a su vez se define en base a la primera.

También debe haber un caso base que termine la recursión

Por ejemplo:

- $x$  es par si  $x-1$  es impar
- $x$  es impar si  $x-1$  es par
- 0 es par

Programas en Scheme:

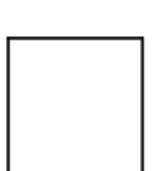
```
(define (par? x)
  (if (= 0 x)
      #t
      (impar? (- x 1)))))

(define (impar? x)
  (if (= 0 x)
      #f
      (par? (- x 1))))
```

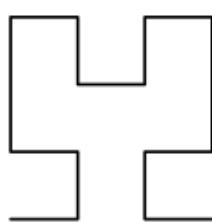
## Ejemplo avanzado: curvas de Hilbert

La curva de Hilbert es una curva fractal que tiene la propiedad de llenar completamente el espacio

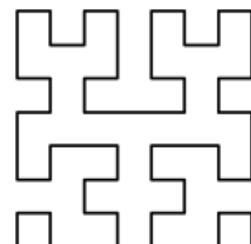
Su dibujo tiene una formulación recursiva:



$H_1$



$H_2$



$H_3$

La curva  $H_3$  se puede construir a partir de la curva  $H_2$ . El algoritmo recursivo se formula dibujando la curva  $i$ -ésima a partir de la curva  $i-1$ .

Para dibujar una curva de Hilbert de orden  $i$  a la *derecha* de la tortuga:

1. Gira la tortuga -90
2. Dibuja una curva de orden  $i-1$  a la izquierda
3. Avanza long dibujando
4. Gira 90
5. Dibuja una curva de orden  $i-1$  a la derecha
6. Avanza long dibujando
7. Dibuja una curva de orden  $i-1$  a la derecha
8. Gira 90
9. Avanza long dibujando
10. Dibuja una curva de orden  $i-1$  a la izquierda
11. Gira -90

El algoritmo para dibujar a la izquierda es simétrico.

Como en la curva de Sierpinsky, utilizamos la librería *graphics/turtles*, que permite usar la tortuga de Logo con los comandos de Logo *draw* y *turn*. Definimos dos funciones simétricas, la función (*h-der i long*) que dibuja una curva de Hilbert de orden *i* con una longitud de trazo *long* a la *derecha* de la tortuga y la función (*h-izq i w*) que dibuja una curva de Hilbert de orden *i* con una longitud de trazo *long* a la *izquierda* de la tortuga.

El algoritmo en Scheme:

```
#lang racket
(require graphics/turtles)

(define (h-izq i long)
  (if (> i 0)
      (begin
```

```

(turn 90)
(h-der (- i 1) long)
(draw long)
(turn -90)
(h-izq (- i 1) long)
(draw long)
(h-izq (- i 1) long)
(turn -90)
(draw long)
(h-der (- i 1) long)
(turn 90))
(move 0)))

(define (h-der i long)
(if (> i 0)
(begin
  (turn -90)
  (h-izq (- i 1) long)
  (draw long)
  (turn 90)
  (h-der (- i 1) long)
  (draw long)
  (h-der (- i 1) long)
  (turn 90)
  (draw long)
  (h-izq (- i 1) long)
  (turn -90))
(move 0)))

```

Podemos probarlo con distintos parámetros de grado de curva y longitud de trazo.

Curva de Hilbert de nivel 3 con trazo de longitud 20:

```
(clear)
(h-izq 3 30)
```

Para entender mejor el algoritmo podemos dibujar paso a paso esta figura usando los siguientes comandos:

```
(clear)
(turn 90)
(h-der 2 30)
(draw 30)
(turn -90)
(h-izq 2 30)
(draw 30)
(h-izq 2 30)
(turn -90)
(draw 30)
```

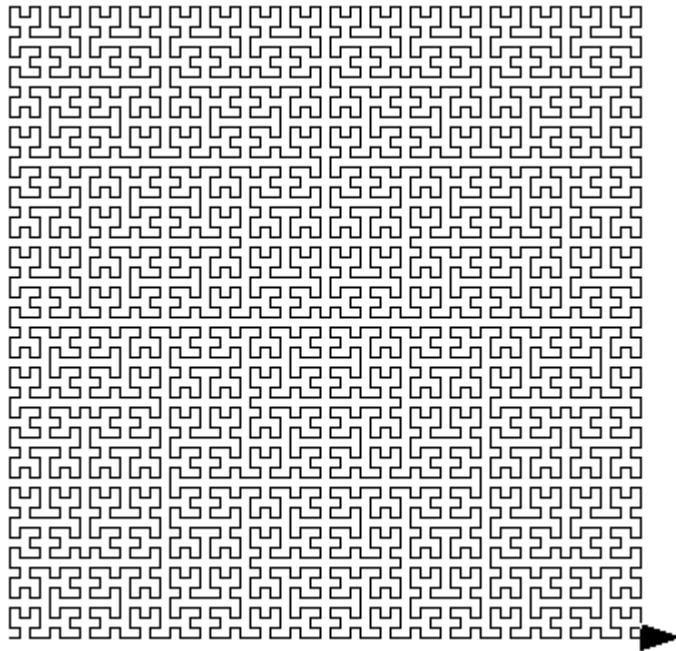
```
(h-der 2 30)  
(turn 90)
```

Curva de Hilbert de nivel 6 con trazo de longitud 10:

```
(clear)  
(move -350)  
(turn -90)  
(move 350)  
(turn 90)  
(h-izq 6 10)
```

Curva de Hilbert de nivel 7 con trazo de longitud 5:

```
(clear)  
(move -350)  
(turn -90)  
(move 350)  
(turn 90)  
(h-izq 7 5)
```



## Bibliografía - SICP

En este tema explicamos conceptos de los siguientes capítulos del libro *Structure and Interpretation of Computer Programs*:

- 1.2 - Procedures and the Processes They Generate
- 1.2.1 - Linear Recursion and Iteration
- 1.2.2 - Tree Recursion

---

Lenguajes y Paradigmas de Programación, curso 2019-20

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante

Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez

## Tema 4: Estructuras de datos recursivas

---

### Listas estructuradas

Hemos visto que las listas en Scheme se implementan como un estructura de datos recursiva, formada por una pareja que enlaza en su parte derecha el resto de la lista y que termina con una parte derecha en la que hay una lista vacía.

En este apartado vamos a volver a estudiar las listas desde un nivel de abstracción alto, usando las funciones:

- (`car lista`) para obtener el primer elemento de una lista
- (`cdr lista`) para obtener el resto de la lista
- (`cons dato lista`) para construir una nueva lista con el dato como primer elemento

En la mayoría de funciones y ejemplos que hemos visto hasta ahora las listas están formadas por datos y el recorrido por la lista es un recorrido lineal, iterando por sus elementos.

En este apartado vamos a ampliar este concepto y estudiar cómo trabajar con *listas que contienen otras listas*.

Veremos que esto cambia fundamentalmente la estructura de las listas y de las funciones que van a operar con ellas. El cambio fundamental es que la función `car lista` puede devolver dos tipos de elementos:

- Un elemento de la lista (del tipo de elementos que hay en la lista)
- Otra lista (formada por el tipo de elementos de la lista)

### Definición y ejemplos

Las listas en Scheme pueden tener cualquier tipo de elementos, incluido otras listas.

Llamaremos **lista estructurada** a una lista que contiene otras sublistas. Lo contrario de lista estructurada es una **lista plana**, una lista formada por elementos que no son listas. Llamaremos **hojas** a los elementos de una lista que no son sublistas.

A las listas estructuradas cuyas hojas son símbolos se les denomina en el contexto de la programación funcional *expresiones-S (S-expression)*.

Por ejemplo, la lista estructurada:

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

es una lista estructurada con 4 elementos:

- El elemento '`a`', una hoja
- El elemento '`b`', otra hoja

- La lista plana (c d e)
- La lista estructurada (f (g h))

Se puede construir con cualquiera de las siguientes expresiones:

```
(define lista (list 'a 'b (list 'c 'd 'e) (list 'f (list 'g 'h))))
(define lista '(a b (c d e) (f (g h))))
```

Una lista formada por parejas la consideraremos una lista plana, ya que no contiene ninguna sublista. Por ejemplo, la lista

```
((a . 3) (b . 5) (c . 12))
```

es una lista plana de tres elementos (hojas) que son parejas.

## Definiciones en Scheme

Vamos a escribir las definiciones anteriores de **hoja**, **plana** y **estructurada** usando código de Scheme.

### Función (hoja? dato)

Definimos una hoja como aquellos elementos de una lista estructurada que no son listas:

```
(define (hoja? elem)
  (not (list? elem)))
```

Utilizaremos esta función para comprobar si un determinado elemento de una lista es o no una hoja. Por ejemplo, supongamos la siguiente lista:

```
((1 2) 3 4 (5 6))
```

Es una lista de 4 elementos, siendo el primero y el último otras sublistas y el segundo y el tercero hojas. Podemos comprobar si son o no hojas sus elementos:

```
(define lista '((1 2) 3 4 (5 6)))
(hoja? (car lista)) ; => #f
(hoja? (cadr lista)) ; => #t
(hoja? (caddr lista)) ; => #t
(hoja? (cadddr lista)) ; => #f
```

La lista vacía no es una hoja

```
(hoja? '()) ; => #f
```

### Función (plana? lista)

Como hemos dicho antes, una lista es plana cuando todos sus elementos son hojas. Queremos implementar la función **(plana? lista)** que lo compruebe.

Por ejemplo:

```
(plana? '(a b c d e f)) ; => #t
(plana? (list (cons 'a 1) "Hola" #f)) ; => #t
(plana? '(a (b c) d)) ; => #f
(plana? '(a () b)) ; => #f
```

Una definición recursiva de lista plana:

Una lista es plana si y solo si el primer elemento es una hoja y el resto es plana.

Y el caso base:

Una lista vacía es plana.

Usando esta definición recursiva, podemos implementar en Scheme la función **(plana? lista)** que comprueba si una lista es plana:

```
(define (plana? lista)
  (or (null? lista)
      (and (hoja? (car lista))
           (plana? (cdr lista)))))
```

Se podría también implementar la función **plana?** usando la función de orden superior **for-all?** que comprueba que todos los elementos de una lista cumplen una propiedad. En este caso, ser hoja.

```
(define (plana-fos? lista)
  (for-all? hoja? lista))
```

!!! Note "Función **for-all?**" Recordemos que la función **(for-all? predicado lista)** se implementa de la siguiente forma:

```
```racket
(define (for-all? predicado lista)
  (or (null? lista)
      (and (predicado (car lista))
           (for-all? predicado (cdr lista))))
```

```
```racket
(for-all? predicado (cdr lista)))))

```

```

### Función (estructurada? lista)

Una lista es estructurada cuando alguno de sus elementos es otra lista. Como caso base, una lista vacía no es estructurada.

Queremos implementar la función (estructurada? lista) que compruebe si una lista es estructurada.

```
(estructurada? '(1 2 3 4)) ; => #f
(estructurada? (list (cons 'a 1) (cons 'b 2) (cons 'c 3))) ; => #f
(estructurada? '(a () b)) ; => #t
(estructurada? '(a (b c) d)) ; => #t
```

```
(define (estructurada? lista)
  (and (not (null? lista))
       (or (list? (car lista))
           (estructurada? (cdr lista))))))
```

Se podría implementar también usando la función de orden superior **exists?** para consultar si algún elemento de la lista es también otra lista.

```
(define (estructurada-fos? lista)
  (exists? list? lista))
```

!!! Note "Función **exists?**" Recordemos que la función (exists? predicado lista) se implementa de la siguiente forma:

```
```racket
(define (exists? predicado lista)
  (if (null? lista)
      #f
      (or (predicado (car lista))
          (exists? predicado (cdr lista)))))

```

```

Realmente bastaría con haber hecho una de las dos definiciones y escribir la otra como la negación de la primera:

```
(define (estructurada? lista)
  (not (plana? lista)))
```

## Ejemplos de listas estructuradas

Las listas estructuradas son muy útiles para representar información jerárquica en donde queremos representar elementos que contienen otros elementos.

Por ejemplo, las expresiones de Scheme son listas estructuradas:

```
(= 4 (+ 2 2))
(if (= x y) (* x y) (+ (/ x y) 45))
(define (factorial x) (if (= x 0) 1 (* x (factorial (- x 1)))))
```

El análisis sintáctico de una oración puede generar una lista estructurada de símbolos, en donde se agrupan los distintos elementos de la oración:

```
((Juan) (compró) (la entrada (de la película)) (el viernes por la tarde))
```

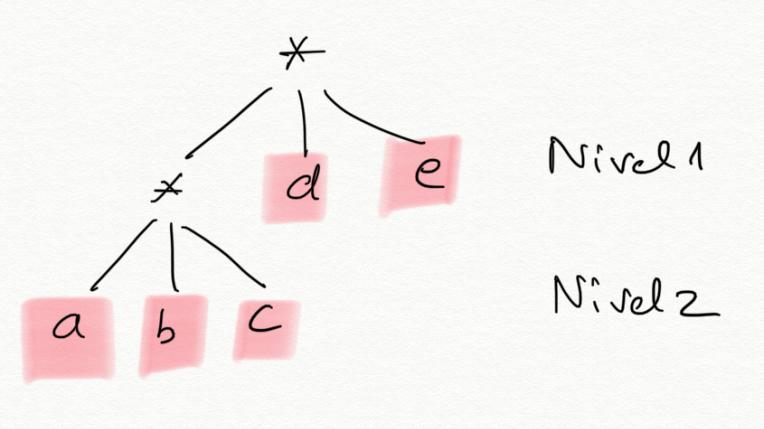
Una página HTML, con sus distintos elementos, unos dentro de otros, también se puede representar con una lista estructurada:

```
((<h1> Mi lista de la compra </h1>)
 (<ul> (<li> naranjas </li>)
        (<li> tomates </li>)
        (<li> huevos </li>) </ul>))
```

## Pseudo árboles con niveles

Las listas estructuradas definen una estructura de niveles, donde la lista inicial representa el primer nivel, y cada sublista representa un nivel inferior. Los datos de las listas representan las hojas.

Por ejemplo, la representación en forma de niveles de la lista `((a b c) d e)` es la siguiente:



Cada asterisco \* representa una lista. Las ramas que salen del asterisco representan los elementos de la lista. En el ejemplo tenemos en un primer nivel una lista con 3 elementos: la lista (a b c), d y e. Y en el segundo nivel se encuentra la lista (a b c) cuyos 3 elementos son hojas.

Las hojas **d** y **e** están en el nivel 1 y en las posiciones 2 y 3 de la lista y las hojas **a**, **b** y **c** en el nivel 2.

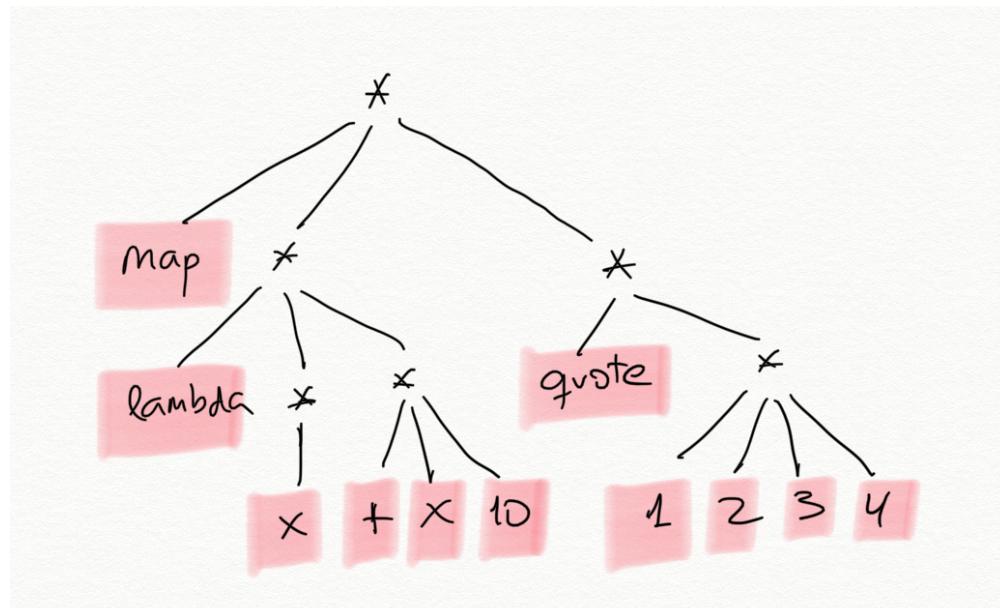
!!! Warning "Una lista estructurada no es un árbol" Una lista estructurada no es un árbol propiamente dicho, porque un árbol tiene datos en todos los nodos, mientras que en la lista estructurada los datos están sólo en las hojas.

Las listas estructuradas sirven para agrupar de forma jerárquica un conjunto de datos en distintos niveles.

A pesar de ser distintas de los árboles, ambas son estructuras de datos jerárquicas (con niveles) que se pueden definir de forma recursiva y sobre las que se pueden definir algoritmos recursivos. Veremos más adelante cómo definir y trabajar con árboles en Scheme.

Otro ejemplo. ¿Cuál sería la representación en niveles de la siguiente lista estructurada?:

```
(map (lambda (x) (+ x 10)) (quote (1 2 3 4)))
```



Funciones recursivas sobre listas estructuradas

## Número de hojas

Veamos como primer ejemplo la función (`num-hojas lista`) que cuenta el número de hojas de una lista estructurada.

Por ejemplo:

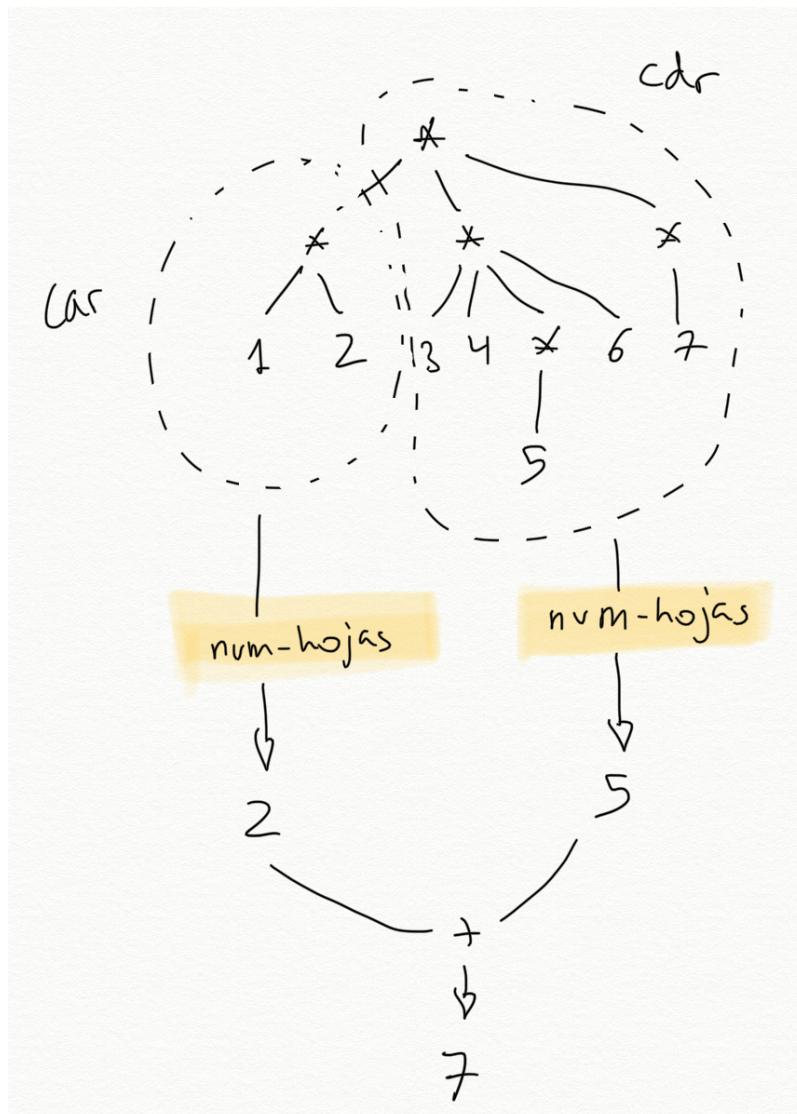
```
(num-hojas '((1 2) (3 4 (5) 6) (7))) ; => 7
```

Como hemos comentado antes, una cuestión clave en las funciones que vamos a construir sobre listas estructuradas es que el `car` de una lista estructurada puede ser a su vez otra lista.

Para calcular el número de hojas de una lista podemos obtener el primer elemento y el resto de la lista, y contar recursivamente el número de hojas del primer elemento y del resto. Al ser una lista estructurada, el primer elemento puede ser a su vez otra lista, por lo que llamamos a la recursión para contar sus hojas.

La definición de este caso general usando *pseudocódigo* es:

El número de hojas de una lista estructurada es la suma del número de hojas de su primer elemento (que puede ser otra lista) y del número de hojas del resto.



La recursión tiene dos llamadas recursivas. Una que recibe el elemento de la cabeza de la lista y otra que recibe el resto de la lista.

```
;Caso general num-hojas
(define (num-hojas lista)
  ; Falta caso base
  (+ (num-hojas (car lista))
    (num-hojas (cdr lista))))
```

!!! Warning "No hay coste exponencial" A pesar de haber dos llamadas recursivas, no pasa lo mismo que en Fibonacci o Pascal ya que no se van a repetir llamadas a la recursión con los mismos datos. La recursión recorre la lista estructurada y su coste será el número de elementos de la lista.

Para considerar el **caso base**, veamos cómo las llamadas recursivas reciben cada vez un problema más pequeño.

La llamada recursiva sobre el resto de la lista recibe cada vez una lista con 1 elemento menos. Al final se llamará a la función con una lista vacía. Ese será un caso base. El número de elementos de una lista vacía es 0.

La llamada recursiva sobre la cabeza de la lista es algo distinta. Recibe una lista en la que se ha descendido un nivel y tiene, por tanto, un nivel menos. Al final se llamará a la función con una hoja (un dato). Ese será el otro caso base y habrá que devolver 1.

La definición completa de la función queda de la siguiente forma:

```
(define (num-hojas elem)
  (cond
    ((null? elem) 0)
    ((hoja? elem) 1)
    (else (+ (num-hojas (car elem))
              (num-hojas (cdr elem))))))
```

Hay que hacer notar que el parámetro `elem` puede ser tanto una lista como un dato atómico (una hoja). Estamos aprovechándonos de la característica de Scheme de ser débilmente tipado para hacer un código bastante conciso.

### Versión con funciones de orden superior

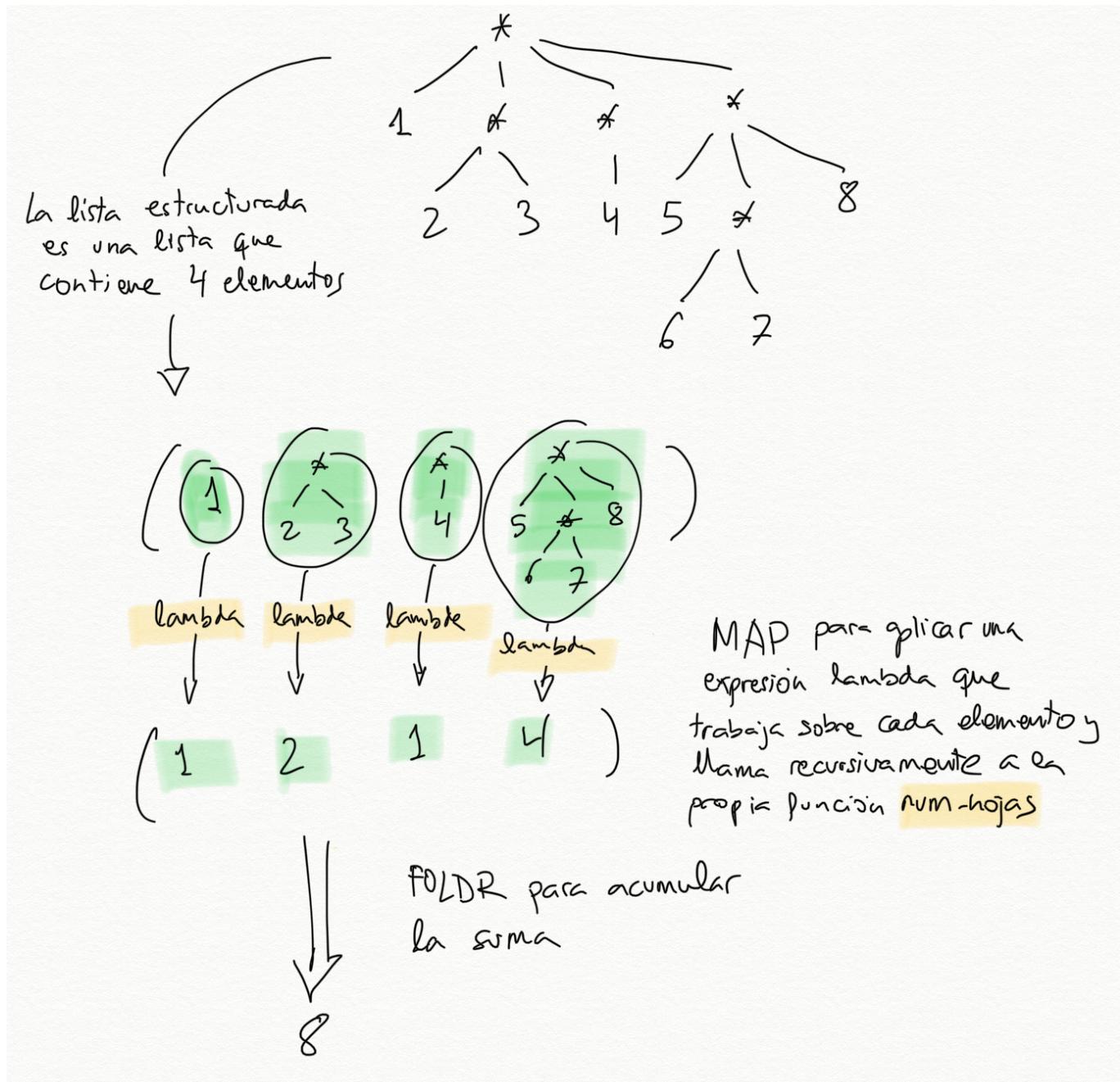
Podemos usar también las funciones de orden superior `map` y `foldr` para obtener una versión más concisa.

Una lista estructurada tiene como elementos en un primer nivel hojas o otras sublistas. Podemos entonces mapear una expresión lambda que se aplica a cada uno de esos elementos. En la expresión lambda comprobamos si el elemento (el parámetro `sublista` de la expresión lambda) es una hoja o una lista. En el primero caso devolvemos 1. En el segundo aplicaremos *la propia función que estamos definiendo* sobre la sublista, con lo que se devolverá el número de hojas de esa sublista.

El resultado del map será una lista de números (el número de hojas de cada componente), que podemos sumar haciendo un **foldr** con la función **+**:

```
(define (num-hojas-fos lista)
  (foldr + 0 (map (lambda (elem)
    (if (hoja? elem)
        1
        (num-hojas-fos elem)))) lista)))
```

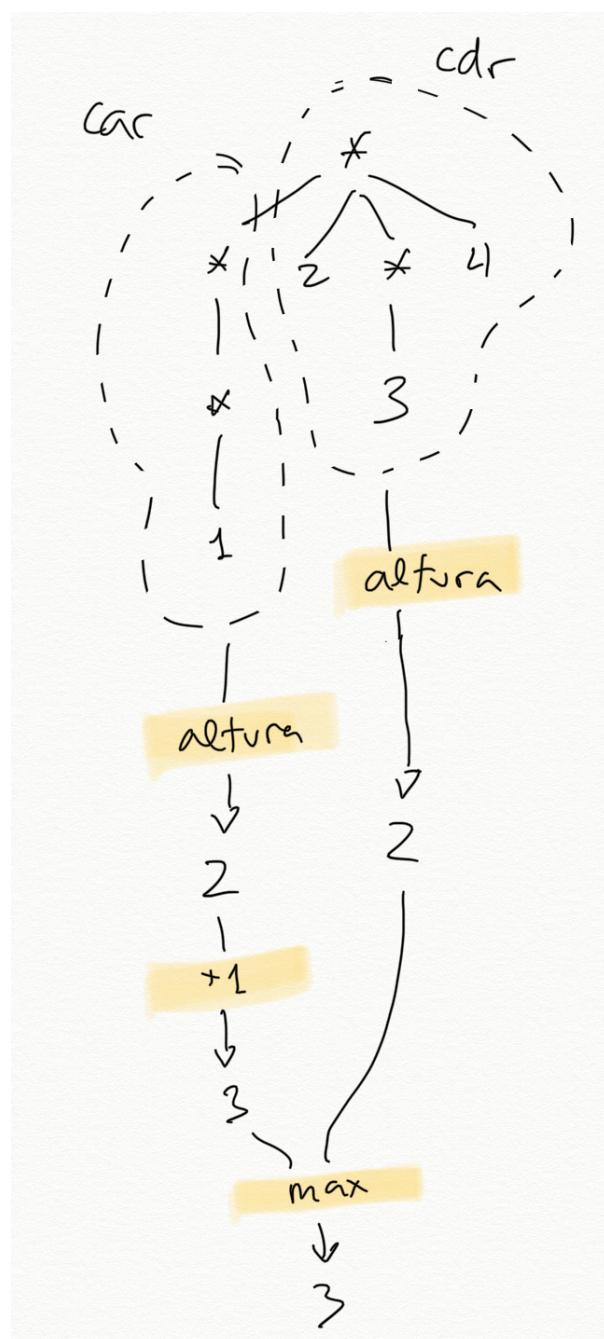
Una explicación gráfica de cómo funciona la función sobre la lista **(1 (2 3) (4) (5 (6 7) 8))**:



### Altura de una lista estructurada

La *altura* de una lista estructurada viene dada por su número de niveles: una lista plana tiene una altura de 1, la lista **((1 2 3) 4 5)** tiene una altura de 2.

Para calcular la altura de una lista estructurada tenemos que obtener (de forma recursiva) la altura de su primer elemento, y la altura del resto de la lista, sumarle 1 a la altura del primer elemento y devolver el máximo de los dos números.



Como casos base, la altura de una lista vacía o de una hoja (dato) es 0.

En Scheme:

```
(define (altura elem)
  (cond
    ((null? elem) 0)
    ((hoja? elem) 0)
    (else (max (+ 1 (altura (car elem)))
               (altura (cdr elem))))))
```

Por ejemplo:

```
(altura '(1 (2 3) 4)) ; => 2
(altura '(1 (2 (3)) 3)) ; => 3
```

### Versión con funciones de orden superior

Y la segunda versión, usando las funciones de orden superior `map` para obtener la altura de sus elementos del primer nivel (puedes ser hojas o sublistas) y `foldr` para quedarse con el máximo de la lista de valores que devuelve el `map`.

```
(define (altura-fos lista)
  (+ 1 (foldr max 0 (map (lambda (elem)
                             (if (hoja? elem)
                                 0
                                 (altura-fos elem)))) lista)))
```

## Otras funciones recursivas

Vamos a diseñar otras funciones recursivas que trabajan con la estructura jerárquica de las listas estructuradas.

- (`aplana lista`): devuelve una lista plana con todas las hojas de la lista
- (`pertenece-lista? dato lista`): busca una hoja en una lista estructurada
- (`nivel-hoja dato lista`): devuelve el nivel en el que se encuentra un dato en una lista
- (`cuadrado-estruct lista`): eleva todas las hojas al cuadrado (suponemos que la lista estructurada contiene números)
- (`map-estruct f lista`): similar a `map`, aplica una función a todas las hojas de la lista estructurada y devuelve el resultado (otra lista estructurada)

### `(aplana lista)`

Devuelve una lista plana con todas las hojas de la lista.

Por ejemplo:

```
(aplana '(1 2 (3 (4 (5))) (((6)))))  
; => (1 2 3 4 5 6)
```

La solución recursiva es:

```
(define (aplana elem)
  (cond
    ((null? elem) '())
    ((hoja? elem) (list elem))
    (else (map aplana (cdr elem))))
```

```
(else
  (append (aplana (car elem))
          (aplana (cdr elem)))))
```

Con funciones de orden superior:

```
(define (aplana-fos lista)
  (foldr append
         '()
         (map (lambda (elem)
                 (if (hoja? elem)
                     (list elem)
                     (aplana-fos elem))) lista)))
```

**(pertenece-lista? dato lista)**

Comprueba si el **dato** aparece en la lista estructurada.

```
(pertenece? 'a '(b c (d (a))) ; => #t
(pertenece? 'a '(b c (d e (f)) g)) ; => #f
```

Solución recursiva:

```
(define (pertenece? dato elem)
  (cond
    ((null? elem) #f)
    ((hoja? elem) (equal? dato elem))
    (else (or (pertenece? dato (car elem))
              (pertenece? dato (cdr elem))))))
```

Con funciones de orden superior:

```
(define (pertenece-fos? dato lista)
  (exists (lambda (elem)
            (if (hoja? elem)
                (equal? dato elem)
                (pertenece-fos? dato elem))) lista))
```

**(nivel-hoja dato lista)**

Veamos una última función **(nivel-hoja dato lista)** que recorre una lista estructurada buscando el dato y devuelve el nivel en que se encuentra. Si el dato no se encuentra en la lista, se devolverá -1. Si el dato se encuentra en más de un lugar de la lista se devolverá el nivel mayor.

Ejemplos:

```
(nivel-hoja 'b '(a b (c))) ; => 1
(nivel-hoja 'b '(a (b) c)) ; => 2
(nivel-hoja 'b '(a (b) d ((b)))) ; => 3
(nivel-hoja 'b '(a c d ((e)))) ; => -1
```

Solución recursiva:

```
(define (nivel-hoja dato elem)
  (cond
    ((null? elem) -1)
    ((hoja? elem) (if (equal? elem dato) 0 -1))
    (else (max (suma-1-si-mayor-igual-que-0
                 (nivel-hoja dato (car elem)))
                (nivel-hoja dato (cdr elem))))))
```

La función auxiliar se define de la siguiente forma:

```
(define (suma-1-si-mayor-igual-que-0 x)
  (if (>= x 0)
      (+ x 1)
      x))
```

Con funciones de orden superior:

```
(define (nivel-hoja-fos dato lista)
  (suma-1-si-mayor-igual-que-0
    (foldr max
           -1
           (map (lambda (elem)
                  (if (hoja? elem)
                      (if (equal? elem dato) 0 -1)
                      (nivel-hoja-fos dato elem)))
                 lista))))
```

Se puede hacer otra versión de la función anterior, que aproveche que Scheme es un lenguaje débilmente tipado y que devuelva **#f** si el dato no se encuentra:

```
(define (incrementa-nivel-si-encontrado x)
  (if (not x)
      x
      (+ x 1)))
```

```
(define (mayor-nivel-si-encontrado x y)
  (cond
    ((not x) y)
    ((not y) x)
    (else (max x y)))))

(define (nivel-hoja2 dato elem)
  (cond
    ((null? elem) #f)
    ((hoja? elem) (if (equal? x dato) 0 #f))
    (else (mayor-nivel-si-encontrado
            (incrementa-nivel-si-encontrado (nivel-hoja dato (car elem)))
            (nivel-hoja2 dato (cdr elem))))))
```

Las funciones auxiliares `incrementa-nivel-si-encontrado` y `mayor-nivel-si-encontrado` realizan la suma de 1 y calculan el máximo tratando con los casos en los que alguno de los argumentos es `#f` porque no se ha encontrado el dato.

Ejemplos:

```
(nivel-hoja2 'b '(a b (c))) ; => 1
(nivel-hoja2 'b '(a (b) c)) ; => 2
(nivel-hoja2 'b '(a (b) d ((b)))) ; => 3
(nivel-hoja2 'b '(a c d ((e)))) ; => #f
```

**(cuadrado-estruct lista)**

Vamos ahora a ver un tipo de función distinta. Una que construye una lista estructurada y la devuelve.

Queremos implementar la función **(cuadrado-estruct lista)** que recibe una lista estructurada y devuelve otra lista estructurada con la misma estructura y sus números elevados al cuadrado.

Por ejemplo:

```
(cuadrado-estruct '(2 3 (4 (5)))) ; => (4 9 (16 (25)))
```

La solución recursiva es:

```
(define (cuadrado-estruct elem)
  (cond ((null? elem) '())
        ((hoja? elem) (* elem elem))
        (else (cons (cuadrado-estruct (car elem))
                    (cuadrado-estruct (cdr elem)))))))
```

Se llama a la recursión con el `car` y con el `cdr` de la lista original. El resultado de ambas llamadas serán las correspondientes listas estructuradas con sus elementos elevados al cuadrado. Y se devuelve la lista resultante de insertar la lista devuelta en la llamada recursiva con el `car` en la primera posición de la lista devuelta en la llamada recursiva con el `cdr`.

Es muy interesante la versión de esta función con funciones de orden superior:

```
(define (cuadrado-estruct-fos lista)
  (map (lambda (elem)
    (if (hoja? elem)
        (* elem elem)
        (cuadrado-estruct-fos elem))) lista))
```

Como una lista estructurada está compuesta de datos o de otras sublistas podemos aplicar `map` para que devuelva la lista resultante de transformar la original con la función que le pasamos como parámetro.

`(map-estruct f lista)`

Podemos generalizar la función anterior y definir la función de orden superior sobre listas estructuradas `(map-estructurada f lista)` que devuelve una lista estructurada igual que la original con el resultado de aplicar a cada uno de sus hojas la función `f`

Por ejemplo:

```
(map-estruct (lambda (x) (* x x)) '(2 3 (4 (5)))) ; => (4 9 (16 (25)))
```

La solución recursiva es una generalización de la función anterior, usando el parámetro `f`:

```
(define (map-estruct f elem)
  (cond ((null? elem) '())
        ((hoja? elem) (f elem))
        (else (cons (map-estruct f (car elem))
                    (map-estruct f (cdr elem))))))
```

Solución con `map`:

```
(define (map-estruct-fos f lista)
  (map (lambda (elem)
    (if (hoja? elem)
        (f elem)
        (map-estruct-fos f elem))) lista))
```

## Definición de árboles en Scheme

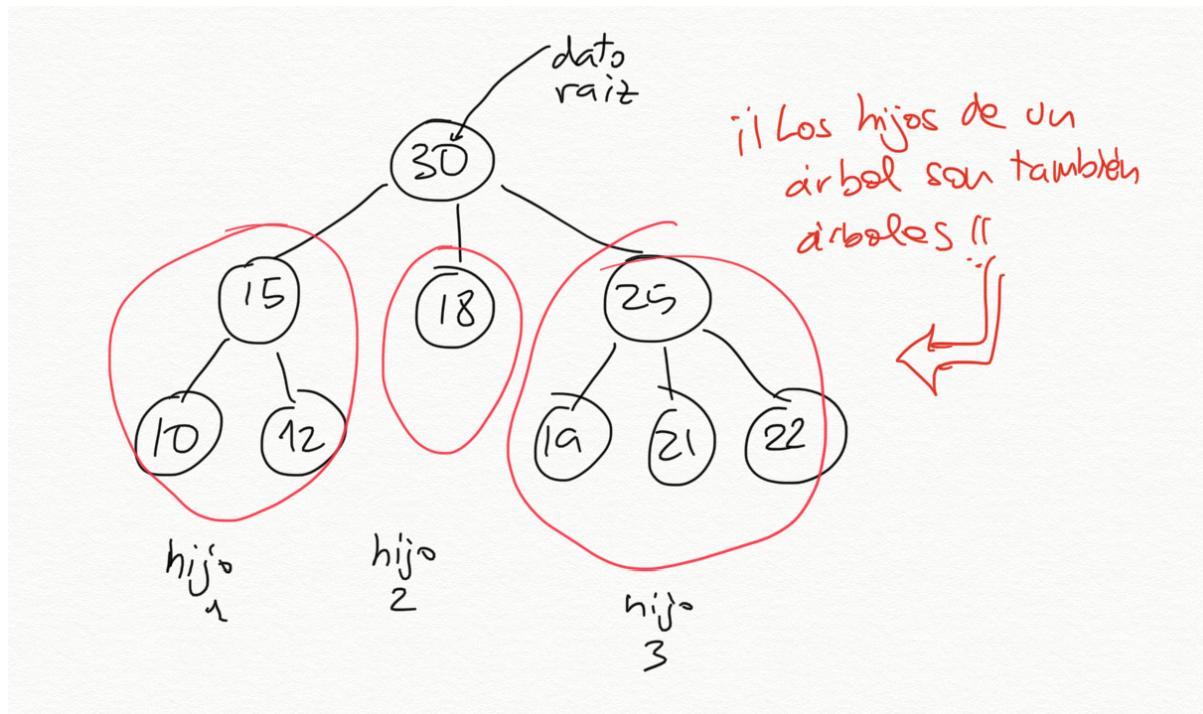
### Definición de árbol

Un **árbol** es una estructura de datos definida por un valor raíz, que es el padre de toda la estructura, del que salen otros subárboles hijos ([Wikipedia](#)).

Un **árbol** se puede definir recursivamente de la siguiente forma:

- Una colección de un **dato** (el valor de la raíz del árbol) y una **lista de hijos** que también son árboles.
- Una **hoja** será un árbol sin hijos (un dato con una lista de hijos vacía).

Un ejemplo de árbol:



El árbol anterior tiene como dato raíz el número 30 y tiene 3 árboles hijos:

- El primer hijo es un árbol con raíz 15 y con dos hijos hoja, el 10 y el 12
- El segundo hijo es un árbol hoja, con valor 18
- El tercer hijo es un árbol con raíz 25 y con tres hijos hoja, el 19, 21 y 22.

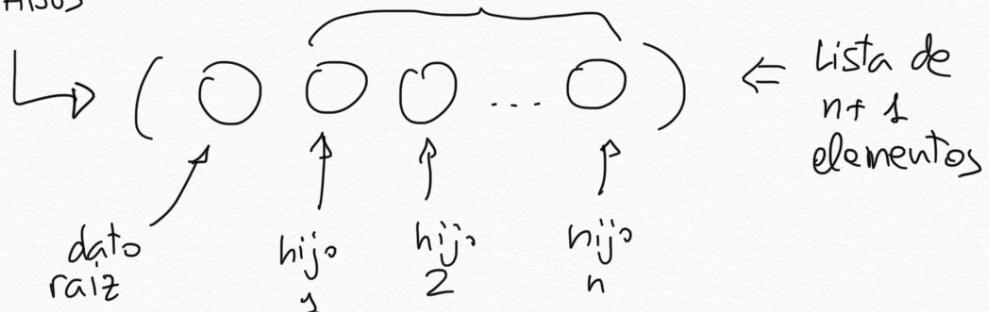
### Representación de árboles con listas

En Scheme la lista es la estructura de datos principal. ¿Cómo podemos construir un árbol usando listas?

Podemos hacerlo de varias formas, pero escogemos la siguiente: usar **una lista de  $n+1$  elementos** para representar un árbol con  $n$  hijos:

REPRESENTACIÓN  
EN FORMA DE LISTA  
DE UN ÁRBOL GENÉRICO  
DE  $n$  HIJOS

árboles hijos



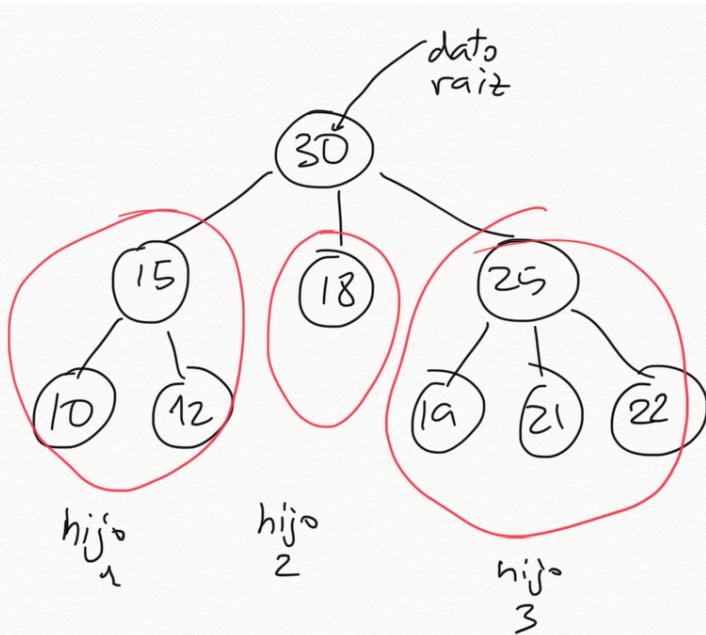
- el primer elemento la lista será el dato de la raíz
- el resto serán los árboles hijos

arbol -> (dato hijo-1 hijo-2 ... hijo-n)

Los nodos hoja (datos al final del árbol que no tienen ningún hijo) son también árboles. Al no tener hijos, se representan como listas con un único elemento, el propio dato.

Nodo hoja -> (dato)

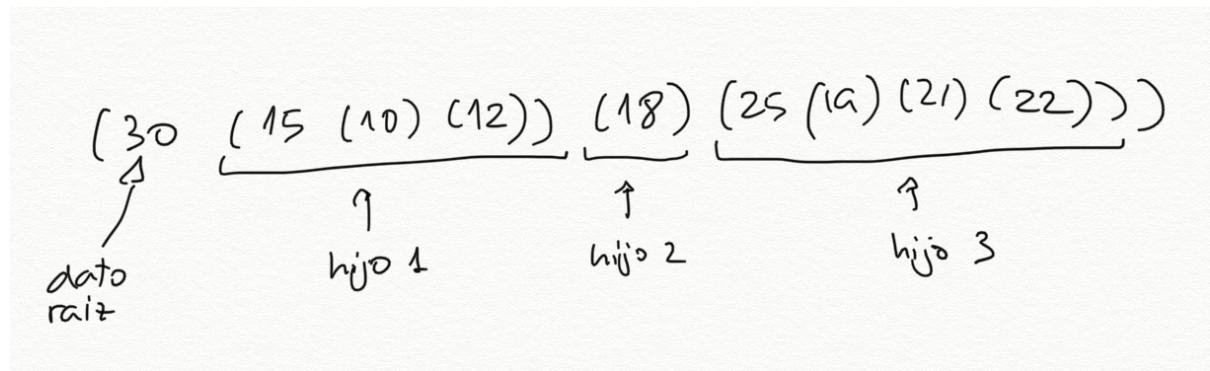
La forma de representar el árbol anterior



será la siguiente lista:

```
(30 (15 (10) (12)) (18) (25 (19) (21) (22)))
```

Los elementos de esta lista son:

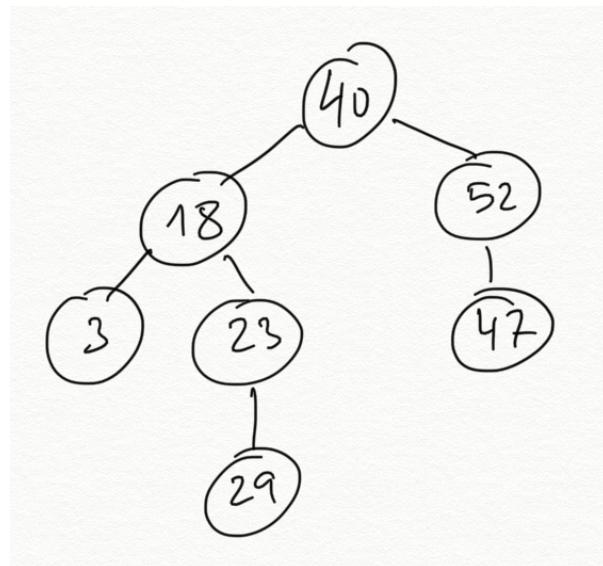


- El primer elemento es el número **30**, el dato valor de la raíz del árbol
- El segundo elemento es la lista **(15 (10) (12))**, que representa el árbol con dato **15** y dos hijos
- El tercer elemento es la lista **(18)** que representa el árbol hoja formado por un **18**
- El cuarto elemento es la lista **(25 (19) (21) (22))**, que representa el árbol con un dato **25** y tres hijos

Podríamos definir el árbol con la siguiente sentencia:

```
(define arbol1 '(30 (15 (10) (12)) (18) (25 (19) (21) (22))))
```

Otro ejemplo más. ¿Cómo se implementa en Scheme el árbol de la siguiente figura?



Se haría con la lista de la siguiente sentencia:

```
(define arbol2 '(40 (18 (3) (23 (29))) (52 (47))))
```

## Barrera de abstracción

Una vez definida la forma de representar árboles, vamos a definir las funciones básicas para manejarlos. Veremos las funciones para obtener el dato y los hijos y la función para construir un árbol nuevo. Estas funciones proporcionan lo que se denomina *barrera de abstracción* del tipo datos *árbol*.

En todos los nombres de las funciones de la barrera de abstracción añadimos el sufijo **-arbol**.

Definimos dos conjuntos de funciones: **constructores** para construir un nuevo árbol y **selectores** para obtener los elementos del árbol. Vamos a empezar por los selectores.

## Selectores

Funciones que obtienen los elementos de un árbol:

```
(define (dato-arbol arbol)
  (car arbol))

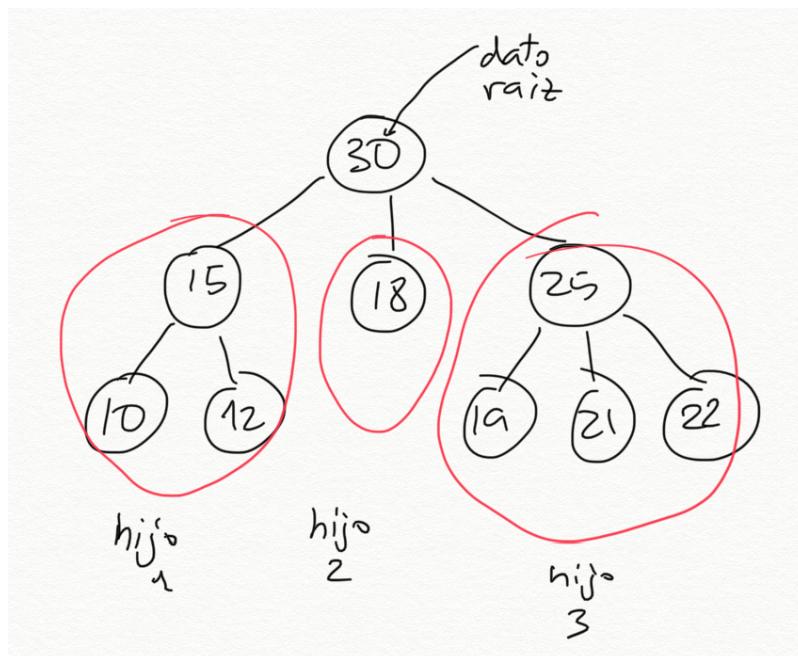
(define (hijos-arbol arbol)
  (cdr arbol))

(define (hoja-arbol? arbol)
  (null? (hijos-arbol arbol)))
```

Es importante tener claro los tipos devueltos por las dos primeras funciones:

- (**dato-arbol arbol**): devuelve **el dato** de la raíz del árbol.
- (**hijos-arbol arbol**): devuelve **una lista de árboles** hijos. En algunas ocasiones llamaremos *bosque* a una lista de árboles. Podremos recorrer esa lista usando las funciones **car** y **cdr** para obtener los árboles hijos.

Volvemos a mostrar el **arbol1** para comprobar estas funciones.



Las funciones anteriores devuelven los siguientes valores:

```
(dato-arbol arbol1) ; => 30
(hijos-arbol arbol1) ; => ((15 (10) (12)) (18) (25 (19) (21) (22)))
(hoja-arbol? (car (hijos-arbol arbol1))) ; => #f
(hoja-arbol? (cadr (hijos-arbol arbol1))) ; => #t
```

- La llamada `(dato-arbol arbol1)` devuelve el dato que hay en la raíz del árbol, el número `30`.
- La invocación `(hijos-arbol arbol1)` devuelve una lista de tres elementos, los árboles hijos:
  - El primer elemento es la lista `(15 (10) (12))`, que representa el árbol formado por el `15` en su raíz y las hojas `10` y `12`.
  - El segundo elemento es el árbol hoja `18`, representado por la lista `(18)`.
  - El tercero es la lista `(25 (19) (21) (22))`, que representa el árbol formado por el `25` en su raíz y las hojas `19`, `21` y `22`.

Es muy importante considerar en cada caso con qué tipo de dato estamos trabajando y usar la barrera de abstracción adecuada en cada caso:

- La función `hijos-arbol` siempre devuelve una **lista de árboles**, que podemos recorrer usando `car` y `cdr`.
- El `car` de una lista de árboles (devuelta por `hijos-arbol`) siempre es un árbol y debemos de usar las funciones de su barrera de abstracción: `dato-arbol` e `hijos-arbol`.
- La función `dato-arbol` devuelve un **dato**, del tipo que guardemos en el árbol. En el caso del árbol ejemplo es un número.

Por ejemplo, para obtener el número `12` en el árbol anterior tendríamos que hacer lo siguiente: acceder al primer elemento de la lista de hijos, después al segundo hijo de éste y por último acceder a su dato.

Recordemos que `hijos-arbol` devuelve la lista de árboles hijos, por lo que utilizaremos las funciones `car` y `cdr` para recorrerlas y obtener los elementos que nos interesen:

```
(dato-arbol (cadr (hijos-arbol (car (hijos-arbol arbol1))))))
; => 12
```

## Constructor

Definimos una función constructora que abstraiga la construcción de un árbol y encapsula su implementación concreta. Para construir un árbol necesitamos un dato y una lista de árboles hijos. Si la lista de árboles hijos es vacía, tendremos un nodo hoja.

```
(define (nuevo-arbol dato lista-arboles)
  (cons dato lista-arboles))
```

Llamaremos a la función `nuevo-arbol` pasando su dato (obligatorio) y la lista de árboles hijos. Si se pasa una lista vacía como parámetro estaremos definiendo un nodo hoja.

Por ejemplo, para definir un nodo hoja con el dato 2:

```
(define arbol3 (nuevo-arbol 2 '()))
```

Y para definir un árbol con 3 hijos:

```
(define arbol4 (nuevo-arbol 10 (list (nuevo-arbol 2 '())
   (nuevo-arbol 5 '())
   (nuevo-arbol 9 '()))))
```

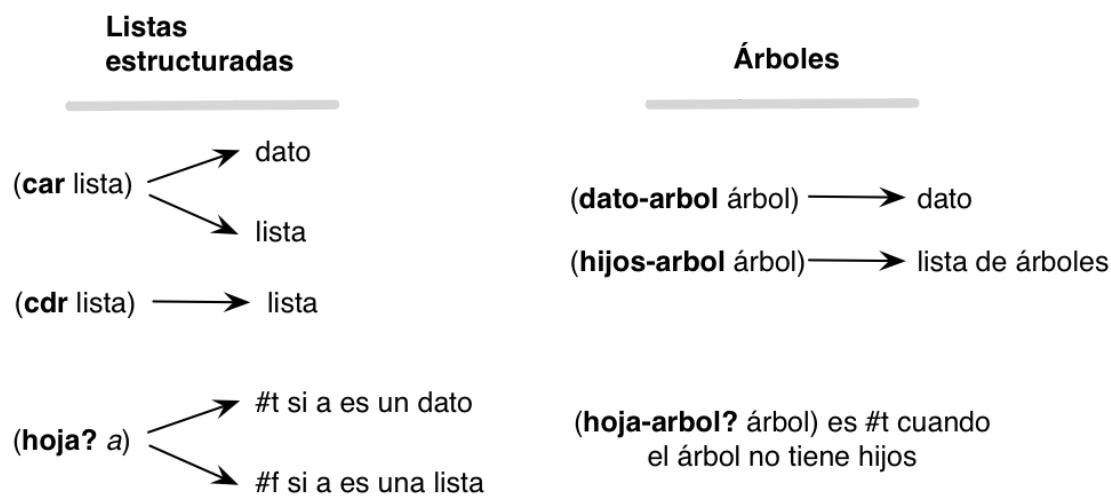
El árbol 1 anterior se puede construir con las siguientes llamadas al constructor. Guardamos los árboles hijos en variables auxiliares para hacer más entendible la expresión:

```
(define arbol-15 (nuevo-arbol 15 (list (nuevo-arbol 10 '())
   (nuevo-arbol 12 '()))))
(define arbol-18 (nuevo-arbol 18 '()))
(define arbol-25 (nuevo-arbol 25 (list (nuevo-arbol 19 '())
   (nuevo-arbol 21 '())
   (nuevo-arbol 22 '()))))
(define arbol1b (nuevo-arbol 30 (list arbol-15 arbol-18 arbol-25)))
arbol1b ; => (30 (15 (10) (12)) (18) (25 (19) (21) (22)))
```

## Barreras de abstracción de árboles y listas estructuradas

Es importante diferenciar la barrera de abstracción de los árboles de la de las listas estructuradas. Aunque un árbol se implementa en Scheme con una lista estructurada, a la hora de definir funciones sobre árboles hay que trabajar con las funciones definidas arriba.

El siguiente esquema resumen las características de los selectores de la barrera de abstracción de listas y árboles:



!!! Important "Importante" Debemos usar la barrera de abstracción al trabajar con árboles porque así separamos nuestro código de la implementación subyacente del tipo de dato. De esta forma es posible

cambiar la implementación del tipo de dato sin afectar a las funciones que hemos definido usando la barrera. Lo único que hay que hacer es cambiar la implementación de la barrera de abstracción.

Otras ventajas de utilizar la barrera de abstracción, tan importantes como la anterior, son:

- El código es mucho más legible. Dado que Scheme es un lenguaje débilmente tipado, en una expresión como `(dato-arbol elem)` sabemos que el elemento sobre el que se trabaja es un árbol (no es un número, ni un string, ni un booleano).
- El código es trasladable a cualquier lenguaje de programación. Si queremos trabajar con árboles en JavaScript, por ejemplo, sólo tendremos que implementar la barrera de abstracción en este lenguaje. Una vez hecho eso todas las funciones que trabajan con árboles, como las que veremos a continuación, funcionarán correctamente.

## Funciones recursivas sobre árboles

Vamos a diseñar las siguientes funciones recursivas:

- (`suma-datos-arbol arbol`): devuelve la suma de todos los nodos
- (`to-list-arbol arbol`): devuelve una lista con los datos del árbol
- (`cuadrado-arbol arbol`): eleva al cuadrado todos los datos de un árbol manteniendo la estructura del árbol original
- (`map-arbol f arbol`): devuelve un árbol con la estructura del árbol original aplicando la función f a subdatos.
- (`altura-arbol arbol`): devuelve la altura de un árbol

Todas comparten un patrón similar de recursión mutua.

### Función `suma-datos-arbol`

Vamos a implementar una función recursiva que sume todos los datos de un árbol.

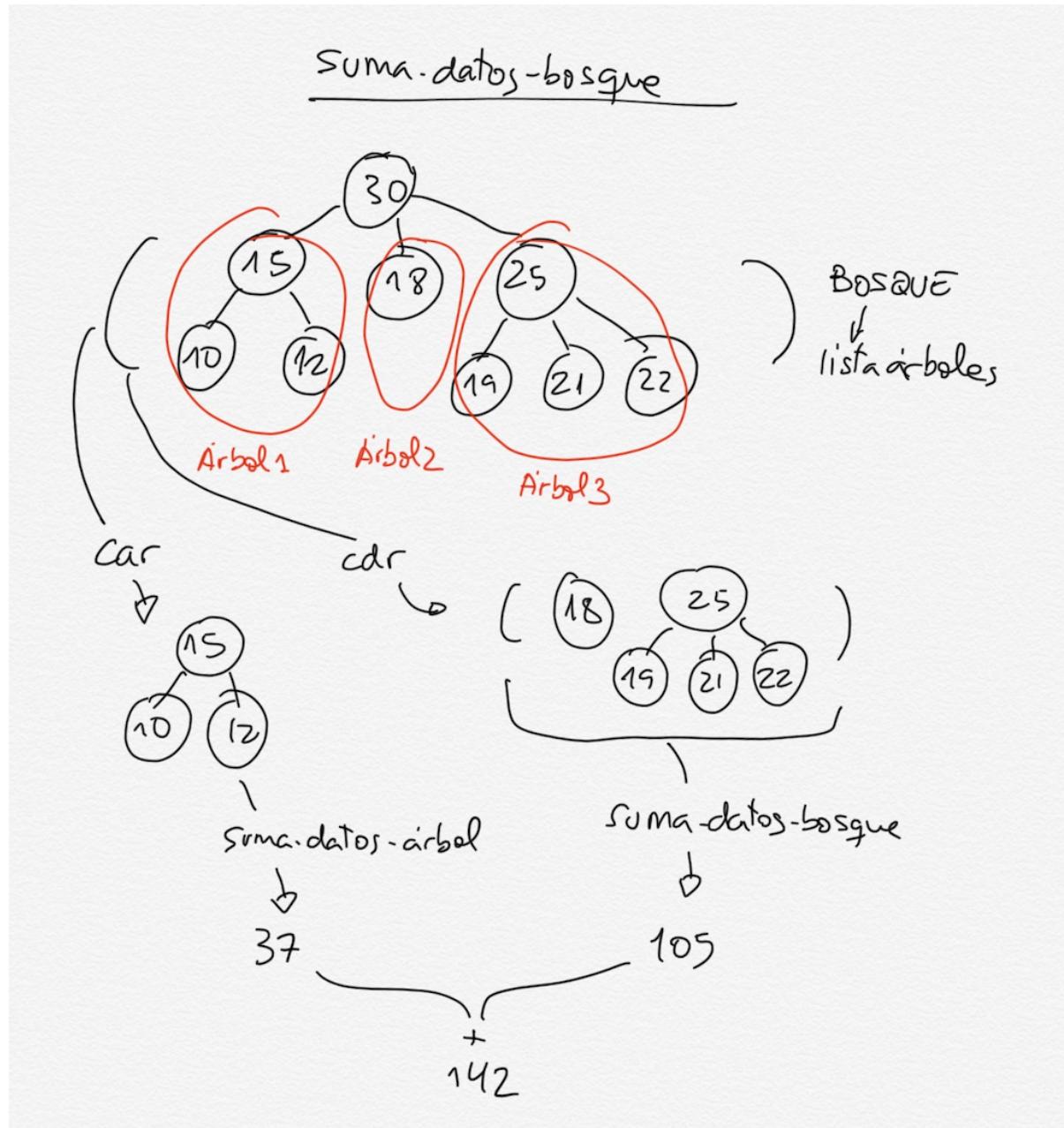
Un árbol siempre va a tener un dato y una lista de hijos (que puede ser vacía) que obtenemos con las funciones `dato-arbol` e `hijos-arbol`. Podemos plantear entonces el problema de sumar los datos de un árbol como la suma del dato de su raíz y lo que devuelva la llamada a una función auxiliar que sume los datos de su lista de hijos (llamamos *bosque* a una lista de hijos):

```
(define (suma-datos-arbol arbol)
  (+ (dato-arbol arbol)
      (suma-datos-bosque (hijos-arbol arbol))))
```

Esta función suma los datos de **un** árbol. La podemos utilizar entonces para construir la siguiente función que suma una lista de árboles:

```
(define (suma-datos-bosque bosque)
  (if (null? bosque)
      0
      (+ (suma-datos-arbol (car bosque)) (suma-datos-bosque (cdr bosque)))))
```

Podemos visualizar el funcionamiento de la `suma-datos-bosque` en la siguiente figura:



El caso general de la función obtiene el primer árbol de la lista (un árbol) y llama a la función `suma-datos-arbol` para obtener la suma de sus datos. También obtiene el resto del bosque (otra lista de árboles) y llama de forma recursiva a la propia función para sumar todos sus árboles.

Tenemos una **recursión mutua**: para sumar los datos de una lista de árboles llamamos a la suma de un árbol individual que a su vez llama a la suma de sus hijos, etc. La recursión termina cuando calculamos la suma de un árbol hoja. Entonces se pasa a `suma-datos-bosque` una lista vacía y ésta devolverá 0.

```
(suma-datos-arbol arbol1) ; => 172
```

### Versión alternativa con funciones de orden superior

Al igual que hicimos con las listas estructuradas, es posible conseguir una versión más concisa y elegante utilizando funciones de orden superior:

```
(define (suma-datos-arbol-fos arbol)
  (foldr +
         (dato-arbol arbol)
         (map (lambda (subarbol)
                  (suma-datos-arbol-fos subarbol)) (hijos-arbol arbol))))
```

La función `map` aplica la propia función que estamos definiendo (`suma-datos-arbol-fos`) a cada uno de los árboles hijos (obtenidos con la función `(hijos-arbol arbol)`). Confirando en que la función hace su trabajo, devolverá para cada arbol hijo la suma de todos sus nodos. De esta forma, el resultado de `map` será una lista con la suma de los nodos de todos los árboles hijos.

La función `foldr` suma todos esos números de la lista y el número de la raíz.

La expresión lambda de `map` no es necesaria, ya que la función `suma-datos-arbol-fos` tiene un único argumento que va a ser cada uno de los árboles hijos:

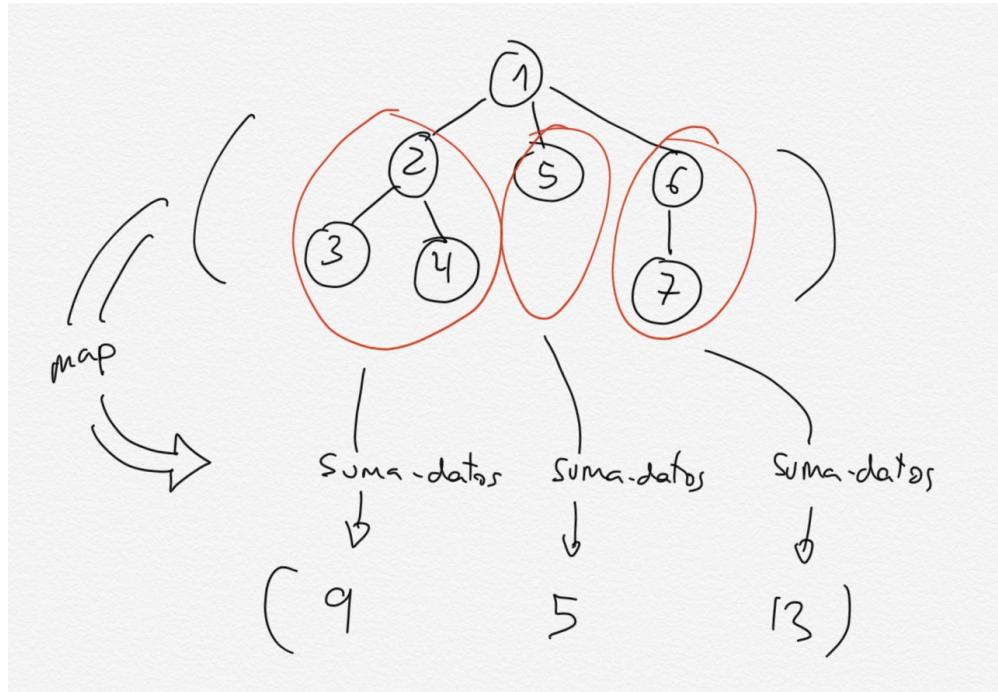
```
(define (suma-datos-arbol-fos arbol)
  (foldr +
         (dato-arbol arbol)
         (map suma-datos-arbol-fos (hijos-arbol arbol))))
```

Un ejemplo de su funcionamiento sería el siguiente:

```
(suma-datos-arbol-fos '(1 (2 (3) (4)) (5) (6 (7)))) =>
(foldr +
      1
      (map suma-datos-arbol-fos '((2 (3) (4))
                                    (5)
                                    (6 (7))))) =>
(foldr + 1 '(9 5 13)) =>
28
```

- El árbol que queremos sumar tiene un 1 en la raíz y tres hijos: `(2 (3) (4))`, `(5)` y `(6 (7))`.
- La aplicación de `map suma-datos-arbol-fos` sobre la lista de hijos devuelve una lista con la suma de los nodos de cada hijo: `(9 5 13)`.
- La función `foldr` suma esa lista y el valor del nodo raíz (`1`).

Podemos visualizar gráficamente el funcionamiento del `map` con la siguiente figura:



### Función `to-list-arbol`

Queremos diseñar una función (`to-list-arbol arbol`) que devuelva una lista con los datos del árbol en un recorrido *preorden* (primero el dato de la raíz y después el dato de sus hijos de izquierda a derecha).

La solución, siguiendo el patrón visto en `suma-datos`, es la siguiente.

```

(define (to-list-arbol arbol)
  (cons (dato-arbol arbol)
        (to-list-bosque (hijos-arbol arbol)))

(define (to-list-bosque bosque)
  (if (null? bosque)
      '()
      (append (to-list-arbol (car bosque))
              (to-list-bosque (cdr bosque)))))

  
```

Igual que antes, la función utiliza una *recursión mutua*: para listar todos los nodos, añadimos el dato a la lista de nodos que nos devuelve la función `to-list-bosque`. Esta función coge una lista de árboles (un *bosque*) y devuelve la lista *preorden* de sus nodos. Para ello, concatena la lista de los nodos de su primer elemento (el primer árbol) a la lista de nodos del resto de árboles (que devuelve la llamada recursiva).

Ejemplo:

```

(to-list-arbol '(* (+ (5) (* (2) (3)) (10)) (- (12))))
; => (* + 5 * 2 3 10 - 12)
  
```

Una definición alternativa usando funciones de orden superior:

```
(define (to-list-arbol-fos arbol)
  (cons (dato-arbol arbol)
        (foldr append '() (map to-list-arbol-fos (hijos-arbol arbol)))))
```

Esta versión es muy elegante y concisa. Usa la función `map` que aplica una función a los elementos de una lista y devuelve la lista resultante. Como lo que devuelve (`hijos-arbol arbol`) es precisamente una lista de árboles podemos aplicar a sus elementos cualquier función definida sobre árboles. Incluso la propia función que estamos definiendo (¡confía en la recursión!).

### Función cuadrado-arbol

Veamos ahora la función (`cuadrado-arbol arbol`) que toma un árbol de números y devuelve un árbol con la misma estructura y sus datos elevados al cuadrado:

```
(define (cuadrado-arbol arbol)
  (nuevo-arbol (cuadrado (dato-arbol arbol))
                (cuadrado-bosque (hijos-arbol arbol))))  
  

(define (cuadrado-bosque bosque)
  (if (null? bosque)
      '()
      (cons (cuadrado-arbol (car bosque))
            (cuadrado-bosque (cdr bosque)))))
```

Ejemplo:

```
(cuadrado-arbol '(2 (3 (4) (5)) (6)))
; => (4 (9 (16) (25)) (36))
```

Versión 2, con la función de orden superior `map`:

```
(define (cuadrado-arbol-fos arbol)
  (nuevo-arbol (cuadrado (dato-arbol arbol))
                (map cuadrado-arbol-fos (hijos-arbol arbol))))
```

### Función map-arbol

La función `map-arbol` es una función de orden superior que generaliza la función anterior. Definimos un parámetro adicional en el que se pasa la función a aplicar a los elementos del árbol.

```
(define (map-arbol f arbol)
  (nuevo-arbol (f (dato-arbol arbol))
    (map-bosque f (hijos-arbol arbol)))))

(define (map-bosque f bosque)
  (if (null? bosque)
    '()
    (cons (map-arbol f (car bosque))
      (map-bosque f (cdr bosque)))))
```

Ejemplos:

```
(map-arbol cuadrado '(2 (3 (4) (5)) (6)))
; => (4 (9 (16) (25)) (36))
(map-arbol (lambda (x) (+ x 1)) '(2 (3 (4) (5)) (6)))
; => (3 (4 (5) (6)) (7))
```

Con **map**:

```
(define (map-arbol-fos f arbol)
  (nuevo-arbol (f (dato-arbol arbol))
    (map (lambda (x)
      (map-arbol-fos f x)) (hijos-arbol arbol))))
```

## Función altura-arbol

Vamos por último a definir una función que devuelve la altura de un árbol.

Recordemos las siguientes definiciones relacionadas con los árboles:

- Longitud de un camino entre dos nodos: número de aristas.
- Altura de un nodo: longitud del camino más largo del nodo a una hoja.
- Profundidad de un nodo: longitud del camino de la raíz al nodo.
- Profundidad de un árbol: profundidad del nodo más profundo.
- Nivel de un nodo: número de predecesores.
- Altura de árbol: altura de la raíz.

Podemos implementar la altura de una forma similar a como hicimos con las listas estructuradas: calculamos la altura de los árboles hijos, nos quedamos con la mayor, y sumamos 1 para añadir la arista del camino de la raíz al hijo.

La mayor altura de los hijos la calculamos con la función **altura-bosque**.

```
(define (altura-arbol arbol)
  (if (hoja-arbol? arbol)
```

```

0
(+ 1 (altura-bosque (hijos-arbol arbol)))))

(define (altura-bosque bosque)
  (if (null? bosque)
    0
    (max (altura-arbol (car bosque))
          (altura-bosque (cdr bosque)))))


```

Ejemplos:

```

(altura-arbol '(2)) ; => 0
(altura-arbol '(4 (9 (16) (25)) (36))) ; => 2

```

La solución con funciones de orden superior es algo distinta de la que vimos con listas estructuradas:

```

(define (altura-arbol-fos arbol)
  (if (hoja-arbol? arbol)
    0
    (+ 1 (foldr max
                 0
                 (map altura-arbol-fos (hijos-arbol arbol))))))


```

La función `map` mapea sobre los árboles hijos la propia función, que calcula la altura de cada hijo (será uno menos que la altura del padre, 0 si se trata de una hoja).

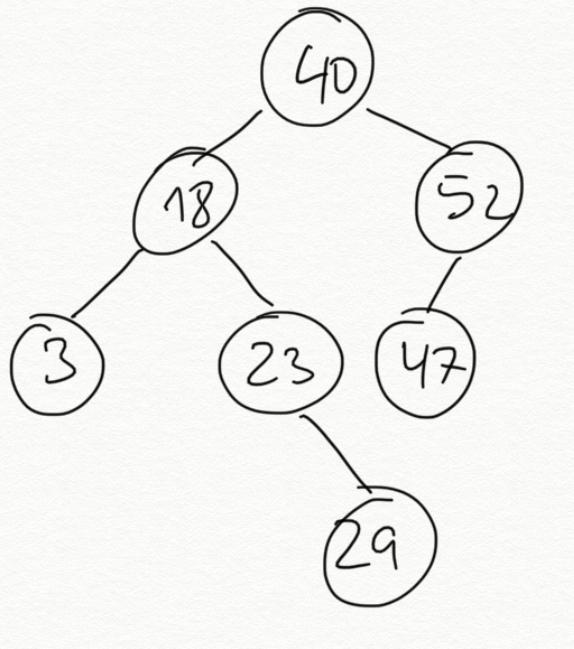
La función `map` devuelve entonces una lista altura de los hijos, de la que obtenemos el máximo plegando la lista con la función `max`.

Por último sumamos 1 para devolver la altura del árbol completo (un nivel más que el nivel máximo de los hijos).

## Arboles binarios

Definición de árboles binarios en Scheme

Los árboles binarios son árboles cuyos nodos tienen 0, 1 o 2 hijos. Por ejemplo, el árbol mostrado en la siguiente figura es un árbol binario.



A diferencia de los árboles genéricos vistos anteriormente un árbol binario no puede tener más de dos hijos.

Los representaremos en Scheme utilizando una lista de tres elementos:

- Dato
- Hijo izquierdo (otro árbol binario)
- Hijo derecho (otro árbol binario)

En el caso en que no exista el hijo izquierdo o el derecho (o ambos) utilizaremos una lista vacía para indicar un nodo vacío.

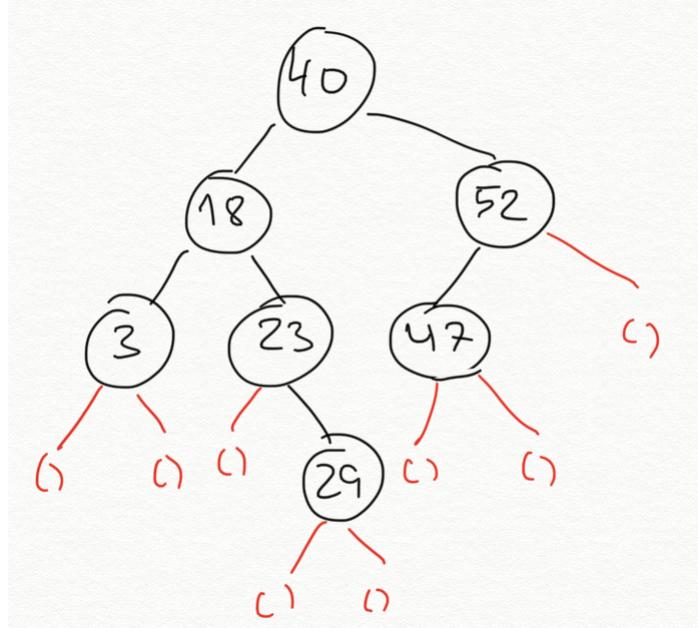
De esta manera, un nodo hoja con el dato 10 se representará en Scheme con la lista:

```
(10 () ())
```

Por ejemplo, representamos el árbol de la figura anterior con la siguiente lista:

```
(40 (18 (3 () ()))
      (23 ())
      (29 () (())))
    (52 (47 () ()))
    ()))
```

Visualmente lo podemos representar de la siguiente forma. La no existencia de un hijo izquierdo o un hijo derecho se representa por una lista vacía.



## Barrera de abstracción

Definimos la siguiente barrera de abstracción para los árboles binarios. Terminamos todos los nombres de las funciones con el sufijo **-arbolb** (árbol binario).

### Selectores

Los selectores de la barrera de abstracción del árbol binario son los siguientes.

```

(define (dato-arbolb arbol)
  (car arbol))

(define (hijo-izq-arbolb arbol)
  (cadr arbol))

(define (hijo-der-arbolb arbol)
  (caddr arbol))

(define arbolb-vacio '())

(define (vacio-arbolb? arbol)
  (equal? arbol arbolb-vacio))

(define (hoja-arbolb? arbol)
  (and (vacio-arbolb? (hijo-izq-arbolb arbol))
       (vacio-arbolb? (hijo-der-arbolb arbol))))
  
```

Como parte de la barrera de abstracción definimos la constante **arbolb-vacio**, que toma el valor de una lista vacía.

### Constructor

```
(define (nuevo-arbolb dato hijo-izq hijo-der)
  (list dato hijo-izq hijo-der))
```

Por ejemplo, para construir un árbol con 10 en la raíz y 8 en su hijo izquierdo y 15 en su derecho utilizando el constructor de la barrera de abstracción:

```
(define arbolb1
  (nuevo-arbolb 10 (nuevo-arbolb 8 arbolb-vacio arbolb-vacio)
    (nuevo-arbolb 15 arbolb-vacio arbolb-vacio)))
```

Otro ejemplo, el árbol binario de la figura anterior utilizando el constructor de la barrera de abstracción:

```
(define arbolb2
  (nuevo-arbolb 40
    (nuevo-arbolb 18
      (nuevo-arbolb 3 arbolb-vacio arbolb-vacio)
      (nuevo-arbolb 23
        arbolb-vacio
        (nuevo-arbolb 29
          arbolb-vacio
          arbolb-vacio)))
    (nuevo-arbolb 52
      (nuevo-arbolb 47 arbolb-vacio arbolb-vacio)
      arbolb-vacio)))
```

## Funciones recursivas sobre árboles binarios

Veamos las siguientes funciones recursivas sobre árboles binarios:

- (**suma-datos-arbolb arbol**): devuelve la suma de todos los nodos
- (**to-list-arbolb arbol**): devuelve una lista con los datos del árbol
- (**cuadrado-arbolb arbol**): eleva al cuadrado todos los datos de un árbol manteniendo la estructura del árbol original

Estas funciones utilizan una mezcla de los patrones usados en la recursión para trabajar con árboles genéricos y la recursión para trabajar con listas estructuradas. Tenemos un dato en la raíz, que tenemos que combinar con lo que devuelve la recursión aplicada sobre el hijo izquierdo y lo que devuelve la recursión aplicada sobre el hijo derecho.

### **suma-datos-arbolb**

```
(define (suma-datos-arbolb arbol)
  (if (vacío-arbolb? arbol)
    0
    (+ (dato-arbolb arbol)
```

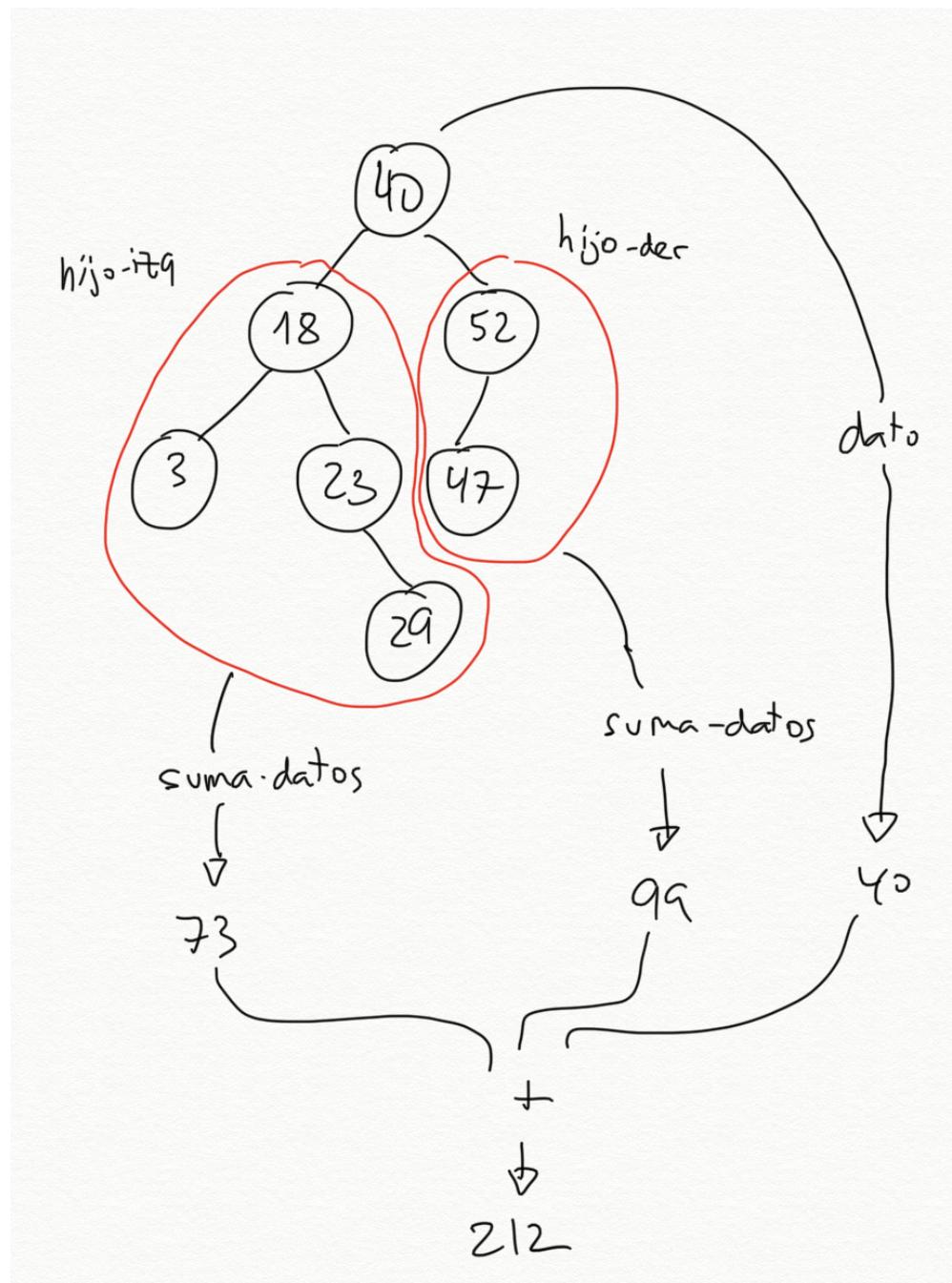
```
(suma-datos-arbolb (hijo-izq-arbolb arbol))
(suma-datos-arbolb (hijo-der-arbolb arbol))))
```

```
(suma-datos-arbolb arbolb2) ; => 212
```

Como el hijo izquierdo y el hijo derecho son también árboles binarios, podemos llamar a la recursión con esos árboles. Esas llamadas recursivas nos devolverán la suma de los datos en cada subárbol. Y sumamos el dato de la raíz.

Para definir el caso base, podemos ver que en cada llamada recursiva vamos obteniendo el hijo izquierdo y el hijo derecho. Al final llegaremos a un árbol vacío, en cuyo caso devolvemos 0.

La siguiente figura representa el funcionamiento del caso general.



## to-list-arbolb

La función `to-list-arbolb` es similar a la vista con los árboles genéricos. Recibe un árbol binario y devuelve una lista con los datos en un recorrido preorder.

```
(define (to-list-arbolb arbol)
  (if (vacio-arbolb? arbol)
      '()
      (cons (dato-arbolb arbol)
            (append (to-list-arbolb (hijo-izq-arbolb arbol))
                    (to-list-arbolb (hijo-der-arbolb arbol))))))

(to-list-arbolb arbolb2) ; => (40 18 3 23 29 52 47)
```

El funcionamiento es similar a la suma: llamamos a la recursión por la izquierda y por la derecha. El resultado de las llamadas recursivas serán dos listas que tenemos que concatenar con `append`. Y por últimos añadimos en cabeza el dato de la raíz con `cons`.

### **cuadrado-arbolb**

Por último, la función `cuadrado-arbolb` construye un nuevo árbol binario elevando al cuadrado el dato de la raíz, su hijo izquierdo y su hijo derecho. Para construir el árbol binario llamamos al constructor `nuevo-arbolb`.

```
(define (cuadrado-arbolb arbol)
  (if (vacio-arbolb? arbol)
      arbolb-vacio
      (nuevo-arbolb (cuadrado (dato-arbolb arbol))
                    (cuadrado-arbolb (hijo-izq-arbolb arbol))
                    (cuadrado-arbolb (hijo-der-arbolb arbol)))))

(cuadrado-arbolb arbolb1) ; => (100 (64 () ()) (225 () ()))
```

## Bibliografía - SICP

En este tema explicamos conceptos de los siguientes capítulos del libro *Structure and Interpretation of Computer Programs*:

- [2.2.2 - Hierarchical Structures](#)

Lenguajes y Paradigmas de Programación, curso 2019-20

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante  
Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez

## 6. Ámbitos de variables, let y closures

Ahora que hemos introducido la forma especial `lambda` y la idea de que una función es un objeto de primera clase, podemos revisar el concepto de ámbito de variables en Scheme e introducir un importante concepto: clausura (`closure` en inglés).

## 6.1. Ámbitos de variables

El concepto del **ámbito** de vida de las variables es un concepto fundamental en los lenguajes de programación. En inglés se utiliza el término *scope*.

Cuando se define una variable, asociándole un valor, esta asociación tiene una extensión determinada, ya sea en términos de tiempo de compilación (**ámbito léxico**) como en términos de tiempo de ejecución (**ámbito dinámico**). El ámbito de una variable determina cuándo podemos referirnos a ella para recuperar el valor asociado.

Al conjunto de variables disponibles en una parte del programa o en una parte de su ejecución se denomina **contexto** o **entorno** (*context* o *environment*).

### Variables de ámbito global

Una variable definida en el programa con la instrucción **define** tiene un ámbito global.

```
(define a "hola")
(define b (string-append "adios" a))
(define cuadrado (lambda (x) (* x x)))
```

Todas las variables definidas fuera de funciones forman parte del **entorno global** del programa.

### Variables de ámbito local

Como en la mayoría de lenguajes de programación, en Scheme se crea un **entorno local** cada vez que se invoca a una función. En este entorno local los argumentos de la función toman los valores de los parámetros usados en la llamada a la función. Consideramos, por tanto, estos parámetros como variables de ámbito local de la función.

Podemos usar en un entorno local una variable con el mismo nombre que en el entorno global. Cuando se ejecute el código de la función se evaluará la variable de ámbito local.

Por ejemplo, supongamos las expresiones:

```
(define x 5)
(define (suma-3 x)
  (+ x 3))

(suma-3 12)
⇒ 15
```

Cuando se ejecuta la expresión **(+ x 3)** en la invocación a **(suma-3 12)** el valor de **x** es 12, no es 5, devolviéndose 15.

Una vez realizada la invocación, desparece el entorno local y las variables locales definidas en él, recuperándose el contexto global. Por ejemplo, en la siguiente expresión, una vez realizada la invocación a

(`suma-3 12`) se devuelve el número 15 y se evalúa en el entorno global la expresión (`+ 15 x`). En este contexto la variable `x` vale 5 por lo que la expresión devuelve 20.

```
(define x 5)
(define (suma-3 x)
  (+ x 3))

(+ (suma-3 12) x)
⇒ 20
```

En el entorno local también se pueden utilizar variables definidas en el entorno global. Por ejemplo:

```
(define y 12)
(define (suma-3-bis x)
  (+ x y 3))

(suma-3-bis 5)
⇒ 20
```

La expresión (`+ x 3 y`) se evalúa en el entorno local en el que `x` vale 5. Al no estar definida la variable `y` en este entorno local, se usa su definición de ámbito global.

## 6.2 Forma especial let

### Forma especial let

En Scheme se define la forma especial let que permite crear un entorno local en el que se da valor a variables y se evalúa una expresión.

Sintaxis:

```
(let ((<var1> <exp-1>
      ...
      (<varn> <exp-n>))
     <cuerpo>)
```

Las variables `var1`, ... `varn` toman los valores devueltos por las expresiones `exp1`, ... `expn` y el cuerpo se evalúa con esos valores.

Por ejemplo:

```
(define x 10)
(define y 20)
(let ((x 1)
      (y 2))
```

```
(z 3)
(+ x y z))
⇒ 6
```

## El ámbito de las variables definidas en el let es local

Las variables definidas en el `let` sólo tienen valores en el entorno creado por la forma especial.

```
(define x 10)
(define y 20)
(let ((x 1)
      (y 2)
      (z 3))
  (+ x y z))
⇒ 6
```

Cuando ha terminado la evaluación del `let` el ámbito local desaparece y quedan los valores definidos en el ámbito global.

```
x ⇒ 5
y ⇒ error, no definida
```

## Let permite usar variables definidas en un ámbito en el que se ejecuta el let

Al igual que en la invocación a funciones, desde el ámbito definido por el `let` se puede usar las variables del entorno en el que se está ejecutando el `let`. Por ejemplo, en el siguiente código se usa la variable `z` definida en el ámbito global.

```
(define z 8)
(let ((x 1)
      (y 2))
  (+ x y z))
⇒ 11
```

## Variables en las definiciones del let

Las expresiones que dan valor a las variables del `let` se evalúan todas en el entorno en el que se ejecuta el `let`, antes de crear las variables locales. No se realiza una asignación secuencial:

```
(define x 1)
(let ((w (+ x 3))
      (z (+ w 2))) ; Error: w no está definida
  (+ w z))
```

## Semántica del let

Para evaluar una expresión **let** debemos seguir las siguientes reglas:

1. Evaluar todas las expresiones de la derecha de las variables y guardar sus valores en variables auxiliares locales.
2. Definir un ámbito local en el que se ligan las variables del **let** con los valores de las variables auxiliares.
3. Evaluar el cuerpo del **let** en el ámbito local

## Let se define utilizando lambda

La semántica anterior queda clara cuando comprobamos que **let** se puede definir en función de **lambda**. En general, la expresión:

```
(let ((<var1> <exp1>) ... (<varn> <expn>)) <cuerpo>)
```

se puede implementar con la siguiente llamada a **lambda**:

```
((lambda (<var1> ... <varn>) <cuerpo>) <exp1> ... <expn>)
```

Para ejecutar un **let** con un **lambda** se debe crear un procedimiento en tiempo de ejecución con tantas variables como las variables del **let**, con el cuerpo del **let**, y se debe invocar a dicho procedimiento con las expresiones de las variables del **let**. Esas expresiones se evalúan antes de invocar al procedimiento y la invocación se realiza con los resultados. La invocación crea un entorno local con los parámetros del procedimiento (las variables del **let**) asociados a los valores, y en este ámbito local se ejecuta el cuerpo del procedimiento (el cuerpo del **let**).

Por ejemplo:

```
(let ((x (+ 2 3))
      (y (+ x 3)))
  (+ x y))
```

Equivale a:

```
((lambda (x y) (+ x y)) (+ 2 3) (+ x 3))
```

## Let dentro de funciones

Podemos usar **let** en el cuerpo de funciones para crear nuevas variables locales, además de los parámetros de la función

```
(define (suma-cuadrados x y)
  (let ((cuadrado-x (cuadrado x))
        (cuadrado-y (cuadrado y)))
    (+ cuadrado-x cuadrado-y)))
(suma-cuadrados 4 10)
⇒ 116
```

Cuando se invoca `(suma-cuadrados 4 10)` se crea un entorno local en el que las variables `x` e `y` toman el valor `4` y `10` y en el que se ejecuta la forma especial `let`. Esta forma especial crea a su vez un entorno local en el que se definen las variables `cuadrado-x` y `cuadrado-y` que toman los valores devueltos por las expresiones `(cuadrado x)` y `(cuadrado y)`: `16` y `100`. En este entorno local se evalúa la expresión `(+ cuadrado-x cuadrado-y)`.

El uso de `let` permite aumentar la legibilidad de los programas, dando nombre a expresiones:

Por ejemplo:

```
(define (distancia x1 y1 x2 y2)
  (let ((distancia-x (- x2 x1))
        (distancia-y (- y2 y1)))
    (sqrt (+ (cuadrado distancia-x)
              (cuadrado distancia-y)))))
```

Otro ejemplo:

```
(define (intersecta-intervalo a1 a2 b1 b2)
  (let ((dentro-b1 (and (≥ b1 a1)
                        (≤ b1 a2)))
        (dentro-b2 (and (≥ b2 a1)
                        (≤ b2 a2))))
    (or dentro-b1 dentro-b2)))
```

### 6.3. Clausuras

Vamos a terminar explicando el concepto de *clausura*. Hemos visto que las funciones son objetos de primera clase de Scheme y que es posible crear funciones en tiempo de ejecución con la forma especial `Lambda`.

Una clausura es una función devuelta por otra función. La clausura **captura las variables locales** de la función principal y puede usarlas en su propio código cuando este se invoque posteriormente.

Veamos un ejemplo. Supongamos que definimos la siguiente función `(make-sumador k)` que devuelve otra función.

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

```
(define f (make-sumador 10))
(f 2)
⇒ 12
```

En la función (`make-sumador k`) se llama a la forma especial lambda para crear un procedimiento. El procedimiento devuelto es lo que se denomina **clausura**. En este caso la clausura captura la variable local `k` (el parámetro de `make-sumador`) y usa su valor cuando posteriormente se invoca. Cuando se invoca a (`f 2`) se ejecuta la clausura y se crea un nuevo entorno local en el que `x` (el parámetro de la clausura) vale `2` y en el que se usa la variable `k` capturada.

En la invocación anterior (`f 2`), cuando se ejecuta la expresión (`+ x k`) las variables tienen los siguientes valores:

```
x: 2 (variable local de la clausura)
k: 10 (valor capturado del entorno local en el que se creó la clausura)
```

### Las variables locales creadas en un `let` también se capturan en las clausuras

Veamos el siguiente ejemplo, en el que creamos una función en un entorno local creado por un `let`:

```
(define x 10)
(define y 12)
(define (prueba x)
  (let ((y 3))
    (lambda (z)
      (+ x y z)))
(define h (prueba 5))
(h 2)
⇒ 8
```

Sucede lo siguiente:

1. Se invoca la expresión (`prueba 5`). Esto crea un entorno local en el que se le da a la variable `x` (el parámetro de `prueba`) el valor `5`. En este contexto se ejecuta el `let`, que crea otro entorno local en el que `y` vale `3`. En el contexto del `let` se crea una clausura con la invocación de la expresión lambda (`lambda (z) (+ x y z)`).
2. La clausura captura las variables locales `x` e `y` con sus valores `5` y `3` y la función `prueba` la devuelve como resultado de la invocación.
3. La clausura se guarda en la variable `h`.
4. Con la invocación (`h 2`) se invoca a la clausura, lo que crea un entorno local en el que se encuentran las siguientes variables:

```
z: 2 (parámetro de la clausura)
x: 5 (variable local de la función prueba capturada en el momento de
creación de la clausura))
y: 3 (variable local del let capturada en el momento de creación de la
clausura)
```

5. En este contexto se ejecuta la expresión `(+ x y z)`, que devuelve 10.

---

Lenguajes y Paradigmas de Programación, curso 2015-16

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante  
Domingo Gallardo, Cristina Pomares

## Tema 5: Programación imperativa

---

### Contenidos

- 1. Historia y características de la programación imperativa
  - 1.1. Historia de la programación imperativa
  - 1.2. Características principales de la programación imperativa
- 2. Programación imperativa en Scheme
  - 2.1. Pasos de ejecución
  - 2.2. Mutación con formas especiales `set!`
  - 2.3. Igualdad de referencia y de valor
- 3. Estructuras de datos mutables
  - 3.1. Mutación de elementos
  - 3.2. Funciones mutadoras: `make-ciclo!`, `append!` y `intercambia!`
  - 3.3. Lista ordenada mutable
  - 3.4. Diccionario mutable
  - 3.5. Ejemplo de mutación con listas de asociación
- 4. Ámbitos de variables y clausuras
  - 4.1. Variables de ámbito global
  - 4.2. Variables de ámbito local
  - 4.3. Clausuras y estado local
- 5. Clausuras con mutación = estado local mutable
  - 5.1. Estado local mutable
  - 5.2. Paso de mensajes

### Bibliografía

En este tema explicamos conceptos de los siguientes capítulos del libro *Structure and Interpretation of Computer Programs*:

- 3.3.1 Mutable List Structure
- 3.3.2 Representing Queues
- 3.3.3 Representing tables

- [3.1.1 Local State Variables](#)
- [3.1.3 The Cost of Introducing Assignment](#)

## 1. Historia y características de la programación imperativa

### 1.1. Historia de la programación imperativa

#### 1.1.1. Orígenes de la programación imperativa

- La programación imperativa es la forma natural de programar un computador, es el estilo de programación que se utiliza en el ensamblador, el estilo más cercano a la arquitectura del computador
- Características de la arquitectura [arquitectura clásica de Von Newmann](#):
  - memoria donde se almacenan los datos (referenciables por su dirección de memoria) y el programa
  - unidad de control que ejecuta las instrucciones del programa (contador del programa)
  - Los primeros lenguajes de programación (como el Fortran) son abstracciones del ensamblador y de esta arquitectura. Lenguajes más modernos como el BASIC o el C han continuado con esta idea.

#### 1.1.2. Programación procedural

- Uso de procedimientos y subrutinas
- Los cambios de estado se localizan en estos procedimientos
- Los procedimientos especifican parámetros y valores devueltos (un primer paso hacia la abstracción y los modelos funcionales y declarativos)
- Primer lenguaje con estas ideas: ALGOL

#### 1.1.3. Programación estructurada

- Artículo a finales de los 60 de Edsger W. Dijkstra: [GOTO statement considered harmful](#) en el que se arremete contra la sentencia GOTO de muchos lenguajes de programación de la época
- La programación estructurada mantiene la programación imperativa, pero haciendo énfasis en la necesidad de que los programas sean correctos (debe ser posible de comprobar formalmente los programas), modulares y mantenibles.
- Lenguajes: Pascal, ALGOL 68, Ada

#### 1.1.4. Programación Orientada a Objetos

- La POO también utiliza la programación imperativa, aunque extiende los conceptos de modularidad, mantenibilidad y estado local
- Se populariza a finales de los 70 y principios de los 80

### 1.2. Características principales de la programación imperativa

- Idea principal de la programación imperativa: la computación se realiza cambiando el estado del programa por medio de sentencias que definen pasos de ejecución del computador
- Estado del programa modificable
- Sentencias de control que definen pasos de ejecución

Vamos a ver unos ejemplos de estas características usando el lenguaje de programación Java.

### 1.2.1. Modificación de datos

- Uno de los elementos de la arquitectura de Von Newmann es la existencia de celdas de memoria referenciables y modificables

```
int x = 0;
x = x + 1;
```

- Otro ejemplo típico de este concepto en los lenguajes de programación es el array: una estructura de datos que se almacena directamente en memoria y que puede ser accedido y modificado.
- En Java los arrays son tipeados, mutables y de tamaño fijo

```
String[] unoDosTres = {"uno", "dos", "tres"};
unoDosTres[0] = unoDosTres[2];
```

- Un ejemplo de un método que recibe un parámetro de tipo array en Java. El parámetro se pasa por referencia:

```
public void llenaCadenas(String[] cadenas, String cadena) {
    for (int i = 0; i < cadenas.length; i++) {
        cadenas[i] = cadena;
    }
}
```

- Todos los ejemplos anteriores en un programa `main` Java:

```
public class Main {

    public static void main(String[] args) {
        int x = 0;
        x = x + 1;
        System.out.println("x: " + x);
        String[] unoDosTres = {"uno", "dos", "tres"};
        unoDosTres[0] = unoDosTres[2];
        for (String valor : unoDosTres) {
            System.out.println(valor);
        }
        llenaCadenas(unoDosTres, "uno");
        for (String valor : unoDosTres) {
            System.out.println(valor);
        }
    }
}
```

```

public static void llenaCadenas(String[] cadenas, String cadena) {
    for (int i = 0; i < cadenas.length; i++) {
        cadenas[i] = cadena;
    }
}

```

### 1.2.2. Almacenamiento de datos en variables

- Todos los lenguajes de programación definen variables que contienen datos
- Las variables pueden mantener valores (tipos de valor o value types) o referencias (tipos de referencia o reference types)
- En C, C++ o Java, los datos primitivos como int o char son de tipo valor y los objetos y datos compuestos son de tipo referencia
- La asignación de un valor a una variable tiene implicaciones distintas si el tipo es de valor (se copia el valor) o de referencia (se copia la referencia)

Copia de valor (datos primitivos en Java):

```

int x = 10;
int y = x;
x = 20;
System.out.println(y); // Sigue siendo 10

```

Copia de referencia (objetos en Java):

```

// import java.awt.geom.Point2D;
Point2D p1 = new Point2D.Double(2.0, 3.0);
Point2D p2 = p1;
p1.setLocation(12.0, 13.0);
System.out.println("p2.x = " + p2.getX()); // 12.0
System.out.println("p2.y = " + p2.getY()); // 13.0

```

- El uso de las referencias para los objetos de clases y para los tipos compuestos está generalizado en la mayoría de lenguajes de programación
- Tiene efectos laterales pero permite obtener estructuras de datos eficientes

### 1.2.3. Igualdad de valor y de referencia

- Todos los lenguajes de programación imperativos que permite la distinción entre valores y referencias implementan dos tipos de igualdad entre variables
- Igualdad de valor (el contenido de los datos de las variables es el mismo)
- Igualdad de referencia (las variables tienen la misma referencia)
- Igualdad de referencia => Igualdad de valor (pero al revés no)

- En Java la igualdad de referencia se define con `==` y la de valor con el método `equals`:

```
Point2D p1 = new Point2D.Double(2.0, 3.0);
Point2D p2 = p1;
Point2D p3 = new Point2D.Double(2.0, 3.0);
System.out.println(p1==p2);           // true
System.out.println(p1==p3);           // false
System.out.println(p1.equals(p3));    // true
```

#### 1.2.4. Sentencias de control

- También tiene su origen en la arquitectura de Von Newmann
- Sentencia que modifica el contador de programa y determina cuál será la siguiente instrucción a ejecutar
- Tipos de sentencias de control en programación estructurada:
  - Las sentencias de secuencia definen instrucciones que son ejecutados una detrás de otra de forma síncrona. Una instrucción no comienza hasta que la anterior ha terminado.
  - Las sentencias de selección definen una o más condiciones que determinan las instrucciones que se deberán ejecutar.
  - Las sentencias de iteración definen instrucciones que se ejecutan de forma repetitiva hasta que se cumple una determinada condición.

#### Bucles y variables

Ejemplo imperativo que imprime en Java una tabla con los productos de los números del 1 al 9:

```
for (int i = 1; i <= 9 ; i++) {
    System.out.println("Tabla del " + i);
    System.out.println("-----");
    for (int j = 1; j <= 9; j++) {
        System.out.println(i + " * " + j + " = " + i * j);
    }
    System.out.println();
}
```

## 2. Programación imperativa en Scheme

- Al igual que LISP, Scheme tiene características imperativas
- Vamos a ver algunas de ellas
  - Pasos de ejecución
  - Asignación con la forma especial `set!`
  - Datos mutables con las formas especiales `set-car!` y `set-cdr!`
- Una nota importante: todos los ejemplos que hay a continuación necesitan importar la librería `mutable-pairs`:

```
#lang r6rs
(import (rnrs)
        (rnrs mutable-pairs))
```

## 2.1. Pasos de ejecución

- Es posible definir pasos de ejecución con la forma especial `begin`
- Todas las sentencias de la forma especial se ejecutan de forma secuencial, una tras otra
- Tanto en la definición de funciones como en `lambda` es posible definir cuerpos de función con múltiples sentencias que se ejecutan también de forma secuencial. Hasta ahora no hemos usado esta característica porque hemos utilizado Scheme de forma funcional.

Ejemplo `begin`:

```
(begin
  (display "Escribe un número: ")
  (define x (read))
  (display "Escribe otro: ")
  (define y (read))
  (define maximo (max x y))
  (display (string-append "El máximo de "
                        (number->string x)
                        " y "
                        (number->string y)
                        " es "
                        (number->string maximo))))
```

Ejemplo de pasos de ejecución en la definición de una función:

```
(define (display-tres-valores a b c)
  (display a)
  (newline)
  (display b)
  (newline)
  (display c)
  (newline))
```

## 2.2. Mutación con formas especiales `set!`

### 2.2.1. Forma especial `set!`

- La forma especial `set!` permite asignar un nuevo valor a una variable
- La variable debe haber sido previamente creada con `define`
- La forma especial no devuelve ningún valor, modifica el valor de la variable usada

Sintaxis:

```
(set! <variable> <nuevo-valor>)
```

Por ejemplo, la típica asignación de los lenguajes imperativos se puede realizar de esta forma en Scheme:

```
(define a 10)
(set! a (+ a 1))
a ; => 11
```

La forma especial `set!` funciona con cualquier tipo de datos. Por ejemplo, utilizando la característica de que Scheme es débilmente tipeado, puede incluso asignar un nuevo tipo de valor a una variable:

```
(define a "Hola")
a ; => "Hola"
(set! a (cons 1 2))
a ; => {1 . 2}
```

## 2.2.2. Datos mutables

- En Scheme se definen las formas especiales `set-car!` y `set-cdr!` que permite modificar (mutar) la parte izquierda o derecha de una pareja una vez creada
- Al igual que `set!`, no devuelven ningún valor

Sintaxis:

```
(set-car! <pareja> <nuevo-valor>)
(set-cdr! <pareja> <nuevo-valor>)
```

Ejemplo

```
(define p (cons 1 2))
(set-car! p 10)
(set-cdr! p 20)
p ; => (10 . 20)
```

## 2.2.3. Efectos laterales

- La introducción de la asignación y los datos mutables hace posible que Scheme se comporte como un lenguaje imperativo en el que más de una variable apunta a un mismo valor y se producen efectos laterales
- Un efecto lateral se produce cuando el valor de una variable cambia debido a una sentencia en la que no aparece la variable

- Podemos comprobar ahora que las parejas son datos que se copian por referencia

Ejemplo:

```
(define p1 (cons 1 2))
(define p2 p1)
p1 ; => (1 . 2)
p2 ; => (1 . 2)
(set-car! p1 20)
p1 ; => (20 . 2)
p2 ; => (20 . 2)
```

## 2.3. Igualdad de referencia y de valor

- La utilización de referencias, la mutación y los efectos laterales hace también necesario definir dos tipos de igualdades: igualdad de referencia e igualdad de valor.
- Igualdad de referencia: dos variables son iguales cuando apuntan al mismo valor
- Igualdad de valor: dos variables son iguales cuando contienen el mismo valor
- En Scheme la función `eq?` comprueba la igualdad de referencia y `equal?` la igualdad de valor
- Igualdad de referencia implica igualdad de valor, pero no al revés

Ejemplo:

```
(define p1 (cons 10 20))
(define p2 p1)
(define p3 (cons 10 20))
(equal? p3 p1) ; => #t
(eq? p3 p1) ; => #f
(eq? p2 p1) ; => #t
(set! p3 p1) ;; La asignación copia referencias
(eq? p3 p1) ; => #t
```

## 3. Estructuras de datos mutables

- La utilización de las formas especiales `set-car!` y `set-cdr!` permite un estilo nuevo de manejo de las estructuras de datos ya vistas (listas o árboles)
- Es posible implementar funciones más eficientes que actualizan la estructura modificando directamente las referencias de unas celdas a otras
- Las operaciones no construyen estructuras nuevas, sino que modifican la ya existente

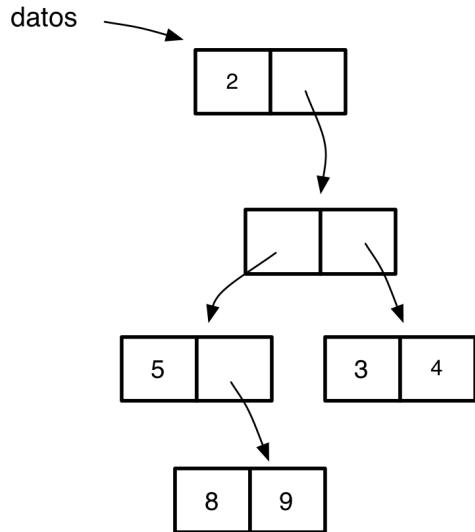
### 3.1. Mutación de elementos

#### Ejemplo 1

Vamos a empezar con un ejemplo sencillo en el que vamos a mutar un elemento de una estructura de datos formada por parejas. Supongamos la siguiente estructura:

```
(define datos (cons 2
    (cons (cons 5
        (cons 8 9))
    (cons 3 4))))
```

El diagrama *box-and-pointer* es el siguiente. Fíjate que hay elementos de las parejas que son datos atómicos y otros que son referencias a otras parejas.



Vamos ahora a *mutar* la estructura utilizando las sentencias `set-car!` y `set-cdr!`. Para mutar una pareja acceder a la pareja y modificar su parte derecha o su parte izquierda con las sentencias anteriores.

Por ejemplo, ¿cómo cambiaríamos el 8 por un 18? Deberíamos obtener la pareja `(8 . 9)` que está al final de la estructura y modificar su parte izquierda:

```
(set-car! (cdr (car (cdr datos))) 18)
```

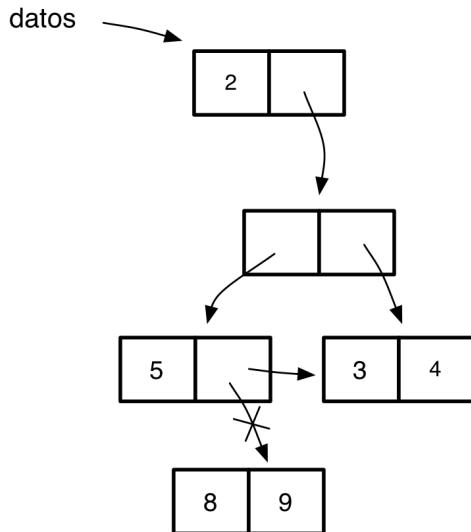
La expresión `(cdr (car (cdr datos)))` devuelve la pareja que queremos modificar y la sentencia `set-car!` modifica su parte izquierda.

Si ahora vemos qué hay en `datos` veremos que se ha modificado la estructura:

```
(print-pareja datos)
; => (2 . ((5 . (18 . 9)) . (3 . 4)))
```

Recuerda que `print-pareja` es una función que vimos anteriormente.

Hemos mutado un dato por otro. También podemos mutar las referencias a las parejas. Por ejemplo, podríamos modificar la parte derecha de la pareja que contiene el 5 para que apunte a la pareja `(3 . 4)`:



Para ello habría que obtener la pareja que contiene el 5 con la expresión `(car (cdr datos))` y mutar su parte derecha `(set-cdr!)` con la referencia a la pareja `(3 . 4)` que se obtiene con la expresión `(cdr (cdr datos))`:

```
(set-cdr! (car (cdr datos)) (cdr (cdr datos)))
```

## Ejemplo 2

Veamos un segundo ejemplo.

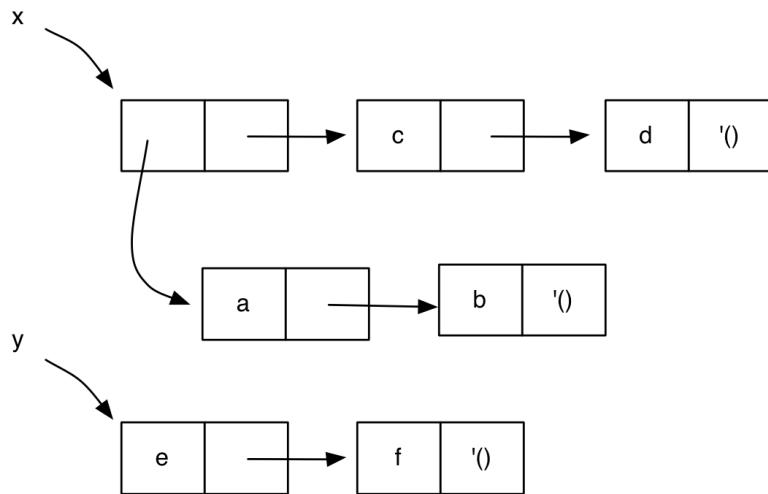
Supongamos las siguientes sentencias. ¿Cuál sería el *box-and-pointer* resultante?

```
(define x '((a b) c d))  
(define y '(e f))  
(set-car! x y)  
(define z (cons y (cdr x)))  
(set-cdr! x y)  
(set-car! z (caar x))
```

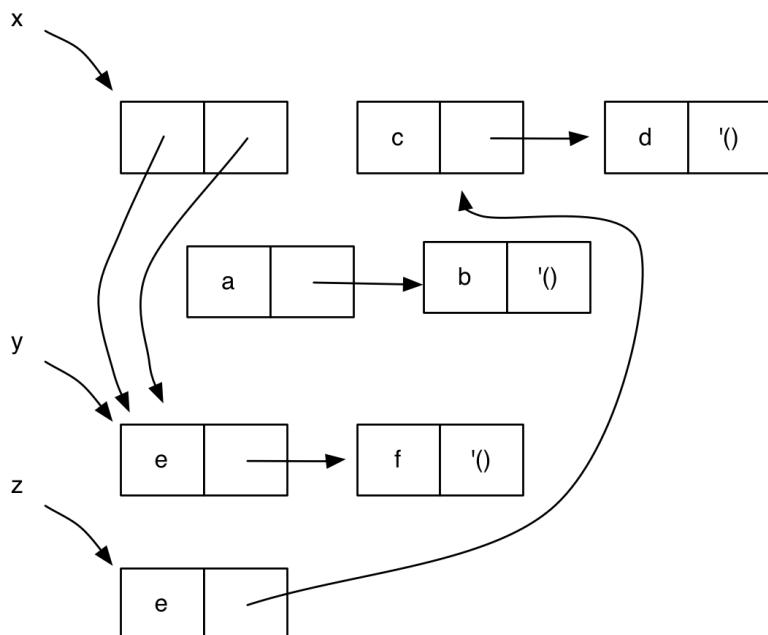
Debemos ir paso a paso ejecutando cada instrucción y dibujando cómo cambia el diagrama. El diagrama representa el **estado del programa**, y es un conjunto de parejas que contienen valores y referencias entre ellas.

Es muy importante considerar cuándo la sentencia copia un valor y cuando copia una referencia. Las parejas siempre tienen semántica de referencia.

El diagrama resultante después de las dos primeras sentencias es:



Y el diagrama después de todas las sentencias es:



Los valores de las variables después de todas las sentencias son los siguientes:

```
x ; => {{e f} e f}
y ; => {e f}
z ; => {e c d}
```

### Ejemplo 3

En este ejemplo vamos a usar una nueva forma especial de Scheme que permite crear variables locales y ejecutar expresiones en las que se utilicen esas variables.

Se trata de la forma especial `let`. Veremos algo más de esta forma especial más adelante, cuando hablemos de ámbitos de variables. Por ahora es suficiente saber que permite crear variables locales y ejecutar un cuerpo (que puede tener más de una sentencia) con esas variables.

```
(let ((x (+ 2 3))
      (y (* 2 10))
      (+ x y))
; => 25
```

En este caso `let` permite crear dos variables locales `x` e `y` y asignarles los valores `5` y `20` respectivamente. Las sentencias del cuerpo del `let` (en este caso la sentencia `(+ x y)`) se evalúan con estos valores recién creados.

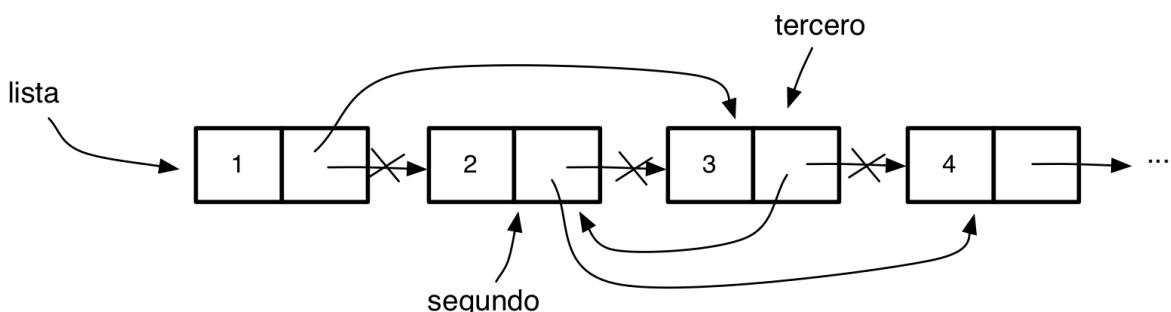
Vamos a usar esta forma especial en ocasiones para guardar valores o referencias que después necesitamos asignar.

Por ejemplo, ¿cómo podríamos definir una función que mute **las referencias** de una lista para intercambiar sus posiciones segunda y tercera (sin crear nuevas parejas)?

Si llamamos a la función `intercambia!` debería hacer lo siguiente:

```
(define lista '(1 2 3 4 5 6))
(intercambia! lista)
lista ; => {1 3 2 4 5 6}
```

Dibujando el diagrama *box-and-pointer* podemos ver las mutaciones que tenemos que hacer:



La función es la siguiente:

```
(define (intercambia! lista)
  (let ((segundo (cdr lista))
        (tercero (cddr lista)))
    (set-cdr! lista tercero)
    (set-cdr! segundo (cdr tercero))
    (set-cdr! tercero segundo)))
```

### 3.2. Funciones mutadoras: `make-ciclo!`, `append!` e `intercambia-lista!`

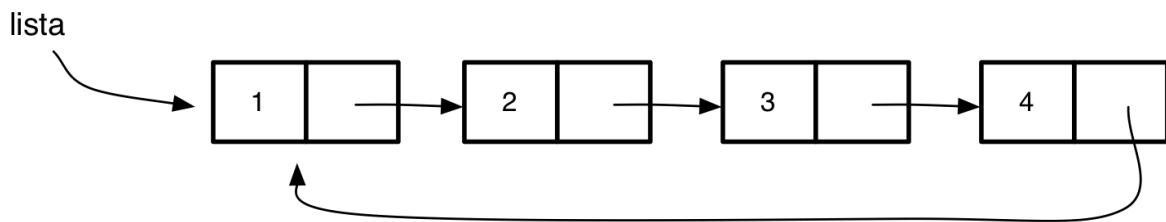
Veamos ahora varios ejemplos de funciones recursivas que recorren listas y realizan mutaciones en ellas.

Normalmente las funciones mutadoras no devuelven una estructura, sino que modifican la que se pasa como parámetro.

Por convenio, indicaremos que una función es mutadora terminando su nombre con un signo de admiración.

### **make-ciclo!**

Empecemos con una función muta una lista, haciendo que la parte derecha de la última pareja apunte a la pareja inicial de la misma.



Para conseguirlo, necesitamos guardar la referencia a la pareja inicial de la lista en el primer parámetro de la función. Y el segundo parámetro irá avanzando hasta encontrar el final de la lista:

```
(define (make-ciclo! lista ref)
  (if (null? (cdr ref))
      (set-cdr! ref lista)
      (make-ciclo! lista (cdr ref))))
```

Si probamos la función con una lista, podemos comprobar que DrRacket detecta el ciclo que se produce al final de la misma:

```
(define lista '(1 2 3 4 5 6))
(make-ciclo! lista lista)
lista ; ⇒ #0={1 2 3 4 5 6 . #0#}
```

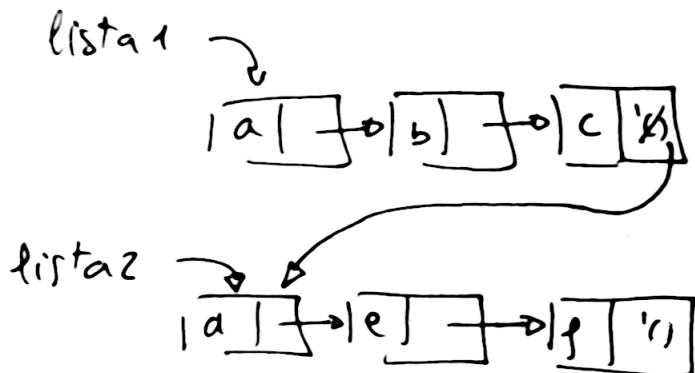
Una forma alternativa de implementar esta función es mediante una función que devuelva la referencia de la última pareja de la lista:

```
(define (ultima-pareja lista)
  (if (null? (cdr lista))
      lista
      (ultima-pareja (cdr lista)))))

(define (make-ciclo2! lista)
  (let ((ultima-pareja (ultima-pareja lista)))
    (set-cdr! ultima-pareja lista)))
```

### **append!**

Y vemos a continuación una versión mutadora de `append` que llamamos `append!` y que mejora la eficiencia de la original, copiando al final de la primera lista una referencia a la segunda:



La implementación es la siguiente:

```
(define (append! l1 l2)
  (if (null? (cdr l1))
      (set-cdr! l1 l2)
      (append! (cdr l1) l2)))
```

Ejemplo:

```
(define a '(1 2 3 4))
(define b '(5 6 7))
(append! a b)
a ; => {1 2 3 4 5 6 7}
```

Algunas puntuaciones:

- Al igual que `set!`, `set-car!` o `set-cdr!`, la función `append!` no devuelve ningún valor, sino que modifica directamente la lista que se pasa como primer parámetro.
- Al modificarse la lista, todas las referencias que apuntan a ellas quedan también modificadas.
- La función daría un error en el caso en que la llamáramos con una lista vacía como primer argumento.

### intercambia-lista!

Un último ejemplo de función recursiva que muta una lista. Queremos definir una función que intercambie todas las parejas excepto la primera de una lista con un número de datos impar, pero **sin usar set-car!**.

Llamamos a la función (`intercambia-lista! lista`) y debería hacer lo siguiente:

```
(define lista '(1 2 3 4 5 6 7))
(intercambia-lista! lista)
```

```
lista ; => {1 3 2 5 4 7 6}
```

La forma de implementar esta función va a ser llamando a la función `intercambia!` (la que hemos definido anteriormente que intercambia el segundo y el tercer elemento) hasta que lleguemos al final de la lista. Lo hacemos con una recursión:

```
(define (intercambia-lista! lista)
  (if (not (null? (cdr lista)))
      (begin
        (intercambia! lista)
        (intercambia-lista! (cddr lista))))
```

La función termina cuando llegamos a la pareja final de la lista, la que tiene la lista vacía en su parte derecha.

Si no estamos en el final de la lista llamamos a la función `intercambia!` para intercambiar la siguiente pareja con su siguiente y llamamos a la recursión con la variable `lista` apuntando a la tercera pareja de la lista.

### 3.3. Lista ordenada mutable

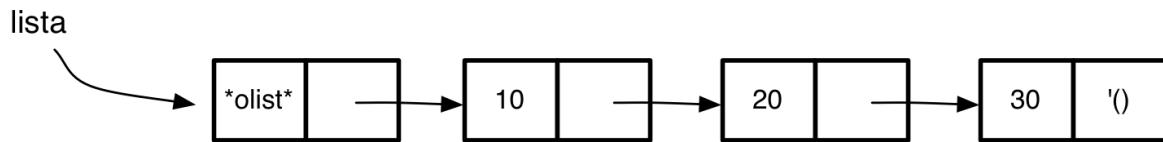
Veamos ahora una lista ordenada mutable de números, en la que insertaremos elementos de forma ordenada.

Definiremos las siguientes funciones:

- `(make-olist)`: construye una lista ordenada vacía
- `(borra-primer-o-olist! olist)`: función mutadora que elimina (con mutación) el primer elemento de la lista
- `(inserta-olist! olist n)`: función mutadora que inserta (con mutación) de forma ordenada un número en la lista

Implementamos el tipo de datos con una lista normal con una pareja adicional en cabeza. Esta pareja adicional funcionará de **cabecera de la lista** y será la referencia inmutable a la que apuntará cualquier variable que apunte a la lista. De esta forma podremos insertar elementos en primera posición de la lista. En la cabecera usaremos por convenio el símbolo `'*olist*`.

```
(define lista '(*olist* 10 20 30))
```



#### Constructor

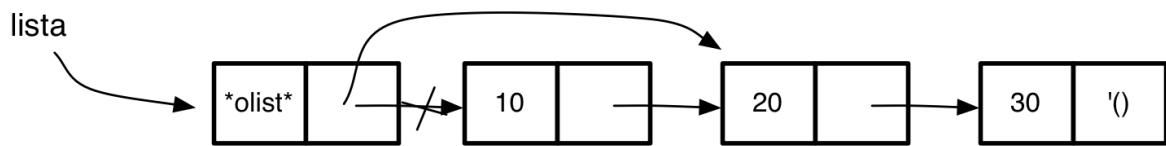
El constructor `(make-olist)` devuelve una lista sólo con la cabecera.

```
(define (make-olist)
  (list '*olist*))
```

## Mutadores

Las funciones mutadoras modifican la estructura de datos.

Comenzamos con la función (**borra-primero-olist!** olist) que elimina con mutación el primer elemento de la lista ordenada:



```
(define (borra-primero-olist! olist)
  (set-cdr! olist (cddr olist)))
```

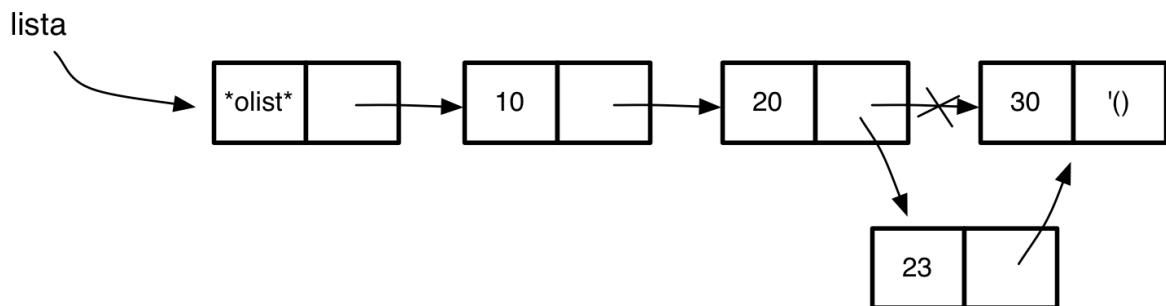
Definimos la función mutadora **inserta-olist!** que modifica la lista ordenada añadiendo una nueva pareja, insertándola en la posición correcta modificando las referencias.

La función (**add-item!** item ref) es la función clave que crea una nueva pareja con el **item** y la añade en el **cdr** de la pareja a la que apunta **ref**.

```
(define (add-item! ref item)
  (set-cdr! ref (cons item (cdr ref))))
```

```
(define (inserta-olist! olist n)
  (cond
    ((null? (cdr olist)) (add-item! olist n))
    ((< n (cadr olist)) (add-item! olist n))
    ((= n (cadr olist)) #f) ; el valor devuelto no importa
    (else (inserta-olist! (cdr olist) n))))
```



Ejemplo de uso:

```
(define c (make-olist))
(inserta-olist! c 5)
(inserta-olist! c 10)
(inserta-olist! c -10)
c ; => {*olist* -10 5 10}
```

### 3.4. Diccionario mutable

- Veamos ahora un ejemplo más: un *diccionario* mutable definido mediante una lista de asociación formada por parejas de clave y valor

```
(define l-assoc (list (cons 'a 1) (cons 'b 2) (cons 'c 3)))
```

- La función de Scheme `assq` recorre la lista de asociación y devuelve la tupla que contiene el dato que se pasa como parámetro como clave

```
(assq 'a l-assoc) ; => (a.1)
(assq 'b l-assoc) ; => (b.2)
(assq 'c l-assoc) ; => (c.3)
(assq 'd l-assoc) ; => #f
(cdr (assq 'c l-assoc)) ; => 3
```

- La función `assq` busca en la lista de asociación usando la igualdad de referencia `eq?`

```
(define p (cons 1 2))
(define l-assoc (list (cons 'a 1) (cons p 2) (cons 'c 3)))
(assq (cons 1 2) l-assoc) ; => #f
(assq p l-assoc) ; => {{1 . 2} . 2}
```

Las funciones que vamos a implementar del diccionario son las siguientes:

- `(make-dic)`: construye un diccionario vacío
- `(put-dic! dic clave valor)`: inserta en el diccionario un nuevo valor asociado a una clave
- `(get-dic dic clave)`: devuelve el valor asociado a una clave en un diccionario

Al igual que la lista ordenada mutable, el diccionario necesita una cabecera:

```
(define (make-dic)
  (list '*dic*))
```

La función `(get-dic dic clave)` usa la función `assq` para buscar la pareja que contiene una clave determinada en la lista de asociación (el `cdr` de `dic`). Y después devuelve el `cdr` de esa pareja (el valor

guardado junto con la clave:

```
(define (get-dic dic clave)
  (let (pareja (assq clave (cdr dic)))
    (if (not pareja)
        #f
        (cdr pareja)))
```

La función `(put-dic! dic clave valor)` usa igual que antes la función `assq` para comprobar si la clave ya está en la lista de asociación. Si existe, sustituye el valor por el nuevo. Si no existe, crea una pareja con la clave y el valor y la inserta al comienzo del diccionario (después de la cabecera):

```
(define (put-dic! dic clave valor)
  (let (pareja (assq clave (cdr dic)))
    (if (not pareja)
        (set-cdr! dic
                   (cons (cons clave valor)
                         (cdr dic)))
        (set-cdr! pareja valor)))
  'ok)
```

En las funciones anteriores volvemos a usar la forma especial `let`.

Ejemplos de uso:

```
(define dic (make-dic))
(put-dic! dic 'a 10) ; => ok
(get-dic dic 'a) ; => 10
(put-dic! dic 'b '(a b c)) ; => ok
(get-dic dic 'b) ; => {a b c}
(put-dic! dic 'a 'ardilla) ; => ok
(get-dic dic 'a) ; => ardilla
```

### 3.5. Ejemplo de mutación con listas de asociación

Una vez introducidos distintas estructuras de datos mutables, incluyendo listas de asociación, vamos a terminar estos ejemplos con un ejemplo práctico en el que intervienen las listas y las listas de asociación. Se trata de escribir un procedimiento `regular->assoc!` que transforme una lista regular en una lista de asociación sin crear nuevas parejas. La lista regular `(k1 v1 k2 v2 k3 v3 ...)` deberá convertirse en la lista de asociación `((k1 . v1) (k2 . v2) (k3 . v3) ...)`.

Ejemplo:

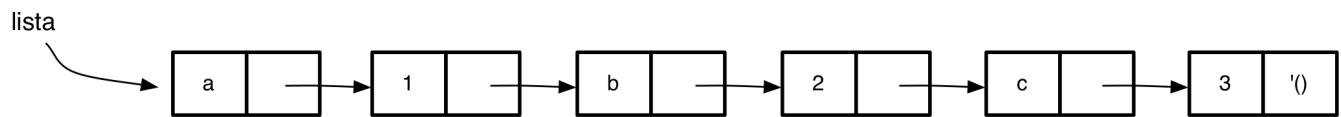
```
(define my-list (list 'a 1 'b 2 'c 3))
lista ; => (a 1 b 2 c 3)
```

```
(regular->assoc! my-list)
lista ; => ((a . 1) (b . 2) (c . 3))
```

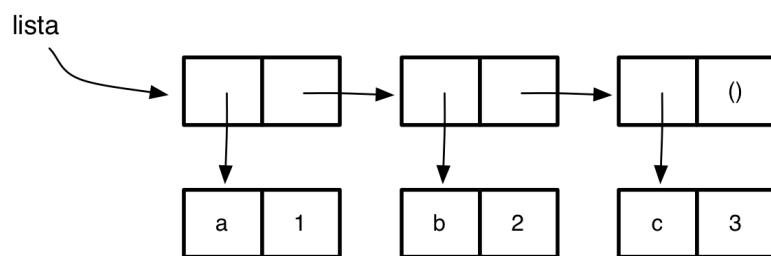
Una posible solución a este problema sería la siguiente (no es la única solución):

Cuando trabajamos con este tipo de problemas, es muy útil ayudarse con los diagramas caja y puntero. Vamos a crear los diagramas caja y puntero para el ejemplo anterior:

Antes de llamar a regular->assoc!:

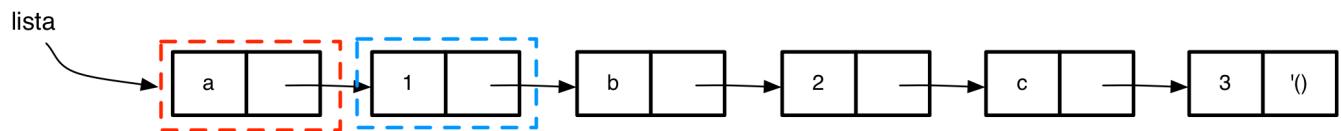


Después de llamar a regular->assoc!:

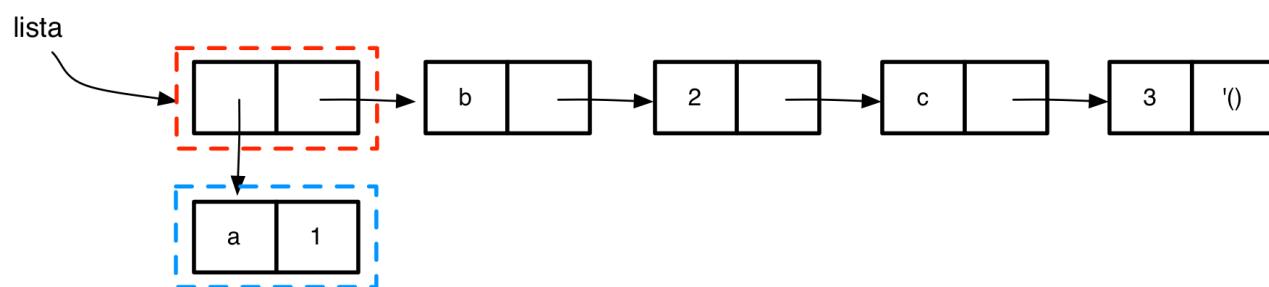


Queremos hacer la transformación sin crear nuevas parejas, por lo que cada pareja en el primer diagrama corresponde a una pareja particular en el segundo diagrama. No podemos modificar la primera pareja de la lista (porque perderíamos la ligadura de la variable **lista**), por lo que es primera pareja tiene que permanecer en el primer lugar. Por otra parte, **a** y **1** van a formar parte del mismo par en la lista de asociación, por lo que vamos a considerar el siguiente cambio de las dos primeras parejas.

Diagrama inicial:



Después del cambio de las dos primeras parejas:

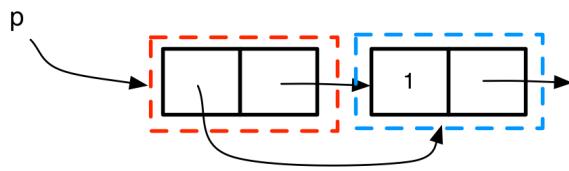


La siguiente función **manejador-dos-parejas!** es la encargada de hacer esta mutación:

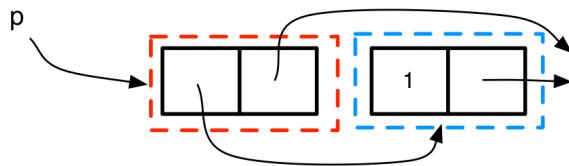
```
(define (manejar-dos-parejas! p)
  (let ((key (car p))) ; 1
    (set-car! p (cdr p)) ; 2
    (set-cdr! p (cdar p)) ; 3
    (set-cdr! (car p) (caar p)) ; 4
    (set-car! (car p) key))) ; 5
```

Se han numerado las líneas para una mejor explicación. En la línea 1, `(define key (car p))`, creamos una variable local `key` que guarda el valor actual del `(car p)`, ese valor será la clave de la primera pareja de la lista de asociación; necesitamos almacenarlo para no perderlo.

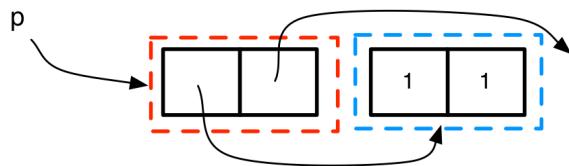
En la línea 2, `(set-car! p (cdr p))`, cambiamos el `car` de `p` para que apunte a la siguiente pareja (la azul):



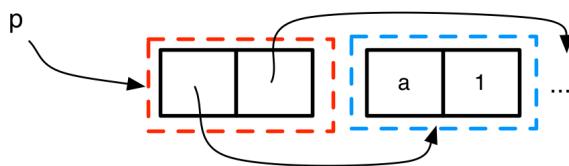
En la línea 3, `(set-cdr! p (cdar p))`, copiamos en el `cdr` de `p` el `cdr` de la pareja en azul para que ambos apunten a la siguiente pareja:



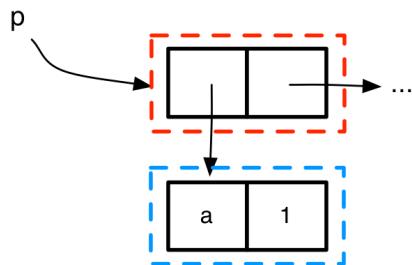
En la línea 4, `(set-cdr! (car p) (caar p))`, cambiamos el `cdr` de la pareja en azul al `car` de la misma pareja. Hacemos ésto porque en una lista de asociación, los valores se guardan en los `cdrs` y las claves en los `cars`:



Por último, en la línea 5 `(set-car! (car p) key)`, completamos el problema poniendo la clave que habíamos guardado, en el `car` de la pareja azul:



Reordenamos el diagrama para verlo más claro:



Hemos definido un procedimiento que maneja un subproblema (dos parejas) del problema. Ahora sólo nos queda definir la función que maneja toda la lista:

```
(define (regular->assoc! lista)
  (if (null? lista)
    'ok
    (begin (manejar-dos-parejas! lista)
      (regular->assoc! (cdr lista)))))
```

## 4. Ámbitos de variables y clausuras

El concepto del **ámbito** de vida de las variables es un concepto fundamental en los lenguajes de programación. Utilizaremos las palabras **ámbito** y *entorno* como sinónimos. En inglés se utiliza el término *scope*.

Cuando se define una variable, asociándole un valor, esta asociación tiene una extensión determinada, ya sea en términos de tiempo de compilación (**ámbito léxico**) como en términos de tiempo de ejecución (**ámbito dinámico**). El ámbito de una variable determina cuándo podemos referirnos a ella para recuperar el valor asociado.

Al conjunto de variables disponibles en una parte del programa o en una parte de su ejecución se denomina **contexto** o **entorno** (*context* o *environment*).

### 4.1. Variables de ámbito global

Una variable definida en el programa con la instrucción `define` tiene un ámbito global.

```
(define a "holo")
(define b (string-append "adios" a))
(define cuadrado (lambda (x) (* x x)))
```

Todas las variables definidas fuera de funciones forman parte del **entorno global** del programa.

### 4.2. Variables de ámbito local

En Scheme existen dos formas de definir variables de ámbito local: la forma especial `define` dentro de una función y la forma especial `let`.

#### 4.2.1 Variables locales con `define`

Como en la mayoría de lenguajes de programación, en Scheme se crea un **entorno o ámbito local** (memoria local de la invocación de la función) cada vez que se invoca a una función.

En este entorno local toman valor los parámetros y las variables locales de la función. Es posible definir variables locales en una función utilizando la forma especial **define** dentro de la propia función. Esto no lo hacíamos dentro del paradigma funcional, para evitar realizar pasos de ejecución. Pero ahora que estamos en el paradigma imperativo podemos utilizarlo.

```
(define (distancia x1 y1 x2 y2)
  (define distancia-x (- x2 x1))
  (define distancia-y (- y2 y1))
  (sqrt (+ (cuadrado distancia-x)
            (cuadrado distancia-y))))
```

Podemos usar en un entorno local una variable con el mismo nombre que en el entorno global. Cuando se ejecute el código de la función se utilizará el valor local.

Por ejemplo, supongamos las siguientes expresiones:

```
(define x 5)
(define (suma-10 y)
  (define x 10)
  (+ x y))

(suma-10 3) ; => 13
```

La primera sentencia de la función (**suma y**) define una variable local **x** a la que se asigna un valor inicial de 10.

Cuando se ejecuta la expresión (**+ x y**), en la invocación a (**suma 3**) el valor de **x** es entonces 10 (el valor local que hemos definido en la sentencia anterior), no es 5, devolviéndose 15.

Sucede igual si un parámetro tiene el mismo nombre que una variable global:

```
(define x 5)
(define (suma-3 x)
  (+ x 3))

(suma-3 12) ; => 15
```

Cuando se ejecuta la expresión (**+ x 3**) en la invocación a (**suma-3 12**) el valor de **x** es 12, no es 5, devolviéndose 15.

En el entorno local también se pueden utilizar variables definidas en el entorno global. Por ejemplo:

```
(define z 12)
(define (foo y)
  (define x 10)
  (+ x y z))

(foo 5) ; => 27
```

La expresión `(+ x y z)` se evalúa en el entorno local en el que `x` vale 10 e `y` vale 5. Al no estar definida la variable `z` en este entorno local, se usa su definición de ámbito global.

Una vez realizada la invocación, desparece el entorno local junto con las variables locales definidas en él, y se recupera el contexto global. Por ejemplo, en la siguiente expresión, una vez realizada la invocación a `(suma-10 2)` se devuelve el número 12 y se evalúa en el entorno global la expresión `(+ 12 x)`. En este contexto la variable `x` vale 5 por lo que la expresión devuelve 17.

```
(define x 5)
(define (suma-10 y)
  (define x 10)
  (+ x y))

(+ (suma-10 2) x) ; => 17
```

Otro ejemplo:

```
(define x 10)
(define (foo2 y)
  (define z 5)
  (+ x y z))

(foo2 2) ; => 17
x ; => 10
z ; => error, no definida
y ; => error, no definida
```

## Define en el cuerpo de una función

La forma especial `define` para crear variables locales sólo puede utilizarse al comienzo de la definición de una función. Si intentamos utilizarla a mitad del código de la función tendremos un error:

```
(define (suma-10-si-mayor-que-0 x)
  (if (> x 0)
    (begin
      ;; ERROR:
      ;; no es posible usar define a mitad del código
      ;; de una función
      (define y 10))
```

```
(+ x y))
x))
```

#### 4.2.1 Variables locales con let

La otra forma de definir variables locales en Scheme es con la forma especial `let`.

La forma especial `let` permite crear un ámbito local en el que se da valor a variables y se evalúan expresiones.

Sintaxis:

```
(let ((<var1> <exp-1>
      ...
      (<varn> <exp-n>))
       <cuerpo>)
```

Las variables `var1`, ... `varn` toman los valores devueltos por las expresiones `exp1`, ... `expn` y el cuerpo se evalúa con esos valores. Esas variables sólo tienen valor en el ámbito de la forma especial.

Por ejemplo:

```
(define x 10)
(let ((x (+ 1 2))
      (y (* 10 2))
      (+ x y)) ; => 23
x ; => 10
y ; => error, no definida
```

Cuando ha terminado la evaluación del `let` el ámbito local desaparece y quedan los valores definidos en el ámbito anterior.

Las expresiones que dan valor a las variables del `let` se evalúan antes de crear el ámbito local. Por ejemplo, en el siguiente código, las expresiones `(+ x 3)` y `(+ y 2)` devuelven 4 y 7 respectivamente:

```
(define x 1)
(define y 5)
(let ((w (+ x 3))
      (z (+ y 2)))
      (+ w z)) ; => 11
```

En el cuerpo del `let` puede haber más de una expresión:

```
(let ((x 10)
      (y 20))
      (display "x: "))
```

```
(display x)
(display "\ny: ")
(display y))
```

## Let en el cuerpo de una función

A diferencia de `define` sí que es posible usar `let` en cualquier expresión de Scheme, por ejemplo, en un `if`:

```
(define (suma-10-si-mayor-que-0 x)
  (if (> x 0)
      (let ((y 10))
        (+ x y))
      x))
```

## 4.3. Clausuras y estado local

Recordemos que la forma especial `lambda` permite crear funciones anónimas en tiempo de ejecución. Si ejecutamos una forma especial `lambda` como último paso de una función, se devolverá la función recién creada.

¿Qué sucede si utilizamos en el cuerpo de esta función anónima una variable local definida en la función principal?

Vamos a verlo:

```
(define (make-sumador)
  (define z 10)
  (lambda (x) (+ x z)))
```

La función `make-sumador` define una variable local `z` con el valor de 10. Y después construye una función con la expresión `lambda`. Al ser la última expresión de la función `make-sumador`, esta función construida en tiempo de ejecución es lo que se devuelve.

Si invocamos a `make-sumador` vemos que devuelve un procedimiento:

```
(make-sumador) ; ⇒ #<procedure>
```

Si guardamos el procedimiento en una variable y después lo invocamos, tendremos que pasarle un parámetro (correspondiente al argumento `x` de la expresión `lambda`). ¿Qué devolverá esa invocación?

```
(define f (make-sumador))
(f 5) ; ⇒ ???
```

Al invocar a `f` se ejecutará el cuerpo del procedimiento, esto es, la expresión `(+ x z)`. La variable `x` valdrá 5 (el valor del parámetro). Pero, ¿cuál es el valor de la variable `z`? Podemos comprobar que es 10, porque la invocación devuelve 15:

```
(f 5) ; => 15
```

¿Por qué? Lo que está pasando es que la función anónima que crea la expresión lambda **captura las variables locales** definidas en el ámbito en el que se crea. Por eso recibe el nombre de **clausura**, porque encierra los valores que tienen esas variables en el momento de su creación y esos valores son los que usa en las variables libres cuando posteriormente la invoquemos.

En el ejemplo anterior se está capturando la variable `z` con su valor 10. Cuando después invitamos a `(f 5)` se evalúa el cuerpo `(+ x z)` con `x` valiendo 5 y `z` valiendo el valor capturado (10).

También podemos utilizar como estado local el valor de un parámetro. Por ejemplo:

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

(define f (make-sumador 10))
(f 2)
; => 12
```

En la función `(make-sumador k)` se llama a la forma especial lambda para crear una clausura. La clausura captura la variable local `k` (el parámetro de `make-sumador`) y usará su valor cuando posteriormente se evalúe. En este caso, la clausura se ha creado con `k` valiendo 10, y captura este valor.

Cuando después se invoca a `(f 2)` se ejecuta la clausura en un nuevo ámbito local con las siguientes variables y valores:

```
x: 2 (variable local de la clausura)
k: 10 (valor capturado del entorno local en el que se creó la clausura)
```

En este ámbito se ejecuta la expresión `(+ x k)`, devolviéndose el valor 12.

Por último, también es posible crear la clausura dentro de un estado local definido con un `let` y usar en el cuerpo del `lambda` variables locales definidas en el `let`:

```
(define (make-sumador-cuadrado k)
  (if (> k 0)
    (let ((y (cuadrado k)))
      (lambda (x)
        (+ x y)))
    (lambda (x)
      x)))
```

La función anterior tiene un condicional en el que se comprueba si `k` es mayor que 0. Si lo es, se devuelve una clausura creada en el ámbito del `let` con la variable `y` capturada con el valor del cuadrado de `k`. Esta clausura suma a su entrada el valor de `y`.

Por ejemplo:

```
(define f (make-sumador-cuadrado 4))
(f 3) ; => 19
```

En el caso en que `k` menor o igual que 0, se devuelve una clausura que devuelve siempre el valor que recibe:

```
(define f (make-sumador-cuadrado -8))
(f 3) ; => 3
```

La definición de clausuras con un estado local inicializado a un valor creado en tiempo de ejecución es una característica muy potente. Permite no sólo crear funciones en tiempo de ejecución para utilizarlas posteriormente, sino configurarlas con el estado local que nos interese.

Veremos en el apartado siguiente que la combinación de clausuras y mutación permite crear funciones con estado local mutable, algo equivalente a objetos que encapsulan código y estado.

## 5. Clausuras con mutación = estado local mutable

### 5.1. Estado local mutable

Si combinamos una clausura con la posibilidad de mutar las variables capturadas por la clausura obtenemos un estado local mutable asociado a funciones creadas en tiempo de ejecución.

Veamos, por ejemplo, la siguiente función (`make-contador i`):

```
(define (make-contador i)
  (define x i)
  (lambda ()
    (set! x (+ x 1))
    x)))
```

La función `make-contador` define una clausura que captura la variable local `x` inicializada a `i`. En cada invocación a la clausura se ejecuta una sentencia `set!` que modifica el valor de la variable capturada. Y después se devuelve el nuevo valor de la variable. Estamos implementando un contador asociado a la clausura.

```
(define f (make-contador 10))
(f) => 11
```

```
(f) => 12
```

El estado (el valor del contador) se mantiene y se modifica entre distintas invocaciones a `f`. A diferencia del paradigma funcional podemos comprobar que distintas invocaciones a la misma función devuelven valores distintos (dependiente del estado de la clausura).

Podemos crear distintas clausuras, cada una con su variable capturada, en distintas invocaciones a `make-contador`:

```
(define h (make-contador 10))
(define g (make-contador 100))
(h) => 11
(h) => 12
(g) => 101
(g) => 102
```

Una forma alternativa de crear la clausura, sin usar la forma especial `lambda` es definiéndola con un `define` en el cuerpo de `make-contador`. Después de los dos `define` la última sentencia devuelve la función `incrementa` (la clausura).

```
(define (make-contador i)
  (define x i)
  (define (incrementa)
    (set! x (+ x 1))
    x)
  incrementa)
```

## 5.2. Paso de mensajes

En el ejemplo anterior creamos una clausura que siempre incrementa el valor del contador. ¿Cómo podríamos crear una clausura que permitiera hacer distintas cosas con el valor capturado?

Una forma de hacerlo es definir en la clausura un parámetro adicional (un símbolo que llamamos *mensaje*) y devolver distintas clausuras en función del valor de ese parámetro adicional:

```
(define (make-contador i)
  (define x i)
  (define (get)
    x)
  (define (inc-1)
    (set! x (+ x 1))
    x)
  (define (inc y)
    (set! x (+ x y))
    x))
```

```
(define (dispatcher mensaje)
  (cond
    ((equal? mensaje 'get) get)
    ((equal? mensaje 'inc-1) inc-1)
    ((equal? mensaje 'inc) inc)
    (else "Error: mensaje desconocido")))
  dispatcher)
```

La función `make-contador` devuelve la clausura `dispatcher` que se encarga de procesar el mensaje y devolver la clausura asociada a ese mensaje. Cuando invoquemos al `dispatcher` con un símbolo, devuelve otro procedimiento que hay que volver a invocar con los parámetros adecuados (sin parámetros en los dos primeros casos y con el valor a incrementar en el último caso):

- Si recibe el símbolo `'get` devuelve la clausura que devuelve el valor del contador
- Si recibe el símbolo `'inc-1` devuelve la clausura que incrementa el valor del contador en 1
- Si recibe el símbolo `'inc` devuelve la clausura que incrementa el valor del contador una cantidad determinada

Por ejemplo:

```
(define c (make-contador 100))
((c 'get)) => 100
((c 'inc-1)) => 101
((c 'inc) 10) => 111
```

---

Lenguajes y Paradigmas de Programación, curso 2016-17

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante  
Domingo Gallardo, Cristina Pomares

## Tema 5: Programación Funcional con Swift

---

### Introducción

Te recomendamos que leas el seminario de Swift en el que se introduce el lenguaje y se explica cómo ejecutar programas en este lenguaje:

- [Seminario de Swift](#)

### Conceptos fundamentales de Programación Funcional

Vamos a repasar en este tema cómo se implementan en Swift conceptos principalmente funcionales como:

- Valores inmutables
- Funciones como objetos de primera clase y cláusulas
- Funciones de orden superior

Repasamos rápidamente algunos conceptos básicos de programación funcional, vistos en los primeros temas de la asignatura.

### Programación Funcional:

La Programación Funcional es un paradigma de programación que trata la computación como la evaluación de funciones matemáticas y que evita cambios de estado y datos mutables.

### Funciones matemáticas o puras:

Las funciones matemáticas tienen la característica de que cuando las invocas con el mismo argumento siempre te devolverán el mismo resultado.

### Funciones como objetos de primera clase:

En programación funcional, las funciones son objetos de primera clase del lenguaje, similares a enteros o *strings*. Podemos pasar funciones como argumentos en las denominadas *funciones de orden superior* o devolver funciones creadas en tiempo de ejecución (clausuras).

## Características básicas de Swift

Swift es un lenguaje principalmente imperativo, pero en su diseño se han introducido conceptos modernos de programación funcional, extraídos de lenguajes como Rust o Haskell. Por ello se puede considerar un lenguaje **multi-paradigma**, en el que se puede definir código funcional que se puede ejecutar junto con código imperativo.

Como dice su creador [Chris Lattner](#):

El lenguaje Swift es el resultado de un esfuerzo incansable de un equipo de expertos en lenguajes, gurús de documentación, ninjas de optimización de compiladores [...]. Por supuesto, también se benefició enormemente de las experiencias ganadas en muchos otros lenguajes, tomando ideas de Objective-C, Rust, Haskell, Ruby, Python, C#, CLU, y demasiados otros para ser enumerados.

## Lenguaje fuertemente tipado

A diferencia de Scheme, Swift es un lenguaje **fuertemente tipado** en el que hay que definir los tipos de variables, parámetros y funciones.

Por ejemplo, en las siguientes declaraciones definimos variables de distintos tipos:

```
let n: Int = 10
let str: String = "Hola"
let array: [Int] = [1,2,3,4,5]
```

El compilador de Swift permite identificar los tipos de las variables cuando se realiza una asignación. La técnica se denomina **inferencia de tipos** y permite declarar variables sin escribir su tipo. Por ejemplo, las variables anteriores se pueden declarar también así:

```
let n = 10
let str = "Hola"
let array = [1,2,3,4,5]
```

Aunque no hayamos declarado explícitamente el tipo de las variables, el compilador les ha asignado el tipo correspondiente. Por ejemplo, no podemos asignarles un valor de distinto tipo:

```
var x = 5
x = 4 // correcto
x = 6.0 // error
// error: cannot assign value of type 'Double' to type 'Int'
// x = 5.0
//      ^~~
//      Int( )
```

El compilador indica el error e incluso sugiere una posible solución del mismo. En este caso llamar al constructor `Int()` pasándole un `Double` como parámetro.

## Lenguaje multi-paradigma

Swift permite combinar características funcionales con características imperativas y de programación orientada a objetos. Veremos en este tema muchas características funcionales que podremos utilizar en cualquier programa Swift que desarrollemos.

Por ejemplo, cuando declaramos una variable podemos declararla como mutable, usando la declaración `var`, o como inmutable, usando la declaración `let`. Si queremos utilizar un enfoque funcional preferiremos siempre declarar las variables con `let`.

```
var x = 10
x = 20 // x es mutable
let y = 10
y = 20 // error: y es inmutable
```

Una ventaja de la inmutabilidad es que permite que el compilador de Swift optimice el código de forma muy eficiente. De hecho, el propio compilador nos indica que es preferible definir una variable como `let` si no la vamos a modificar:

```
func saluda(nombre: String) -> String {
    var saludo = "Hola " + nombre + "!"
    return saludo
}
//warning: variable 'saludo' was never mutated; consider changing to 'let'
//constant
//    var saludo = "Hola " + nombre
```

```
//      ~~~^
//      let
```

## Inmutabilidad

Una de las características funcionales importantes de Swift es el énfasis en la inmutabilidad para reforzar la seguridad del lenguaje.

Hemos visto que la palabra clave `let` permite definir constantes y que Swift recomienda su uso si el valor que definimos es un valor que no va a ser modificado.

El valor asignado a una constante `let` puede no conocerse en tiempo de compilación, sino que puede ser obtenido en tiempo de ejecución como un valor devuelto por una función:

```
let respuesta: String = respuestaUsuario.respuesta()
```

Al declarar una variable como `let` se bloquea su contenido y no se permite su modificación. Una de las ventajas del paradigma funcional y de la inmutabilidad es que garantiza que el código que escribimos no tiene efectos laterales y puede ser ejecutado sin problemas en entornos multi-procesador o multi-hilo.

## Creación de nuevas estructuras y mutación

En la [biblioteca estándar de Swift](#) existen una gran cantidad de estructuras (como `Int`, `Double`, `Bool`, `String`, `Array`, `Dictionary`, etc.) que tienen dos tipos de métodos: métodos que mutan la estructura y métodos que devuelven una nueva estructura. Cuando estemos escribiendo código con estilo funcional deberemos utilizar siempre estos últimos métodos, los que construyen estructuras nuevas.

Por ejemplo, en el struct `Array` se define el método `sort` y el método `sorted`. El primero ordena el array con mutación y el segundo devuelve una copia ordenada, sin modificar el array original. En el siguiente código no se modifica el array original, sino que se construye un array nuevo ordenado:

```
// Código recomendable en programación funcional
// porque utiliza el método sorted que devuelve una
// copia del array original
let miArray = [10, -1, 3, 80]
let arrayOrdenado = miArray.sorted()
print(miArray)
print(arrayOrdenado)
// Imprime:
// [10, -1, 3, 80]
// [-1, 3, 10, 80]
```

Este código es el recomendable cuando estemos escribiendo código con un estilo de programación funcional.

Sin embargo, el siguiente código es imperativo y utiliza la mutación del array original:

```
// Código no recomendable en programación funcional
// porque utiliza el método sort que muta el array original
var miArray = [10, -1, 3, 80]
miArray.sort()
print(miArray)
// Imprime:
// [-1, 3, 10, 80]
```

Otro ejemplo es en la forma de añadir elementos a un array. Podemos hacerlo con un enfoque funcional, usando el operador `+` que construye un array nuevo:

```
// Código recomendable en programación funcional
let miArray = [10, -1, 3, 80]
let array2 = miArray + [100]
print(array2)
// Imprime:
// [10, -1, 3, 80, 100]
```

Y podemos hacerlo usando un enfoque imperativo, con el método `append`:

```
// Código no recomendable en programación funcional
var miArray = [10, -1, 3, 80]
miArray.append(100)
print(miArray)
// Imprime:
// [10, -1, 3, 80, 100]
```

!!! Important "Importante" En programación funcional debemos usar siempre los métodos **que no modifican las estructuras**. Así evitaremos los efectos laterales y nuestro código funcionará correctamente en entornos multi-hilo.

Cuando definimos una variable de tipo `let` el valor que se asigne a esa variable se convierte en inmutable. Si se trata de una estructura o una clase con métodos mutables el compilador dará un error. Por ejemplo:

```
let miArray = [10, -1, 3, 80]
miArray.append(100)
// error: cannot use mutating member on immutable value: 'miArray' is a 'let'
constant
```

Otro ejemplo. El método `append(_ :)` de un `String` es un método mutable. Si definimos una cadena con `let` no podremos modificarla y daría error el siguiente código:

```
var cadenaMutable = "Hola"
let cadenaInmutable = "Adios"
```

```

cadenaMutable.append(cadenaInmutable) // cadenaMutable es "HolaAdios"
cadenaInmutable.append("Adios")
// error: cannot use mutating member on immutable value: 'cadenaInmutable' is a
'let' constant

```

## Funciones

### Definición de una función en Swift

Para definir una función en Swift se debe usar la palabra `func`, definir el nombre de la función, sus parámetros y el tipo de vuelto. El valor devuelto por la función se debe devolver usando la palabra `return`.

Código de la función `saluda(nombre:)`:

```

func saluda(nombre: String) -> String {
    let saludo = "Hola, " + nombre + "!"
    return saludo
}

```

Para invocar a la función `saluda(nombre:)`:

```

print(saluda(nombre:"Ana"))
print(saluda(nombre:"Pedro"))
// Imprime "Hola, Ana!"
// Imprime "Hola, Pedro!"

```

### Etiquetas de argumentos y nombres de parámetros

Cada parámetro de una función tiene una etiqueta del argumento y un nombre de parámetro. La etiqueta del argumento se usa cuando se llama a la función y el nombre del parámetro se usa internamente en su implementación. Por defecto, los parámetros usan su nombre de parámetro como etiqueta del argumento:

```

func unaFuncion(primerNombreParametro: Int, segundoNombreParametro: Int) {
    // En el cuerpo de la función, primerNombreParametro y
    // segundoNombreParametro se refieren a los valores de los
    // argumentos del primer y el segundo parámetro
}
unaFuncion(primerNombreParametro: 1, segundoNombreParametro: 2)

```

Es posible hacer distintos la etiqueta del argumento del nombre del parámetro:

```

func saluda(nombre: String, de ciudad: String) -> String {
    return "Hola \(nombre)! Me alegra de que hayas podido visitarnos desde \
(ciudad)."

```

```

    }
    print(saluda(nombre: "Bill", de: "Cupertino"))
    // Imprime "Hola Bill! Me alegra de que hayas podido visitarnos desde Cupertino."
}

```

En este caso el nombre externo del parámetro (el que usamos al invocar la función) es `de` y el nombre interno (el que se usa en el cuerpo de la función) es `ciudad`.

Otro ejemplo, la siguiente función `concatena(palabra:con:)`:

```

func concatena(palabra str1: String, con str2: String) -> String {
    return str1+str2
}

print(concatena(palabra:"Hola", con:"adios"))

```

Si no se quiere una etiqueta del argumento para un parámetro, se puede escribir un subrayado (`_`) en lugar de una etiqueta del argumento explícita para ese parámetro. Esto nos permite llamar a la función sin usar un nombre de parámetro. Por ejemplo, la función `max(_:_:)` y la función `divide(_:entre:)`:

```

func max(_ x:Int, _ y: Int) -> Int {
    if x > y {
        return x
    } else {
        return y
    }
}

print(max(10,3))

func divide(_ x:Double, entre y: Double) -> Double {
    return x / y
}

print(divide(30, entre:4))

```

El perfil de la función está formado por el nombre de la función, las etiquetas de los argumentos y el tipo devuelto por la función. En la documentación de las funciones usaremos las etiquetas separadas por dos puntos. Por ejemplo, las funciones anteriores son `max(_:_:)` y `divide(_:entre:)`.

Las etiquetas de los argumentos son parte del nombre de la función. Es posible definir funciones distintas con sólo distintos nombres de argumentos, como las siguientes funciones `mitad(par:)` y `mitad(impar:)`:

```

func mitad(par: Int) -> Int{
    return par/2
}

```

```

func mitad(impar: Int) -> Int{
    return (impar+1)/2
}

print(mitad(par: 8))
// Imprime 4
print(mitad(impar: 9))
// Imprime 5

```

## Parámetros y valores devueltos

Es posible definir funciones sin parámetros:

```

func diHolaMundo() -> String {
    return "hola, mundo"
}
print(diHolaMundo())
// Imprime "hola, mundo"

```

Podemos definir funciones sin valor devuelto. Por ejemplo, la siguiente función `diAdios(nombre:)`. No hay que escribir flecha con el tipo devuelto. Cuidado, no sería propiamente programación funcional.

```

func diAdios(nombre: String) {
    print("Adiós, \(nombre)!")
}
diAdios(nombre: "Dave")
// Imprime "Adiós, Dave!"

```

Es posible devolver múltiples valores, construyendo una tupla. Por ejemplo, la siguiente función `ecuacion(a:b:c:)` calcula las dos soluciones de una ecuación de segundo grado:

```

func ecuacion(a: Double, b: Double, c: Double) -> (pos: Double, neg: Double) {
    let discriminante = b*b-4*a*c
    let raizPositiva = (-b + discriminante.squareRoot()) / 2*a
    let raizNegativa = (-b - discriminante.squareRoot()) / 2*a
    return (raizPositiva, raizNegativa)
}

```

Recordemos (consultar el seminario de Swift) que podemos acceder a los valores de la tupla por posición:

```

let resultado = ecuacion(a: 1, b: -5, c: 6)
print("Las raíces de la ecuación son \(resultado.0) y \(resultado.1)")
//Imprime "Las raíces de la ecuación son 3.0 y 2.0"

```

En este caso en la definición del tipo devuelto por la función estamos etiquetando esos valores con las etiquetas `pos` y `neg`. De esta forma podemos acceder a los componentes de la tupla usando esas etiquetas definidas:

```
let resultado = ecuacion(a: 1, b: -5, c: 6)
print("Las raíces de la ecuación son \(resultado.pos) y \(resultado.neg)")
//Imprime "Las raíces de la ecuación son 3.0 y 2.0"
```

## Recursión

Veamos algunos ejemplos de funciones recursivas en Swift.

Primero una función `suma(hasta:)` que devuelve la suma desde 0 hasta el número que le pasamos como parámetro.

```
func suma(hasta x: Int) -> Int {
    if x == 0 {
        return 0
    } else {
        return x + suma(hasta: x - 1)
    }
}

print(suma(hasta: 5))
// Imprime "15"
```

También es posible definir recursiones que recorran arrays de una forma similar a cómo trabajábamos en Scheme. Los arrays en Swift no funcionan exactamente como las listas de Scheme (no son listas de parejas), pero podemos obtener el primer elemento y el resto de la siguiente forma.

```
let a = [10, 20, 30, 40, 50, 60]
let primero = a[0]
let resto = Array(a.dropFirst())
```

En `primero` se guarda el número 10. En `resto` se guarda el `Array` del 20 al 60. El método `dropFirst` devuelve una `ArraySlice`, que es una vista de un rango de elementos del array, en este caso el que va desde la posición 1 hasta la 5 (la posición inicial de un array es la 0). Es necesario el constructor `Array` para convertir ese `ArraySlice` en un `Array`.

Usando las instrucciones anteriores podemos definir la función recursiva que suma los valores de un Array de la siguiente forma similar a cómo lo hacíamos en Scheme:

```
func sumaValores(_ valores: [Int]) -> Int {
    if (valores.isEmpty) {
        return 0
    }
```

```

    } else {
        let primero = valores[0]
        let resto = Array(valores.dropFirst())
        return primero + sumaValores(resto)
    }
}

print(sumaValores([1,2,3,4,5,6,7,8]))
// Imprime "36"

```

Un último ejemplo es la siguiente función `minMax(array:)` que devuelve el número más pequeño y más grande de un array de enteros:

```

func minMax(array: [Int]) -> (min: Int, max: Int) {
    if (array.count == 1) {
        return (array[0], array[0])
    } else {
        let primero = array[0]
        let resto = Array(array.dropFirst())

        // Llamada recursiva que devuelve el mínimo y el máximo del
        // resto del array
        let minMaxResto = minMax(array: resto)

        let minimo = min(primer, minMaxResto.min)
        let maximo = max(primer, minMaxResto.max)
        return (minimo, maximo)
    }
}

let limites = minMax(array: [8, -6, 2, 100, 3, 71])
print("El mínimo es \(limites.min) y el máximo es \(limites.max)")
// Imprime "El mínimo es -6 y el máximo es 100"

```

En este ejemplo nos apartamos un poco de la solución vista en Scheme porque permitimos pasos de ejecución que inicializan variables. Pero no nos salimos del paradigma funcional, porque todas son variables inmutables definidas con `let`.

## Tipos función

En Swift las funciones son objetos de primera clase y podemos asignarlas a variables, pasárlas como parámetro o devolverlas como resultado de otra función. Al ser un lenguaje fuertemente tipado, las variables, parámetros o resultados deben ser objetos de tipo función.

Cada función tiene un tipo específico, definido por el tipo de sus parámetros y el tipo del valor devuelto.

```

func sumaDosInts(a: Int, b: Int) -> Int {
    return a + b
}

```

```
func multiplicaDosInts(a: Int, b: Int) -> Int {
    return a * b
}
```

El tipo de estas funciones es `(Int, Int) -> Int`, que se puede leer como:

"Un tipo función que tiene dos parámetros, ambos de tipo `Int` y que devuelve un valor de tipo `Int`".

En Swift se puede usar un tipo función de la misma forma que cualquier otro tipo, por ejemplo asignando la función a una variable:

```
var f = sumaDosInts
print(f(2,3))
// Imprime "5"
f = multiplicaDosInts
print(f(2,3))
// Imprime "6"
```

La variable `f` es una variable de tipo `(Int, Int) -> Int`, o sea, una variable que contiene funciones de dos argumentos `Int` que devuelven un `Int`.

!!! Note "Nota" Habrás notado que al invocar a `f` no se ponen etiquetas en los argumentos. De hecho, si las pusiéramos el compilador de Swift se quejaría:

```
```swift
print(f(a:2, b:3))
//error: extraneous argument labels 'a:b:' in call
```

```

Esto es debido a que al ser `f` una variable se le puede asignar cualquier función que tenga el perfil `(Int, Int) -> Int` sin tener en cuenta las etiquetas de los argumentos.

## Funciones que reciben otras funciones

Podemos usar un tipo función en parámetros de otras funciones:

```
func printResultado(funcion: (Int, Int) -> Int, _ a: Int, _ b: Int) {
    print("Resultado: \(funcion(a, b))")
}
printResultado(funcion: sumaDosInts, 3, 5)
// Prints "Resultado: 8"
```

La función `printResultado(funcion:_:_:_)` toma como primer parámetro otra función que recibe dos `Int` y devuelve un `Int`, y como segundo y tercer parámetro dos `Int`. Y en el cuerpo llama a la función que se pasa

como parámetro con los argumentos **a** y **b**.

Veamos otro ejemplo, que ya vimos en Scheme. Supongamos que queremos calcular el sumatorio desde **a** hasta **b** en el que aplicamos una función **f** a cada número que sumamos:

```
sumatorio(a, b, f) = f(a) + f(a+1) + f(a+2) + ... + f(b)
```

Recordamos que se resuelve con la siguiente recursión:

```
sumatorio(a, b, f) = f(a) + sumatorio(a+1, b, f)
sumatorio(a, b, f) = 0 si a > b
```

Veamos cómo se implementa en Swift:

```
func sumatorio(desde a: Int, hasta b: Int, func f: (Int) -> Int) -> Int {
    if a > b {
        return 0
    } else {
        return f(a) + sumatorio(desde: a + 1, hasta: b, func: f)
    }
}

func identidad(_ x: Int) -> Int {
    return x
}

func doble(_ x: Int) -> Int {
    return x + x
}

func cuadrado(_ x: Int) -> Int {
    return x * x
}

print(sumatorio(desde: 0, hasta: 10, func: identidad)) // Imprime 55
print(sumatorio(desde: 0, hasta: 10, func: doble)) // Imprime 110
print(sumatorio(desde: 0, hasta: 10, func: cuadrado)) // Imprime 385
```

## Funciones en estructuras

Como cualquier otro tipo Las funciones pueden también incluirse en estructuras de datos compuestas, como arrays:

```
var funciones = [identidad, doble, cuadrado]
print(funciones[0](10)) // 10
```

```
print(funciones[1](10)) // 20
print(funciones[2](10)) // 100
```

El tipo de la variable `funciones` sería `[(Int) -> Int]`.

Al ser Swift fuertemente tipado, no podríamos hacer un array con distintos tipos de funciones. Por ejemplo el siguiente código daría un error:

```
func suma(_ x: Int, _ y: Int) -> Int {
    return x + y
}
// La siguiente línea genera un error
var misFunciones = [doble, cuadrado, suma]
// error: heterogenous collection literal could only be inferred to
// '[Any]'; add explicit type annotation if this is intentional
```

## Funciones que devuelven otras funciones

Por último, veamos un ejemplo de funciones que devuelven otras funciones.

Es un ejemplo sencillo, una función que devuelve otra que suma 10:

```
func construyeSumador10() -> (Int) -> Int {
    func suma10(x: Int) -> Int {return x+10}
    return suma10
}

var g = construyeSumador10()
print(g(20))
// Imprime 30
```

La función devuelta por `construyeSumador10()` es una función con el tipo `(Int) -> Int` (recibe un parámetro entero y devuelve un entero). En la llamada a `construyeSumador10()` se crea esa función y se asigna a la variable `g`.

Estas funciones devueltas se denominan **clausuras**. Más adelante hablaremos algo más de ellas. Veremos también más adelante que es posible usar **expresiones de clausura** que construyen clausuras anónimas.

Podemos modificar el ejemplo anterior, haciendo que la función `construyeSumador` reciba el número a sumar como parámetro:

```
func construyeSumador(inc: Int) -> (Int) -> Int {
    func suma(x: Int) -> Int {return x+inc}
    return suma
}
```

```

var f2 = construyeSumador(inc: 10)
var f3 = construyeSumador(inc: 100)
print(f2(20))
// Imprime "30"
print(f3(20))
// Imprime "120"

```

Invocamos dos veces a `construyeSumador(inc:)` y guardamos las clausuras construidas en las variables `f2` y `f3`. En `f2` se guarda una función que suma `10` a su argumento y en `f3` otra que suma `100`.

## Tipos

Entre las ventajas del uso de tipos está la detección de errores en los programas en tiempo de compilación o las ayudas del entorno de desarrollo para autocompletar código. Entre los inconvenientes se encuentra la necesidad de ser más estrictos a la hora de definir los parámetros y los valores devueltos por las funciones, lo que impide la flexibilidad de Scheme.

Se utilizan tipos para definir los posibles valores de:

- variables
- parámetros de funciones
- valores devueltos por funciones

Tal y como hemos visto cuando hemos comentado que Swift es fuertemente tipado las definiciones de tipos van precedidas de dos puntos en las variables y parámetros, o de una flecha (`->`) en la definición de los tipos de los valores devueltos por una función:

```

let valorDouble : Double = 3.0
let unaCadena: String = "Hola"

func calculaEstadisticas(valores: Array<Int>) -> (min: Int, max: Int, media: Int)
{
    ...
}

```

En Swift existen dos clases de tipos: tipos con nombre y tipos compuestos.

### Tipos con nombre

Un tipo con nombre es un tipo al que podemos dar un nombre determinado cuando se define. Por ejemplo, al definir un nombre de una clase o de un enumerado estamos también definiendo un nombre de un tipo.

En Swift es posible definir los siguientes tipos con nombre:

- nombres de clases
- nombres de estructuras
- nombres de enumeraciones
- nombres de protocolos

Por ejemplo, instancias de una clase definida por el usuario llamada `MiClase` tienen el tipo `MiClase`.

Además de los tipos definidos por el usuario, la biblioteca estándar de Swift tiene un gran número de tipos predefinidos. A diferencia de otros lenguajes, estos tipos no son parte del propio lenguaje sino que se definen en su mayoría como estructuras implementadas en esta biblioteca estándar. Por ejemplo, arrays, diccionarios o incluso los tipos más básicos como `String` o `Int` están construidos en esa biblioteca. La implementación de estos elementos está disponible en abierto en el [sitio GitHub de Swift](#).

## Tipos compuestos

Los tipos compuestos son tipos sin nombre. En Swift se definen dos: tuplas y tipos función. Un tipo compuesto puede tener tipos con nombre y otros tipos compuestos. Por ejemplo la tupla `(Int, (Int, Int))` contiene dos elementos: el primero es el tipo con nombre `Int` y el segundo el tipo compuesto que define la tupla `(Int, Int)`. Los tipos función los hemos visto previamente.

```
let tupla: (Int, Int, String) = (2, 3, "Hola")
let otraTupla: (Int, Int, String) = (5, 8, "Adios")

func sumaTupla(tupla t1: (Int, Int), con t2: (Int, Int)) -> (Int, Int) {
    return (t1.0 + t2.0, t1.1 + t2.1)
}

print(sumaTupla(tupla: (tupla.0, tupla.1),
                con: (otraTupla.0, otraTupla.1)))

// Imprime (7, 11)
```

## Typealias

En Swift se define la palabra clave `typealias` para darle un nombre asignado a cualquier otro tipo. Ambos tipos son iguales a todos los efectos (es únicamente azúcar sintáctico).

Por ejemplo, en el siguiente código definimos un `typealias` llamado `Resultado` que corresponde a una tupla con dos `Int` correspondientes al resultado de un partido de fútbol. Una vez definido, podemos usarlo como un tipo. La función `quiniela(partido:)` devuelve un `String` correspondiente al resultado de la quiniela de un partido:

```
typealias Resultado = (Int, Int)

func quiniela(partido: Resultado) -> String {
    switch partido {
        case let (goles1, goles2) where goles1 < goles2:
            return "Dos"
        case let (goles1, goles2) where goles1 > goles2:
            return "Uno"
        default:
            return "Equis"
```

```

    }
}

print(quiniela(partido: (1,3)))
// Imprime "Dos"
print(quiniela(partido: (2,2)))
// Imprime "Equis"

```

## Tipos valor y tipos referencia

En Swift existen dos tipos de construcciones que forman la base de la programación orientada a objetos: las estructuras (*structs*) y las clases. En el tema siguiente hablaremos sobre ello.

En la [biblioteca estándar de Swift](#) la mayor parte de los tipos definidos (como `Int`, `Double`, `Bool`, `String`, `Array`, `Dictionary`, etc.) son estructuras, no clases.

Una de las diferencias más importantes entre estructuras y clases es su comportamiento en una asignación: las estructuras tienen una **semántica de copia** (son tipos valor) y las clases tienen una **semántica de referencia** (son tipos referencia).

Un *tipo valor* es un tipo que tiene semántica de copia en las asignaciones y cuando se pasan como parámetro en llamadas a funciones.

Los tipos valor son muy útiles porque evitan los efectos laterales en los programas y simplifican el comportamiento del compilador en la gestión de memoria. Al no existir referencias, se simplifica enormemente la gestión de memoria de estas estructuras. No es necesario llevar la cuenta de qué referencias apuntan a un determinado valor, sino que se puede liberar la memoria en cuanto se elimina el ámbito actual.

Frente a un tipo valor, un tipo de referencia es aquel en los que los valores se asignan a variables con una semántica de referencia. Cuando se realizan varias asignaciones de una misma instancia a distintas variables todas ellas guardan una referencia a la misma instancia. Si la instancia se modifica, todas las variables reflejarán el nuevo valor. Cuando veamos las clases en el próximo tema veremos algunos ejemplos.

Veamos ahora algunos ejemplos de copia por valor en estructuras.

Por ejemplo, si asignamos una cadena a otra, se realiza una copia:

```

var str1 = "Hola"
var str2 = str1
str1.append("Adios")
print(str1) // Imprime "HolaAdios"
print(str2) // Imprime "Hola"

```

Los arrays también son estructuras y, por tanto, también tienen semántica de copia:

```

var array1 = [1, 2, 3, 4]
var array2 = array1
array1[0] = 10

```

```
print(array1) // [10, 2, 3, 4]
print(array2) // [1, 2, 3, 4]
```

A diferencia de otros lenguajes como Java, los parámetros de una función siempre son inmutables y se pasan por copia, para reforzar el carácter funcional de las funciones. Por ejemplo, es incorrecto escribir lo siguiente:

```
func ponCero(array: [Int], pos: Int) {
    array[pos] = 0
// error: cannot assign through subscript: 'array' is a 'let' constant
}
```

Se podría pensar que es muy costoso copiar un array entero. Por ejemplo, si asignamos o pasamos como parámetro un array de 1000 elementos. Pero no es así. El compilador de Swift optimiza estas sentencias y sólo realiza la copia en el momento en que hay una modificación de una de las variables que comparten el array. Es lo que se llama *copy on write*.

## Enumeraciones

Las enumeraciones definen un tipo con un valor restringido de posibles valores:

```
enum Direccion {
    case norte
    case sur
    case este
    case oeste
}
```

Cualquier variable del tipo `Direccion` solo puede tener uno de los cuatro valores definidos. Se obtiene el valor escribiendo el nombre de la enumeración, un punto y el valor definido. Si el tipo de enumeración se puede inferir no es necesario escribirlo.

```
let hemosGirado = true
var direccionActual = Direccion.norte
if hemosGirado {
    direccionActual = .sur
}
```

En sentencias switch:

```
let direccionAIr = Direccion.sur
switch direccionAIr {
case .norte:
    print("Nos vamos al norte")
case .sur:
```

```

print("Cuidado con los pinguinos")
case .este:
    print("Donde nace el sol")
case .oeste:
    print("Donde el cielo es azul")
}
// Imprime "Cuidado con los pinguinos"

```

Otro ejemplo:

```

enum Planeta {
    case mercurio, venus, tierra, marte, jupiter, saturno, urano, neptuno
}

```

Y, por último, es más correcto definir el resultado de una quiniela con un enumerado en lugar de con un **String**:

```

enum Quiniela {
    case uno, equis, dos
}

```

## Valores brutos de enumeraciones

Es posible asignar a las constantes del enumerado un valor concreto de un tipo subyacente, por ejemplo enteros:

```

enum Quiniela: Int {
    case uno=1, equis=0, dos=2
}

```

Se puede obtener el valor bruto a partir del propio tipo o de una variable del tipo, usando **rawValue**:

```

// Obtenemos el valor bruto a partir del tipo
let valorEquis: Int = Quiniela.equis.rawValue

// Obtenemos el valor bruto a partir de una variable
let res = Quiniela.equis
let valorEquis = res.rawValue

```

También se puede asignar los valores de forma implícita, dando un valor a la primera constante. Las siguientes tienen el valor consecutivo:

```
enum Planeta: Int {
    case mercurio=1, venus, tierra, marte, jupiter, saturno, urano, neptuno
}
let posicionTierra = Planeta.tierra.rawValue
// posicionTierra es 3
```

Podemos escoger cualquier tipo subyacente. Por ejemplo el tipo **Character**:

```
enum CaracterControlASCII: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

El carácter nueva línea (*lineFeed*) se puede obtener de la siguiente forma:

```
let nuevaLinea = CaracterControlASCII.LineFeed.rawValue
```

Y por último, se puede definir como tipo subyacente **String** y los valores brutos de las constantes serán sus nombres convertidos a cadenas:

```
enum Direccion: String {
    case norte, sur, este, oeste
}
let direccionAtardecer = Direccion.oeste.rawValue
// direccionAtardecer es "oeste"
```

Cuando se definen valores brutos es posible inicializar el enumerado de una forma similar a una estructura o una clase pasando el valor bruto. Devuelve el valor enumerado correspondiente o **nil** (un opcional):

```
let posiblePlaneta = Planeta(rawValue: 7)
// posiblePlaneta es de tipo Planeta? y es igual a Planeta.urano
```

## Enumeraciones instanciables

Una característica singular de las enumeraciones en Swift es que permiten definir valores variables asociados a cada caso de la enumeración, creando algo muy parecido a una instancia de la enumeración.

### Valores asociados a instancias de enumeraciones

Veamos un ejemplo inicial muy sencillo, con una enumeración con un único caso, en el que se define una variable de tipo **Int**:

```
enum Prueba {
    case x(Int)
}
```

Esta notación obliga a definir un valor concreto del `Int` asociado al caso `x` en el momento de creación del enumerado:

```
let valor1 = Prueba.x(10)
let valor2 = Prueba.x(40)
```

Las variables `valor1` y `valor2` son de tipo `Prueba` y tiene como valor el caso `x`, y el entero asociado a ese caso (`10` y `40` en cada instancia). Son parecidas a instancias de una clase.

Para obtener el valor asociado debemos usar una expresión `case let` en una sentencia `switch` con una variable a la que se asigna el valor. Por ejemplo, en el siguiente código el valor del enumerado se asigna a la variable `a`:

```
switch valor1 {
    case let .x(a):
        print("Valor asociado a x: \(a)")
}
// Imprime "Valor asociado a x: 10"
```

No hay que confundir un valor asociado a un caso y un valor bruto: el valor bruto de un caso de enumeración es el mismo para todas las instancias, mientras que el valor asociado es distinto y se proporciona cuando se define el valor concreto de la enumeración.

Cuando unimos a la característica del valor asociado la posibilidad de los enumerados de tener más de un caso tenemos lo que en otros lenguajes de programación se llaman *uniones etiquetadas* o *variantes*.

Por ejemplo, podemos definir un enumerado que permita guardar un `Int` o un `String`:

```
enum Multiple {
    case x(Int)
    case str(String)
}
```

De esta forma, podemos crear valores de tipo `Multiple` que contienen un `Int` (instanciando el caso `x`) o un `String` (instanciando el caso `str`):

```
let valor3 = Multiple.x(10)
let valor4 = Multiple.str("Hola")
```

Y podemos definir una función que reciba instancias de tipo `Multiple` y use un `switch` para comprobar qué caso tienen instanciado:

```
func imprime(multiple: Multiple) {
    switch multiple {
        case let .x(a):
            print("Multiple tiene un Int: \(a)")
        case let .str(s):
            print("Multiple tiene un String: \(s)")
    }
}

imprime(multiple: valor3)
// Imprime "Multiple tiene un Int: 10"
imprime(multiple: valor4)
// Imprime "Multiple tiene un String: Hola"
```

En un último ejemplo podemos ver que el tipo del caso también puede ser un tipo compuesto, como una tupla. Usamos un enum para definir posibles valores de un código de barras, en el que incluimos dos posibles tipos de código de barras: el código de barras lineal (denominado UPC) y el código QR:

```
enum CódigoBarras {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}
```

Se lee de la siguiente forma: "Definimos un tipo enumerado llamado `CódigoBarras`, que puede tomar como valor un `upc` (código de barras lineal) con un valor asociado de tipo `(Int, Int, Int, Int)` (una tupla de 4 enteros que representan los 4 números que hay en los códigos de barras lineales) o un valor `qrCode` con valor asociado de tipo `String`".

Veamos un ejemplo de uso, en el que creamos un código de barras de producto de tipo UPC, después lo modificamos a otro de tipo código QR y por último lo imprimimos:

```
var códigoBarrasProducto = CódigoBarras.upc(8, 85909, 51226, 3)
códigoBarrasProducto = .qrCode("ABCDEFGHIJKLMNP")

switch códigoBarrasProducto {
    case let .upc(sistemaNumeración, fabricante, producto, control):
        print("UPC: \(sistemaNumeración), \(fabricante), \(producto), \(control).")
    case let .qrCode(códigoProducto):
        print("Código QR: \(códigoProducto).")
}
// Imprime "Código QR : ABCDEFGHIJKLMNOP."
```

## Enumeraciones recursivas

Es posible combinar las características de las enumeraciones con valor con la recursión para crear enumeraciones recursivas. Hay que preceder la palabra clave `enum` con `indirect`:

```
indirect enum ExpresionAritmetica {
    case numero(Int)
    case suma(ExpresionAritmetica, ExpresionAritmetica)
    case multiplicacion(ExpresionAritmetica, ExpresionAritmetica)
}

let cinco = ExpresionAritmetica.numero(5)
let cuatro = ExpresionAritmetica.numero(4)
let suma = ExpresionAritmetica.suma(cinco, cuatro)
let producto = ExpresionAritmetica.multiplicacion(suma,
    ExpresionAritmetica.numero(2))
```

Es muy cómodo manejar enumeraciones recursivas de forma recursiva:

```
func evalua(expresion: ExpresionAritmetica) -> Int {
    switch expresion {
        case let .numero(valor):
            return valor
        case let .suma(izquierda, derecha):
            return evalua(expresion: izquierda) + evalua(expresion: derecha)
        case let .multiplicacion(izquierda, derecha):
            return evalua(expresion: izquierda) * evalua(expresion: derecha)
    }
}

print(evalua(expresion: producto))
// Imprime 18
```

Otro ejemplo de enums recursivos, para definir un tipo de datos `Lista` similar al que vimos en Scheme. La lista puede ser una lista vacía o puede contener dos elementos: un valor `Int` y otra lista:

```
indirect enum Lista {
    case vacia
    case nodo(Int, Lista)
}
```

Para crear una lista de tipo `nodo` deberemos dar un valor entero (el valor de la cabeza de la lista) y otra lista (el resto de la lista). También podemos crear una lista vacía.

Por ejemplo, podemos crear la lista `(10, 20, 30)` de la siguiente manera:

```
let lista1 = Lista.nodo(30, Lista.vacia)
let lista2 = Lista.nodo(20, lista1)
```

```
let lista3 = Lista.nodo(10, lista2)
```

Podríamos crear esta misma lista de una forma más abreviada:

```
let lista: Lista = .nodo(10, .nodo(20, .nodo(30, .vacia)))
```

Una vez definido el tipo enumerado, podemos definir funciones que trabajen con él. La siguiente función, por ejemplo, es una función recursiva que recibe una lista y devuelve la suma de sus elementos. Funciona de una forma muy similar a la definición que hicimos en Scheme:

```
func suma(lista: Lista) -> Int {
    switch lista {
        case .vacia:
            return 0
        case let .nodo(car, cdr):
            return car + suma(lista: cdr)
    }
}

let z: Lista = .nodo(20, .nodo(10, .vacia))

print(suma(lista: z))
// Imprime 30
```

Podemos también definir una función recursiva `construye(lista:[Int])` que devuelve una lista a partir de una array de enteros:

```
func construye(lista: [Int]) -> Lista {
    if (lista.isEmpty) {
        return Lista.vacia
    } else {
        let primero = lista[0]
        let resto = Array(lista.dropFirst())
        return Lista.nodo(primeros, construye(lista: resto))
    }
}

let lista2 = construye(lista: [1,2,3,4,5])

print(suma(lista: lista2))
// Imprime 15
```

## Opcionales

Una de las características principales que Swift intenta promover es la seguridad y la robustez. Debe ser difícil que el desarrollador escriba código con errores y que rompa la aplicación. Por ejemplo, la comprobación estática de los tipos de datos o el manejo automático de la gestión de memoria son dos características del lenguaje que van en esta dirección.

Otro de los elementos más importantes del lenguaje para promover la seguridad son los opcionales. Vamos a estudiar su uso y utilidad.

En muchos lenguajes existe el concepto de *valor vacío*. Por ejemplo, en Java se usa *null* o en Python *None*.

!!! Note "Nota" Tony Hoare introdujo el concepto de *Null* en ALGOL, en 1965. En una conferencia en 2009 habla sobre esta idea y la considera un costoso error: [Null References: The Billion Dollar Mistake](#).

El concepto de *null* es un concepto peligroso, como lo saben bien los desarrolladores Java. En Java, si intentamos usar una variable que contiene *null* se produce la típica excepción *null pointer exception* y la aplicación se rompe. Todos hemos caído en este error, y con más frecuencia de la que sería deseable.

En Swift también existe el valor nulo. La forma de representarlo es el identificador *nil*.

La característica de seguridad que introduce Swift con respecto a Java y a otros lenguajes es que no es posible asignar *nil* a una variable de un tipo normal.

Por ejemplo, la siguiente línea daría un error de compilación:

```
let cadena: String = nil
// error: 'nil' cannot initialize specified type 'String'
```

Si queremos utilizar *nil* debemos declarar la variable usando lo que se denomina **tipo opcional**:

```
var cadena: String? = "Hola"
cadena = nil
```

El tipo *String?* indica que podemos tener un valor *nil* o un valor del tipo original. Primero estamos definiendo la variable *cadena* del tipo *String?* (*String* opcional) y le estamos asignando un valor determinado (de tipo *String*). Y después le asignamos *nil*.

El uso de opcionales es necesario en situaciones en las que podemos obtener un valor desconocido. Por ejemplo, en alguna función en la que pedimos un valor al usuario y el usuario puede no introducir ninguno. O en estructuras de datos en las que hacemos búsquedas que pueden no devolver ningún valor, como en un diccionario:

```
var edades = [
    "Raquel": 30,
    "Pedro": 22,
]
let edad1 = edades["Raquel"]
let edad2 = edades["Ana"]
```

En el código anterior definimos un diccionario `edades` con claves de tipo `String` y valores `Int`. Después buscamos en el diccionario por la clave `"Raquel"` y se devuelve el valor `30`, que se guarda en la variable `edad1`. Cuando se busca por la clave `"Ana"` se devuelve un `nil` porque no está definida. Por ello, la variable `edad2` será de tipo `Int?` (`Int` opcional) y contendrá un `nil`.

Un valor opcional no puede ser usado directamente. Primero debemos comprobar si el valor es distinto de `nil` y sólo después podremos usarlo.

Para reforzar esto, Swift *esconde* o *envuelve* (*wrap*) el valor real del opcional y obliga a llamar al operador `!` para *desenvolverlo* (*unwrap*) y usarlo. Este operador se denomina de **desenvoltura forzosa** (*forced unwrapping*).

Por ejemplo, el siguiente código produce un error de compilación porque intentamos usar un opcional sin desenvolverlo:

```
var x: Int? = 10
let y = x + 10
// error: value of optional type 'Int?' must be unwrapped to a value of type 'Int'
```

Para usar el valor asignado a `x` debemos desenvolverlo con el operador `!`:

```
var x: Int? = 10
let y = x! + 10
print(y)
// Imprime "20"
```

Podemos definir como opcional variables, parámetros o valores devueltos por funciones de cualquier tipo, añadiéndoles la interrogación al final.

Por ejemplo, la siguiente función `max` es una función que devuelve un `Int?`, un entero opcional en el caso de que se le pase un array vacío. Al devolver un opcional, debemos desenvolver el valor devuelto cuando queramos usarlo como `Int` (por ejemplo, en la llamada recursiva).

```
func max(array:[Int]) -> Int? {
    if (array.isEmpty) {
        return nil
    } else if (array.count == 1) {
        return array[0]
    } else {
        let primero = array[0]
        let resto = Array(array.dropFirst())
        return max(primer, max(array:resto)!)
    }
}

let maximo = max(array:[10, 200, -100, 2])
```

```
print(maximo!)
// Imprime "200"
```

Una variable opcional sin asignar ningún valor se inicializa automáticamente a `nil`:

```
var respuestaEncuesta: String?
// respuestaEncuesta es inicializado automáticamente a nil
```

Si se aplica el operador `!` a un valor `nil` se produce un error en tiempo de ejecución y la aplicación se rompe:

```
var respuestaEncuesta: String?
print(respuestaEncuesta!)
// Fatal error: Unexpectedly found nil while unwrapping an Optional value
```

## Ligado opcional

Para comprobar si un valor opcional es `nil` podemos usar un `if`. Es obligado hacerlo si desconocemos el valor que nos llega. Por ejemplo, supongamos que la función `leerRespuesta()` lee una respuesta del usuario y devuelve un `String?`. Para usar esta función deberíamos comprobar si el valor devuelto es distinto de `nil`:

```
let respuestaEncuesta = leerRespuesta()
if respuestaEncuesta != nil {
    let respuesta = respuestaEncuesta!
    print("Respuesta: " + respuesta)
}
```

Como es muy habitual hacer lo anterior, en Swift es posible comprobar si un opcional tiene valor y asignar su valor a otra variable al mismo tiempo con una construcción llamada *ligado opcional (optional binding)*:

```
let respuestaEncuesta = leerRespuesta()
if let respuesta = respuestaEncuesta {
    print ("Respuesta: " + respuesta)
}
```

Podemos leer el código anterior de la siguiente forma: "Si el opcional `respuestaEncuesta` contiene un valor, define la constante `respuesta` con el valor contenido en el opcional".

Otro ejemplo, el método `first` de un array devuelve un opcional que contiene `nil` si el array está vacío o el primer elemento del array en el caso en que exista. El siguiente código utiliza un ligado opcional para implementar otra versión de la función `sumaValores`:

```

func sumaValores(_ valores: [Int]) -> Int {
    if let primero = valores.first {
        let resto = Array(valores.dropFirst())
        return primero + sumaValores(resto)
    } else {
        return 0
    }
}

print(sumaValores([1,2,3,4,5,6,7,8]))
// Imprime "36"

```

Si tenemos varios opcionales es posible comprobar que todos ellos son distintos de `nil` usando varios `let` en el mismo `if`:

```

var x1: Int? = pedirNumUsuario()
var x2: Int? = pedirNumUsuario()
var x3: Int? = pedirNumUsuario()
if let dato1 = x1, let dato2 = x2, let dato3 = x3 {
    let suma = dato1+dato2+dato3
    print("Ningún nil y la suma de todos los datos es: \(suma)")
} else {
    print("Algún dato del usuario es nil")
}

```

## Operador *nil-coalescing*

El operador *nil-coalescing* (`??`) permite definir un valor por defecto en una asignación si un opcional es `nil`.

```

let a: Int? = nil
let b: Int? = 10
let x = a ?? -1
let y = b ?? -1
print("Resultado: \(x), \(y)")
// Imprime Resultado: -1, 10

```

En el ejemplo anterior, en la variable `x` se guardará el valor `-1` y en la variable `y` el valor `10`.

## Encadenamiento de opcionales

El encadenamiento de opcionales (*optional chaining*) permite llamar a un método de una variable que contiene un opcional. Si la variable no es `nil`, se ejecuta el método y se devuelve su valor como un opcional. Si la variable es `nil` se devuelve `nil`.

```

let nombre1: String? = "Pedro"
let nombre2: String? = nil

```

```
// Error: let str1 = nombre1.lowercased()
// No podemos llamar al método lowercased() del String
// porque nombre es opcional y puede tener nil

let str1 = nombre1?.lowercased()
let str2 = nombre2?.lowercased()
// str1: String? = "pedro"
// str2: String? = nil
```

## Definición de `Lista` con opcionales

Veamos como último ejemplo una segunda versión del enum `Lista`, en el que utilizamos un único `case`, pero dando la posibilidad de que el resto de la lista sea `nil` haciéndolo opcional.

Definimos el enumerado y también la función `suma(lista:)`:

```
indirect enum Lista{
    case nodo(Int, Lista?)
}

func suma(lista: Lista) -> Int {
    switch lista {
        case let .nodo(car, cdr):
            if (cdr == nil) {
                return car
            } else {
                return car + suma(lista: cdr!)
            }
    }
}

let z: Lista = .nodo(20, .nodo(10, nil))
print(suma(lista: z))
/// Devuelve 30
```

## Bibliografía

- Swift Language Guide
  - [The Basics](#)
  - [Collection Types](#)
  - [Functions](#)
  - [Closures](#)
  - [Enumerations](#)
  - [Generics](#)
- [Biblioteca estándar de Swift](#)

Lenguajes y Paradigmas de Programación, curso 2019–20

© Departamento Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante

Domingo Gallardo, Cristina Pomares, Antonio Botía, Francisco Martínez

## Tema 6: Programación Orientada a Objetos con Swift

### Más conceptos sobre clausuras

#### Repaso de clausuras

Ya vimos en el tema anterior que las clausuras en Swift son bloques de código con funcionalidades que se pueden pasar de un sitio a otro en nuestro programa. Son muy similares a las expresiones lambda de Scheme o de Java 8 o a los bloques de Objective-C. Una clausura puede capturar o almacenar referencias a cualquier constante o variable del contexto o ámbito dentro de la cual han sido definidas.

Se pueden definir clausuras en Swift de las siguientes formas:

- Funciones anidadas que pueden capturar valores de la función englobante:

```
func construyeFunc() -> () -> Int {  
    var x = 0  
    func funcion() -> Int {  
        x = x + 1  
        return x  
    }  
    return funcion  
}  
  
let f = construyeFunc()  
f() // -> 1  
f() // -> 2
```

La función devuelta por `construyeFunc()` también se puede formular como una expresión de clausura:

```
func construyeFunc() -> () -> Int {  
    var x = 0  
    return {  
        x = x + 1  
        return x  
    }  
}
```

En el siguiente ejemplo vemos claramente que la función captura las variables definidas en el ámbito en el que se creó. En el caso anterior la clausura devuelta captura la variable `x` con el valor 0.

```
func usaFunc(_ f: () -> Int) -> Int {
    var x = 4
    return f()
}

let f = construyeFunc()
usaFunc(f) // -> 1
```

El valor devuelto por `usaFunc` en el resultado de la invocación a `f()` en su segunda línea de código. Y el valor de `x` usado por la clausura `f` es el capturado en el momento de su creación (el valor `x=0` definido en la primera línea de `construyeFunc`).

- Funciones anónimas que pueden capturar valores del contexto que las rodean:

```
var x = 100
usaFunc {return x + 10} // -> 110
```

En el ejemplo anterior la expresión de clausura se define en el ámbito global donde la `x` toma el valor 100. Ese valor queda capturado y es el que se usa en la evaluación de la función. Por tanto, en el contexto donde se evalúa esa clausura (dentro del cuerpo de la función `usaFunc`), la `x` que toma es la capturada y no la definida en el cuerpo de `usaFunc`.

## Clausuras escapadas

Se dice que una clausura escapa de una función cuando la función recibe una función o clausura como parámetro que se llama después de que la función finalice su ejecución.

Por ejemplo, definimos un array global de tipo función:

```
var array : [() -> Int] = []
```

Definimos una función que recibe una clausura como parámetro, pero no se llama dentro de la función que la recibe, sólo la almacena en el array. Al no ser llamada dentro de la función, hay que ponerle el atributo `@escaping` delante del tipo del parámetro para que Swift permita que sea llamada posteriormente fuera de la función que la recibe. Si no lo ponemos, el compilador da un error:

```
func usaClausura(_ f: @escaping () -> Int) -> Void {
    array.append(f)
}

usaClausura {return 4}
```

Se llama posteriormente, fuera de la función:

```
array[0]() //-> 4
```

## Autoclausuras

Una autoclausura permite crear automáticamente una función en tiempo de ejecución sin necesidad de utilizar la sintaxis de las expresiones de clausura.

Se puede ver la autoclausura como una forma de retardar la evaluación, porque esa expresión no se evaluará hasta que se llame a la clausura. Retardar la evaluación es útil con trozos de código que provocan efectos laterales o son costosos computacionalmente, ya que nos permiten controlar el momento de su evaluación.

Por ejemplo, podemos modificar la función anterior `usaFunc` añadiendo la anotación `@autoclosure` antes del tipo del parámetro clausura:

```
func usaFunc(_ f: @autoclosure () -> Int) -> Int {  
    var x = 4  
    return f()  
}
```

Al declararlo de esta forma, se puede llamar a `usaFunc` escribiendo como parámetro una sentencia que el compilador convierte a clausura:

```
var x = 100  
usaFunc(x + 10)
```

La expresión `x + 10` se considera *retardada*: se construye una clausura con ella y su evaluación se realiza dentro de `usaFunc`.

Otro ejemplo en el que podemos comprobar el orden de evaluación:

```
func printTest2(_ result: @autoclosure () -> Void) {  
    print("Antes")  
    result()  
    print("Después")  
}  
  
printTest2(print("Hola"))  
  
// Imprime:  
//     Antes  
//     Hola  
//     Después
```

La sentencia `print("Hola")` se pasa como una autoclausura, por lo que el compilador crea automáticamente una clausura con ella sin tener nosotros que escribir la expresión de clausura.

## Introducción, historia y características de la Programación Orientada a Objetos

### Nacimiento

- La Programación Orientada a Objetos es un paradigma de programación que explota en los 80 pero nace a partir de ideas a finales de los 60 y 70
- Primer lenguaje con las ideas fundamentales de POO: Simula
- Smalltalk (1980) como lenguaje paradigmática de POO
- **Alan Kay** es el creador del término “Object-Oriented” y una de las figuras fundamentales de la historia de la informática moderna. Trabajó en Xerox Park y desarrolló allí ideas que han sido clave para la informática personal (como el Dynabook, precursor de tablets y dispositivos móviles y el lenguajes de programación Smalltalk)
- Artículo de Alan Kay: [“The Early History of Smalltalk”](#), ACM SIGPLAN, March 1993

### Alan Kay

“I invented the term Object-Oriented and I can tell you I did not have C++ in mind.”

“Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term objects for this topic because it gets many people to focus on the lesser idea. The big idea is messaging.”

“Smalltalk's design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.”

### ¿Interesados en Smalltalk?

Visitar:

- <http://www.squeak.org/>
- [http://swiki.agro.uba.ar/small\\_land](http://swiki.agro.uba.ar/small_land)
- <http://www.squeakland.org>

### Lenguajes OO

- Smalltalk, Java, Scala, Ruby, Python, C#, C++, Swift, ...

### Del paradigma imperativo al OO

- Programación procedural: estado abstracto (tipos de datos y barrera de abstracción) + funciones
- Siguiente paso: agrupar estado y funciones en una única entidad
- Los objetos son estas entidades

### Características de la POO

- Objetos (creados/instanciados en tiempo de ejecución) y clases (plantillas estáticas/tiempo de compilación)
- Los objetos agrupan estado y conducta (métodos)
- Los métodos se invocan mediante mensajes
- *Dispatch dinámico*: cuando una operación es invocada sobre un objeto, el propio objeto determina qué código se ejecuta. Dos objetos con la misma interfaz pueden tener implementaciones distintas.
- Herencia: las clases se pueden definir utilizando otras clases como plantillas y modificando sus métodos y/o variables de instancia.

## Clases y objetos

Objeto:

- Un objeto contiene un estado (propiedades, atributos o variables de instancia) y un conjunto de funciones (métodos) que se ejecutan en el ámbito del objeto e implementan las funcionalidades soportadas
- Al ejecutar un método, el objeto modifica su estado
- Pedimos a un objeto que ejecute un método

Clase:

- Una clase es la plantilla que sirve para definir los objetos
- En una clase se define los elementos que componen el objeto (sus atributos o campos) y sus métodos
- En algunos lenguajes se pueden definir también en las clases variables (variables de clase) compartidas por todos los objetos de esa clase

## Lenguajes POO dinámicos vs. estáticos

Dos tendencias:

- Lenguajes **dinámicos**: muchas características del programa se obtienen en tiempo de ejecución
  - Mayor flexibilidad y generalidad del código
  - Dispatch dinámico
  - Reflexión (posibilidad de consultar características de la instancia (nombres de métodos, propiedades, etc.) en tiempo de ejecución)
  - Ejemplos: Smalltalk, Ruby, Python, JavaScript, Java (en menor medida)
- Lenguajes **estáticos**: la mayoría de características del programa se obtienen en tiempo de compilación
  - Mayor eficiencia
  - Se conoce a priori el tipo de la mayor parte de instancias del programa
  - Fuertemente tipado
  - Ejemplos: C++, Swift

Vamos a detallar a continuación las **características de Programación Orientada a Objetos de Swift**. Para tener una introducción rápida puedes empezar leyendo los últimos apartados del [seminario de Swift](#) (los apartados **Objetos, clases y estructuras, Protocolos y extensiones y Genéricos**).

## Clases y estructuras

En el caso de Swift, las clases y las estructuras son muchas más cercanas en funcionalidad que en otros lenguajes, como C o C++, y tienen muchas características comunes. Muchas características de las instancias de una clase se pueden aplicar también a las instancias de una estructura. Por eso en Swift se suele hablar de *instancias* (un término más general) en lugar de *objetos*.

Las clases y las estructuras en Swift tienen muchas cosas en común. Ambos pueden:

- Definir propiedades y almacenar valores
- Definir métodos para proporcionar funcionalidad
- Definir subíndices para proporcionar acceso a sus valores usando una sintaxis de subíndice
- Definir inicializadores para configurar el estado inicial
- Ser extendidas para expandir su funcionalidad más allá de una implementación por defecto
- Ajustarse a un protocolo

Las clases tienen características adicionales que no tienen las estructuras:

- Mediante la herencia una clase puede heredar las características de otra
- El casting de tipos permite comprobar e interpretar el tipo de una instancia de una clase en tiempo de ejecución
- Los deinicializadores permiten a una instancia de una clase liberar los recursos que ha asignado
- Mediante el conteo de referencias se permite que exista más de una referencia a una instancia de una clase

## Definición

```
class UnaClase {  
    // definición de clase  
}  
struct UnaEstructura {  
    // definición de una estructura  
}
```

```
struct CoordsPantalla {  
    var posX = 0  
    var posY = 0  
}  
  
class Ventana {  
    var esquina = CoordsPantalla()  
    var altura = 0  
    var anchura = 0  
    var visible = true  
    var etiqueta: String?  
}
```

El ejemplo define una nueva estructura llamada `CoordsPantalla`, que describe una coordenada de pantalla con posiciones basadas en píxeles. La estructura tiene dos propiedades almacenadas llamadas `posX` y `posY`.

Las propiedades son constantes o variables que se almacenan en la instancia de la clase o de la estructura. El compilador infiere que estas dos propiedades son `Int` al inicializarlas a los valores iniciales de 0.

El ejemplo también define una nueva clase llamada `Ventana` que describe una ventana en una pantalla. Esta clase tiene cinco propiedades variables. La primera, `esquina`, se inicializa con una instancia nueva de una estructura `CoordsPantalla` y se infiere que es de tipo `CoordsPantalla`. Representa la posición superior izquierda de la pantalla. Las propiedades `altura` y `anchura` representan el número de píxeles de las dimensiones de la pantalla. Se inicializan a 0. La propiedad `visible` es un `Bool` que indica si la ventana es visible en pantalla. Por ejemplo, una ventana que esté minimizada no será visible. Por último, `etiqueta` representa el nombre que aparece en la parte superior de la ventana. Es un `String` opcional que se inicializa a `nil` porque no se le asigna un valor inicial.

## Instancias de clases y estructuras

La definición de las estructuras y las clases únicamente definen sus aspectos generales. Para describir una configuración específica (una resolución o un modo de vídeo concreto) es necesario crear una instancia de una estructura o una clase. La sintaxis para crear ambas es similar:

```
var unasCoordsPantalla = CoordsPantalla()  
var unaVentana = Ventana()
```

La forma más sencilla de inicialización es la anterior. Se utiliza el nombre del tipo de la clase o estructura seguidos de paréntesis vacíos. Esto crea una nueva instancia de una clase o estructura, con sus propiedades inicializadas a los valores por defecto definidos en la declaración de las propiedades.

Swift proporciona este **inicializador por defecto** para clases y estructuras, siempre que no se defina algún inicializador explícito. Más adelante comentaremos cómo definir estos inicializadores explícitos.

En el caso de la instancia `unasCoordsPantalla` los valores a los que se han inicializado sus propiedades son:

```
unasCoordsPantalla.posX // 0  
unasCoordsPantalla.posY // 0
```

Las propiedades de la instancia `unaVentana` son:

```
unaVentana.esquina // CoordsPantalla con posX = 0 y posY = 0  
unaVentana.altura // 0  
unaVentana.anchura // 0  
unaVentana.visible // true  
unaVentana.etiqueta // nil
```

Todas las propiedades de una instancia deben estar definidas después de haberse inicializado, a no ser que la propiedad sea un opcional.

## Acceso a propiedades

Se puede acceder y modificar las propiedades usando la *sintaxis de punto*:

```
// Accedemos a la propiedad
coords.posX // Devuelve 0
// Actualizamos la propiedad
coords.posX = 100
ventana.esquina.posY = 100
```

## Inicialización de las estructuras por sus propiedades

Si en las estructuras no se definen inicializadores explícitos (veremos más adelante cómo hacerlo) podemos utilizar un **inicializador memberwise** (de todas las propiedades) en el que hay que proporcionar los valores de todas sus propiedades.

En las clases no existen los inicializadores *memberwise*, sólo en las estructuras.

```
var coords = CoordsPantalla(posX: 200, posY: 400)
```

Es necesario inicializar todas las propiedades. Si no, el compilador da un error:

```
var coords2 = CoordsPantalla(posX: 200)
// error
```

## Estructuras y enumeraciones son tipos valor

Un *tipo valor* es un tipo cuyo valor se copia cuando se asigna a una variable o constante, o cuando se pasa a una función.

Todos los tipos básicos de Swift -enteros, números en punto flotante, cadenas, arrays y diccionarios- son tipos valor y se implementan como estructuras. Las estructuras y las enumeraciones son tipos valor en Swift.

```
var coords1 = CoordsPantalla(posX: 600, posY: 600)
var coords2 = coords1
coords2 posX = 1000
coords1 posX // devuelve 600
```

En el ejemplo se declara una constante llamada `coords1` y se asigna a una instancia de `CoordsPantalla` inicializada con la posición x de 600 y la posición y de 600. Después se declara una variable llamada `coords2` y se asigna al valor actual de `coords1`. Debido a que `CoordsPantalla` es una estructura, se crea *una copia* de la instancia existente y esta nueva copia se asigna a `coords2`. Aunque ahora `coords2` y `coords1` tienen las mismas `posX` y `posY`, son dos instancias completamente distintas. Después, la propiedad `posX` de `coords2` se actualiza a 1000.

Podemos comprobar que la propiedad se modifica, pero que el valor de `posX` en `coords1` sigue siendo el mismo.

## Las clases son tipos referencia

A diferencia de los tipos valor, los tipos de referencias no se copian cuando se asignan o se pasan a funciones. En su lugar se usa una referencia a la misma instancia existente.

En Swift las clases son tipos referencias. Veamos, por ejemplo, una instancia de la clase `Ventana`:

```
var ventana1 = Ventana()
ventana1.esquina = coords1
ventana1.altura = 800
ventana1.anchura = 800
ventana1.etiqueta = "Finder"
var ventana2 = ventana1
ventana2.anchura = 1000
ventana1.anchura // devuelve 1000
```

Declaramos una variable llamada `ventana1` inicializada con una instancia nueva de la clase `Ventana`. Le asignamos a la propiedad `esquina` una copia de la resolución anterior `coords1`. Después declaramos la altura, anchura y etiqueta de la ventana. Y, por último, `ventana1` se asigna a una nueva constante llamada `ventana2`, y la anchura se modifica.

Debido a que son tipos de referencia, `ventana1` y `ventana2` se refieren a la misma instancia de `Ventana`. Son sólo dos nombres distintos para la misma única instancia.

Lo podemos comprobar modificando una propiedad mediante una variable y viendo que esa misma propiedad en la otra variable se ha modificado también (líneas 7 y 8).

Si tienes experiencia con C, C++, o Objective-C, puedes saber que estos lenguajes usan punteros para referirse a una dirección de memoria. Una constante o variable en Swift que se refiere a una instancia de un tipo referencia es similar a un puntero en C, pero no es un puntero que apunta a una dirección de memoria y no requiere que se escriba un asterisco (\*) para indicar que estas creando una referencia. En su lugar, estas referencias se definen como cualquier otra constante o variable en Swift.

## Declaración de instancias con `let`

Las estructuras y clases también tienen comportamientos distintos cuando se declaran las variables con `let`.

Si definimos con `let` una instancia de una estructura estamos declarando constante la variable y todas las propiedades de la instancia. No podremos modificar ninguna:

```
let coords3 = CoordsPantalla(posX: 400, posY: 400)
coords3 posX = 800
// error: cannot assign to property: 'coords3' is a 'let' constant
```

Si definimos con un `let` una instancia de una clase sólo estamos declarando constante la variable. No podremos reasignarla, pero sí que podremos modificar las propiedades de la instancia referenciada por la variable:

```
let ventana3 = Ventana()
// Sí que podemos modificar una propiedad de la instancia:
ventana3.etiqueta = "Listado"
// Pero no podemos reasignar la variable:
ventana3 = ventana1
// error: cannot assign to value: 'ventana3' is a 'let' constant
```

## Operadores de identidad

A veces puede ser útil descubrir si dos constantes o variables se refieren exactamente a la misma instancia de una clase. Para permitir esto, Swift proporciona dos operadores de identidad:

- Idéntico a (`==`)
- No idéntico a (`!=`)

```
ventana1 === ventana2 // devuelve true
ventana1 === ventana3 // devuelve false
```

Estos operadores "idéntico a" no son los mismos que los de "igual a" (representado por dos signos iguales `==`):

- "Idéntico a" significa que dos constantes o variables de una clase se refieren exactamente a la misma instancia de la clase.
- "Igual a" significa que dos instancias se consideran "iguales" o "equivalentes" en su valor. Es responsabilidad del diseñador de la clase definir la implementación de estos operadores.

## Criterios para usar estructuras y clases

Podemos usar tanto clases como estructuras para definir nuestros tipos de datos y utilizarlos como bloques de construcción del código de nuestros programas. Sin embargo, se utilizan para distintos tipos de tareas.

Como regla general, utilizaremos una estructura cuando se cumplen una o más de las siguientes condiciones:

- El principal objetivo de la estructura es encapsular unos pocos datos relativamente sencillos.
- Es razonable esperar que los valores encapsulados serán copiados, más que referenciados, cuando asignamos o pasamos una instancia de esa estructura.
- Todas las propiedades almacenadas en la estructura son a su vez tipos valor, que también se espera que sean copiados más que referenciados.
- La estructura no necesita heredar propiedades o conducta de otro tipo existente.

Ejemplos de buenos candidatos de estructuras incluyen:

- El tamaño de una forma geométrica, encapsulando por ejemplo las propiedades `ancho` y `alto` de tipo `Double`.
- Una forma de referirse a rangos dentro de una serie, encapsulando por ejemplo, una propiedad `comienzo` y otra `longitud`, ambos del tipo `Int`.
- Un punto en un sistema de coordenadas 3D, encapsulando quizás las propiedades `x`, `y` y `z`, todos ellos de tipo `Double`.

En el resto de casos, definiremos una clase y crearemos instancias de esa clase que tendrán que ser gestionadas y pasadas por referencia. En la práctica, esto representa que la mayoría de datos que construiremos en nuestros programas deberían clases, no estructuras. Aunque usaremos muchas de las estructuras estándar de Swift.

## Propiedades

Las *propiedades* asocian valores con una clase, estructura o enumeración particular. Las propiedades almacenadas (*stored properties*) almacenan valores constantes y variables como parte de una instancia, mientras que las propiedades calculadas (*computed properties*) calculan (en lugar de almacenar) un valor. Las propiedades calculadas se definen en clases, estructuras y enumeraciones. Las propiedades almacenadas se definen sólo en clases y estructuras.

- Enumeraciones: pueden contener sólo propiedades calculadas.
- Clases y estructuras: pueden contener propiedades almacenadas y calculadas.

Las propiedades calculadas y almacenadas se asocian habitualmente con instancias de un tipo particular. Sin embargo, las propiedades también pueden asociarse con el propio tipo. Estas propiedades se conocen como propiedades del tipo (*type properties*).

Además, en Swift es posible definir observadores de propiedades que monitoricen cambios en los valores de una propiedad, a los que podemos responder con acciones programadas. Los observadores de propiedades pueden añadirse tanto a propiedades almacenadas definidas por nosotros como a propiedades heredadas de la superclase.

### Propiedades almacenadas

En su forma más simple, una propiedad almacenada es una constante o variable que está almacenada como parte de una instancia de una clase o estructura particular. Las propiedades almacenadas pueden ser o bien variables (usando la palabra clave `var`) o bien constantes (usando la palabra clave `let`).

Podemos proporcionar un valor por defecto para la inicialización de las propiedades almacenadas, tanto variables como constantes.

El siguiente ejemplo define una estructura llamada `RangoLongitudFija` que describe un rango de valores enteros cuya longitud no puede ser modificada una vez que se crea:

```
struct RangoLongitudFija {
    var primerValor: Int
    let longitud: Int
}
var rangoDeTresItemss = RangoLongitudFija(primerValor: 0, longitud: 3)
```

```
// el rango representa ahora valores enteros the range represents integer values
0, 1, and 2
rangoDeTresItemss.primerValor = 6
// el rango representa ahora valores enteros 6, 7 y 8
```

Las instancias de **RangoLongitudFija** tienen una propiedad almacenada variable llamada **primerValor** y una propiedad almacenada constante llamada **longitud**. En el ejemplo, **longitud** se inicializa cuando se crea el nuevo rango y no puede ser cambiada en el futuro, por ser una propiedad constante.

## Propiedades calculadas

Además de las propiedades almacenadas, las clases, estructuras y enumeraciones pueden definir *propiedades calculadas*, que no almacenan realmente un valor. En su lugar, proporcionan un *getter* y un opcional *setter* que devuelven y modifican otras propiedades y valores de forma indirecta.

```
struct Punto {
    var x = 0.0, y = 0.0
}
struct Tamaño {
    var ancho = 0.0, alto = 0.0
}
struct Rectangulo {
    var origen = Punto()
    var tamaño = Tamaño()
    var centro: Punto {
        get {
            let centroX = origen.x + (tamaño.ancho / 2)
            let centroY = origen.y + (tamaño.alto / 2)
            return Punto(x: centroX, y: centroY)
        }
        set(centroNuevo) {
            origen.x = centroNuevo.x - (tamaño.ancho / 2)
            origen.y = centroNuevo.y - (tamaño.alto / 2)
        }
    }
}
var cuadrado = Rectangulo(origen: Punto(x: 0.0, y: 0.0),
                           tamaño: Tamaño(ancho: 10.0, alto: 10.0))
let centroCuadradoInicial = cuadrado.centro
cuadrado.centro = Punto(x: 15.0, y: 15.0)
print("cuadrado.origen está ahora en (\(cuadrado.origen.x), \
(cuadrado.origen.y))")
// Prints "cuadrado.origen está ahora en (10.0, 10.0)"
```

Este ejemplo define tres estructuras para trabajar con formas geométricas:

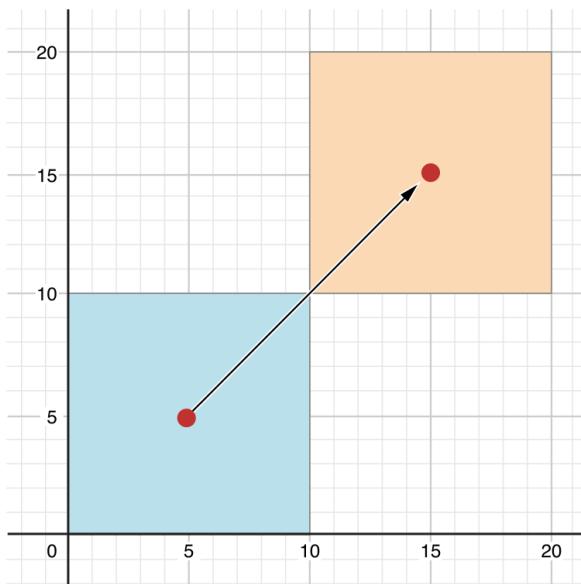
- **Punto** encapsula una coordenada (**x**, **y**)
- **Tamaño** encapsula un ancho y un alto
- **Rectangulo** define una rectángulo por un punto de origen y un tamaño

La estructura `Rectangulo` proporciona una propiedad calculada llamada `centro`. La posición actual del centro de un `Rectangulo` puede ser siempre determinada a partir de su origen y su tamaño, por lo que no necesitamos almacenarlo como un `Punto` explícito. En su lugar, `Rectangulo` define un *getter* y un *setter* programado para una variable calculada llamada `centro`, para permitirnos trabajar con el centro del rectángulo como si fuera una propiedad almacenada.

En el ejemplo se crea una variable `Rectangulo` llamada `cuadrado`. La variable `cuadrado` se inicializa un punto origen de `(0, 0)` y un ancho y tamaño de `10`. Este cuadrado está representado por el cuadrado azul en el diagrama de abajo.

Accedemos entonces a la propiedad `centro` de la variable `cuadrado` usando la sintaxis del punto (`cuadrado.centro`), lo que causa que se llame al *getter* de `centro` para devolver el valor actual de la propiedad. En lugar de devolver los valores existentes, el *getter* calcula realmente y devuelve un nuevo `Punto` para representar el centro del cuadrado. Como puede verse arriba, el *getter* devuelve correctamente un punto con los valores `(5, 5)`.

Después la propiedad `centro` se actualiza al nuevo valor de `(15, 15)` lo que mueve el cuadrado arriba a la derecha, a la nueva posición mostrada por el cuadrado naranja en el diagrama de abajo. Al asignar el nuevo valor a la propiedad se llama al *setter* del centro, lo que modifica los valores `x` e `y` de las propiedades almacenadas originales, y mueve el cuadrado a su nueva posición.



Se puede definir una versión acortada del *setter* usando la variable por defecto `newValue` que contiene el nuevo valor asignado en el *setter*:

```
struct Rectangulo {
    var origen = Punto()
    var tamaño = Tamaño()
    var centro: Punto {
        get {
            let centroX = origen.x + (tamaño.ancho / 2)
            let centroY = origen.y + (tamaño.alto / 2)
            return Punto(x: centroX, y: centroY)
        }
        set {
            // Implementation of the setter logic
        }
    }
}
```

```

        origen.x = newValue.x - (tamaño.ancho / 2)
        origen.y = newValue.y - (tamaño.alto / 2)
    }
}
}

```

## Propiedades solo-lectura

Una propiedad calculada con un *getter* y sin *setter* se conoce como una propiedad calculada de solo-lectura. Una propiedad calculada de solo-lectura siempre devuelve un valor, y puede accederse a ella usando la sintaxis de punto, pero no puede modificarse a un valor distinto.

Es posible simplificar la declaración de una propiedad calculada de solo-lectura eliminando la palabra clave **get** y sus llaves:

```

struct Cuboide {
    var ancho = 0.0, alto = 0.0, profundo = 0.0
    var volumen: Double {
        return ancho * alto * profundo
    }
}
let cuatroPorCincoPorDos = Cuboide(ancho: 4.0, alto: 5.0, profundo: 2.0)
print("el volumen de cuatroPorCincoPorDos es \(cuatroPorCincoPorDos.volumen)")
// Imprime "el volumen de cuatroPorCincoPorDos es 40.0"

```

Este ejemplo define una nueva estructura llamada **Cuboide**, que representa una caja rectangular 3D con propiedades **ancho**, **alto** y **profundo**. Esta estructura tiene una propiedad calculada llamada **volumen**, que calcula y devuelve el volumen actual del cuboide. No tendría sentido que el volumen fuera modificable, porque no sería ambiguo determinar qué valores concretos de ancho, alto y profundo deberían usarse para un valor particular del volumen.

## Observadores de propiedades

Los observadores de propiedades (*property observers*) observan y responden a cambios en el valor de una propiedad. Los observadores de propiedades se llaman cada vez que el valor de una propiedad es actualizado, incluso si el nuevo valor es el mismo que el valor actual de la propiedad.

Se pueden añadir observadores a cualquier **propiedad almacenada** que se definan. Se pueden también añadir observadores a cualquier propiedad heredada (ya sea almacenada o calculada) sobreescribiendo la propiedad en la subclase. No es necesario definir observadores de propiedades calculadas no sobreescritas porque siempre es posible observar y responder a cambios en su valor en el *setter* de la propiedad.

Es posible definir alguno o ambos de estos observadores sobre una propiedad:

- **willSet** es llamado justo antes de que el nuevo valor se almacena en la propiedad.
- **didSet** es llamado inmediatamente después de que el nuevo valor es almacenado en la propiedad.

Si implementamos un observador `willSet`, se le pasa el nuevo valor de la propiedad como un parámetro constante. Podemos especificar un nombre para este parámetro como parte de la implementación de `willSet`. Si no escribimos el nombre del parámetro y los paréntesis dentro de la implementación, el parámetro estará disponible con el nombre por defecto de `newValue`.

De forma similar, si implementamos un observador `didSet`, se pasa como un parámetro constante que contiene el valor antiguo de la propiedad. Podemos darle nombre al parámetro o usar el nombre por defecto de `oldValue`. Si asignamos un valor a la propiedad dentro de su propio observador `didSet`, el nuevo valor que asignamos reemplaza el que acaba de añadirse a la propiedad.

A continuación podemos ver un ejemplo de `willSet` y `didSet` en acción. En él definimos una clase nueva llamada `CuentaPasos`, que hace un seguimiento del número total de pasos que una persona hace al caminar. Esta clase puede usarse con datos de entrada de un *podómetro* o cualquier otro sistema de seguir el ejercicio de la persona durante su rutina diaria.

```
class ContadorPasos {
    var totalPasos: Int = 0 {
        willSet(nuevoTotalPasos) {
            print("A punto de actualizar totalPasos a \(nuevoTotalPasos)")
        }
        didSet {
            if totalPasos > oldValue {
                print("Añadidos \(totalPasos - oldValue) pasos")
            }
        }
    }
    let contadorPasos = ContadorPasos()
    contadorPasos.totalPasos = 200
    // Imprime: "A punto de actualizar totalPasos a 200"
    // Imprime: "Añadidos 200 pasos"
    contadorPasos.totalPasos = 360
    // Imprime: "A punto de actualizar totalPasos a 360"
    // Imprime: "Añadidos 160 pasos"
    contadorPasos.totalPasos = 896
    // Imprime: "A punto de actualizar totalPasos a 896"
    // Imprime: "Añadidos 536 pasos"
```

La clase `CuentaPasos` declara la propiedad `totalPasos` de tipo `Int`. Esta es una propiedad almacenada con observadores `willSet` y `didSet`.

Los observadores `willSet` y `didSet` de `totalPasos` se llaman siempre que se le asigna un nuevo valor a la propiedad. Esto es así incluso si el nuevo valor es el mismo que el valor actual.

El observador `willSet` usa un parámetro definido por nosotros con el nombre de `nuevoTotalPasos` para el valor que llega. En el ejemplo, sencillamente imprime el valor que está a punto de establecer.

El observador `didSet` se llama después de que el valor de `totalPasos` se ha actualizado. Compara el nuevo valor de `totalPasos` con el valor antiguo. Si el número total de pasos se ha incrementado, se imprime un

mensaje indicando cuántos pasos se han tomado. El observador `didSet` no proporciona un parámetro definido por nosotros para el valor antiguo, sino que usa el nombre por defecto `oldValue`.

## Variables locales y globales

Las capacidades anteriores de propiedades calculadas y de observadores también están disponibles para variables globales y locales.

El siguiente ejemplo muestra un ejemplo con una variable calculada a partir de otras dos:

```
var x = 10 {
    didSet {
        print("El nuevo valor: \(x) y el valor antiguo: \(oldValue)")
    }
}
var y = 2 * x
var z : Int {
    get {
        return x + y
    }
    set {
        x = newValue / 2
        y = newValue / 2
    }
}

print(z)
z = 100
print(x)
```

## Propiedades del tipo

Las propiedades de las instancias son propiedades que pertenecen a una instancia de un tipo particular. Cada vez que creamos una nueva instancia de ese tipo, tiene su propio conjunto de valores de propiedades, separados de los de cualquier otra instancia.

Podemos definir también propiedades que pertenecen al tipo propiamente dicho, no a ninguna de las instancias de ese tipo. Sólo habrá una copia de estas propiedades, sea cual sea el número de instancias de ese tipo que creemos. Estos tipos de propiedades se llaman propiedades del tipo (*type properties*). Se pueden definir en tanto en estructuras, clases como en enumeraciones.

Las propiedades del tipo son útiles para definir valores que son universales a todas las instancias de un tipo particular, como una propiedad constante que todas las instancias pueden usar (como una constante estática en C), o una propiedad variable que almacena un valor que es global a todas las instancias de ese tipo (como una variable estática en C).

Las propiedades del tipo almacenadas pueden ser variables o constantes. Las propiedades del tipo calculadas se declaran siempre como propiedades variables, de la misma forma que las propiedades calculadas de instancias.

A diferencia de las propiedades almacenadas de instancias, debemos siempre proporcionar un valor por defecto para las propiedades almacenadas de tipo. Esto es debido a que el tipo por si mismo no tiene un inicializador que pueda asignar un valor en tiempo de inicialización.

En Swift, las propiedades del tipo se definen como parte de la definición del tipo, dentro de las llaves del tipo. Las propiedades del tipo toman valor en el ámbito del tipo. Para definir una propiedad del tipo hay que usar la palabra clave `static`. Para propiedades de tipo calculadas de clases, podemos usar en su lugar la palabra clave `class` para permitir a las subclases que sobreescrbían la implementación de la superclase.

Las propiedades del tipo pueden ser también constantes (`let`) o variables (`var`).

Ejemplo:

```
struct UnaEstructura {
    static var almacenada = "A"
    static var calculada : Int {
        return 1
    }
}
enum UnaEnumeracion {
    static var almacenada = "A"
    static var calculada: Int {
        return 1
    }
}
class UnaClase {
    static var almacenada = "A"
    static var calculada: Int {
        return 1
    }
}
```

Las propiedades del tipo se consultan y actualizan usando también la sintaxis de punto, pero sobre *el tipo*:

```
UnaEstructura.almacenada // devuelve "A"
UnaEstructura.almacenada = "B"
UnaClase.calculada // devuelve 1
```

- No es posible acceder a la variable del tipo a través de una instancia:

```
let a = UnaEstructura()
a.almacenada // error
```

El siguiente ejemplo muestra cómo es posible usar una variable del tipo para almacenar información global a todas las instancias de ese tipo:

```

struct Valor {
    var valor: Int = 0 {
        didSet {
            Valor.sumaValores += valor
        }
    }
    static var sumaValores = 0
}

var c1 = Valor()
var c2 = Valor()
var c3 = Valor()
c1.valor = 10
c2.valor = 20
c3.valor = 30
print("Suma de los cambios de valores: \(Valor.sumaValores)")
// Imprime 60

```

## Métodos

Los *métodos* son funciones que están asociadas a un tipo particular. Las clases, estructuras y enumeraciones pueden definir todas ellas métodos de instancia, que encapsulan tareas y funcionalidades específicas que trabajan con una instancia de un tipo dado. Las clases, estructuras y enumeraciones también pueden definir métodos del tipo, que están asociados con el propio tipo. Los métodos del tipo son similares a los métodos de clase en Java.

El hecho de que las estructuras y las enumeraciones puedan definir métodos en Swift es una diferencia importante con C y Objective-C.

### Métodos de instancia

Los métodos de instancia son funciones que pertenecen a instancias de una clase, estructura o enumeración. Proporcionan la funcionalidad de esas instancias, bien proporcionando formas de acceder y modificar propiedades de las instancias, o bien proporcionando funcionalidades relacionadas con el propósito de la instancia. Los métodos de instancia tienen exactamente la misma sintaxis que las funciones.

Los métodos de instancia se escriben dentro de las llaves del tipo al que pertenecen. Un método de instancia tiene acceso implícito a todos los otros métodos de instancia y propiedades del tipo. Un método de instancia puede ser invocado sólo sobre una instancia específica del tipo al que pertenece. No puede ser invocado de forma aislada sin una instancia existente.

A continuación podemos ver un ejemplo que define una sencilla clase **Contador**, que puede usarse para contar el número de veces que sucede una acción:

```

class Contador {
    var veces = 0
    func incrementa() {
        veces += 1
    }
}

```

```

func incrementa(en cantidad: Int) {
    veces += cantidad
}
func reset() {
    veces = 0
}
}

```

Y un ejemplo de uso:

```

let contador = Contador()
// el valor inicial del contador es 0
contador.incrementa()
// el valor del contador es ahora 1
contador.incrementa(en: 5)
// el valor del contador es ahora 6
contador.reset()
// el valor del contador es ahora 0

```

En el ejemplo anterior, los métodos no devuelven ningún valor. Podemos modificar el ejemplo para que los métodos devuelvan el valor actualizado del contador:

```

class Contador {
    var veces = 0
    func incrementa() -> Int {
        veces += 1
        return veces
    }
    func incrementa(en cantidad: Int) -> Int {
        veces += cantidad
        return veces
    }
    func reset() -> Int {
        veces = 0
        return veces
    }
}

```

## Nombres locales y externos de parámetros

Ya vimos que los parámetros de las funciones pueden tener un nombre interno y un nombre externo. Lo mismo sucede con los métodos, porque los métodos no son más que funciones asociadas con un tipo.

Los nombres de los métodos en Swift se refieren normalmente al primer parámetro usando una preposición como **con**, **en**, **a** o **por**, como hemos visto en el ejemplo anterior `incrementa(en:)`. El uso de la preposición permite que el método se lea como una frase.

El nombre de una parámetro se utiliza también como etiqueta del argumento (nombre externo). Al igual que en las funciones, es posible definir dos nombres del parámetro, uno externo y otro interno. Y el nombre externo puede ser un \_ para indicar que no es necesario usar la etiqueta del argumento.

Esta forma de invocar a los métodos hace que el lenguaje sea más expresivo, sin necesidad de nombres largos de métodos o funciones.

Consideremos por ejemplo esta versión alternativa de la clase **Contador**, que define una forma más compleja del método **incrementa(en:)**:

```
class Contador {
    var valor: Int = 0
    func incrementa(en cantidad: Int, numeroDeVeces: Int) {
        valor += cantidad * numeroDeVeces
    }
}
```

El método **incrementa(en:numeroDeVeces:)** tiene dos parámetros: **cantidad** y **numeroDeVeces**. El primer parámetro tiene un nombre externo y otro interno. En el cuerpo del método se utiliza el nombre interno (**cantidad**). El segundo parámetro **numeroDeVeces** es tanto nombre externo como interno. Podemos llamar al método de la siguiente forma:

```
let contador = Contador()
contador.incrementa(en: 5, numeroDeVeces: 3)
// el valor del contador es ahora 15
```

Al igual que en las funciones, podemos definir explícitamente los nombres externos de los parámetros y usar el subrayado (\_) para indicar que ese parámetro no tendrá nombre externo.

## La propiedad **self**

Toda instancia de un tipo tiene una propiedad implícita llamada **self**, que es exactamente equivalente a la instancia misma. Podemos usar la propiedad **self** para referirnos a la instancia actual dentro de sus propios métodos de instancia.

El método **incrementa()** en el ejemplo anterior podría haberse escrito de esta forma:

```
func incrementa() {
    self.veces += 1
}
```

En la práctica no es necesario usar **self** casi nunca. Swift asume que cualquier referencia a una propiedad dentro de un método se refiere a la propiedad de la instancia. Es obligado usarlo es cuando el nombre de la propiedad coincide con el nombre de un parámetro. En esta situación el nombre del parámetro toma precedencia y es necesario usar **self** para poder referirse a la propiedad de la instancia.

Un ejemplo:

```
struct Punto {
    var x = 0.0, y = 0.0
    func estaAlaDerecha(de x: Double) -> Bool {
        return self.x > x
    }
}
let unPunto = Punto(x: 4.0, y: 5.0)
if unPunto.estaAlaDerecha(de: 1.0) {
    print("Este punto está a la derecha de la línea donde x == 1.0")
}
// Imprime "Este punto está a la derecha de la línea donde x == 1.0"
```

## Operaciones con instancias de tipo valor

Las estructuras y las enumeraciones son **tipos valor**. Por defecto, las propiedades de un tipo valor no pueden ser modificadas desde dentro de los métodos de instancia.

Si queremos modificar una propiedad de un tipo valor la forma más natural de hacerlo es creando una instancia nueva, usando el estilo de programación funcional:

```
struct Punto {
    var x = 0.0, y = 0.0
    func incrementa(incX: Double, incY: Double) -> Punto {
        return Punto(x: x+incX, y: y+incY)
    }
}
let unPunto = Punto(x: 1.0, y: 1.0)
var puntoMovido = unPunto.incrementa(incX: 2.0, incY: 3.0)
print("Hemos movido el punto a (\(puntoMovido.x), \(puntoMovido.y))")
// Imprime "Hemos movido el punto a (3.0, 4.0)"
```

## Modificación de tipos valor desde dentro de la instancia

Sin embargo, hay ocasiones en las que necesitamos modificar las propiedades de nuestra estructura o enumeración dentro de un método particular.

Necesitamos conseguir una conducta *mutadora* para ese método. El método puede mutar (esto es, cambiar) sus propiedades desde dentro del método, así como asignar una instancia completamente nueva a su propiedad implícita `self`, con lo que esta nueva instancia reemplazará la existente cuando el método termine.

Podemos conseguir esta conducta colocando la palabra clave `mutating` antes de la palabra `func` del método:

```
struct Punto {
    var x = 0.0, y = 0.0
    mutating func incrementado(incX: Double, incY: Double) {
```

```

        x += incX
        y += incY
    }
}

var unPunto = Punto(x: 1.0, y: 1.0)
unPunto.incrementado(incX: 2.0, incY: 3.0)
print("El punto está ahora en (\(unPunto.x), \(unPunto.y))")
// Imprime "El punto está ahora en (3.0, 4.0)"

```

La estructura `Punto` anterior define un método mutador `incrementado(incX:incY:)` que mueve una instancia de `Punto` una cierta cantidad. En lugar de devolver un nuevo punto, el método modifica realmente el punto en el que es llamado. La palabra clave `mutating` se añade a su definición para permitirle modificar sus propiedades.

Hay que hacer notar que no es posible llamar a un método mutador sobre una constante de un tipo estructura, porque sus propiedades no se pueden cambiar, incluso aunque sean propiedades variables:

```

let puntoFijo = Punto(x: 3.0, y: 3.0)
puntoFijo.incrementado(incX: 2.0, incY: 3.0)
// esto provocará un error

```

## Asignación a `self` en un método mutador

Los métodos mutadores pueden asignar una nueva instancia completamente nueva a la propiedad `self`. El anterior ejemplo `Punto` podría haber sido escrito de la siguiente forma:

```

struct Punto {
    var x = 0.0, y = 0.0
    mutating func incrementa(incX: Double, incY: Double) {
        self = Punto(x: x + incX, y: y + incY)
    }
}

```

Esta versión del método mutador `incrementa(incX:incY:)` crea una estructura nueva cuyos valores `x` e `y` se inicializan a los valores deseados. El resultado final de llamar a esta versión alternativa será exactamente el mismo que llamar a la versión anterior (aunque con una pequeña penalización de eficiencia: este método es 1,3 veces más lento que el anterior en la versión 2.2 del compilador de Swift).

Los métodos mutadores de enumeraciones pueden establecer el parámetro `self` implícito para que tenga un subtipo distinto de la misma enumeración:

```

enum InterruptorTriEstado {
    case apagado, medio, alto
    mutating func siguiente() {
        switch self {
            case .apagado:

```

```

        self = .medio
    case .medio:
        self = .alto
    case .alto:
        self = .apagado
    }
}
}

var luzHorno = InterruptorTriEstado.medio
luzHorno.siguiente()
// luzHorno es ahora .alto
luzHorno.siguiente()
// luzHorno es ahora .apagado

```

## Métodos del tipo

Los métodos de instancia, como los descritos antes, se llaman en instancias de un tipo particular. Es posible también definir métodos que se llaman en el propio tipo. Esta clase de métodos se denominan *métodos del tipo*. Se define un método del tipo escribiendo la palabra clave `static` antes de la palabra clave `func` del método. Las clases también pueden usar la palabra clave `class` para permitir a las subclases sobreescibir la implementación del método.

Los métodos del tipo se invocan también con la sintaxis de punto, escribiendo el nombre del tipo. Por ejemplo:

```

class NuevaClase {
    static func unMetodoDelTipo() {
        print("Hola desde el tipo")
    }
}
NuevaClase.unMetodoDelTipo()

```

Dentro del cuerpo del método, la propiedad implícita `self` se refiere al propio tipo, más que a una instancia de ese tipo. Para estructuras y enumeraciones, esto significa que puedes usar `self` para desambiguar entre propiedades del tipo y los parámetros del método, de la misma forma que se hace en los métodos de instancias.

Cualquier nombre de método o propiedad que se utilice en el cuerpo de un método del tipo se referirá a otras propiedades o métodos de nivel del tipo. Se puede utilizar estos nombres sin necesidad de añadir el prefijo del nombre del tipo.

Por ejemplo, podemos añadir a la clase `Ventana` una propiedad y método de clase con la que almacenar instancias de ventanas. Inicialmente guardamos un array vacío.

```

class Ventana {

    // Propiedades

```

```

static var ventanas: [Ventana] = []
static func registrar(ventana: Ventana) {
    ventanas.append(ventana)
}
}

```

Cada vez que creamos una ventana podemos llamar al método `registrar` de la clase para añadirlo a la colección de ventanas de la clase:

```

let v1 = Ventana()
Ventana.registrar(ventana: v1)
print("Se han registrado \(Ventana.ventanas.count) ventanas")
// Imprime "Se han registrado 1 ventanas"

```

## Inicialización

*Inicialización* es el proceso de preparar para su uso una instancia de una clase, estructura o enumeración. Este proceso incluye la asignación de un valor inicial para cada propiedad almacenada y la ejecución de cualquier otra operación de inicialización que se necesite para que la nueva instancia esté lista para usarse.

Para implementar este proceso de inicialización hay que definir *inicializadores*, que son como métodos especiales que pueden llamarse para crear una nueva instancia de un tipo particular. A diferencia de otros lenguajes, los inicializadores en Swift no devuelven un valor. Su papel principal es que las nuevas instancias del tipo estén correctamente inicializadas antes de poder ser usadas por primera vez.

También es posible implementar *deinicializadores*, métodos que se ejecutan cuando las instancias son eliminadas de la memoria (no vamos a explicarlos por falta de tiempo).

El proceso de inicialización de una instancia puede resultar un proceso complicado, sobre todo cuando se tienen relaciones de herencia y hay que especificar también cómo realizar la inicialización de la subclase utilizando la superclase. Por falta de tiempo no vamos a explicar todo el proceso completo. Recomendamos consultar la [documentación original de Swift](#).

### Inicializadores por defecto y *memberofwise*

Ya hemos visto que es posible inicializar clases y estructuras definiendo valores por defecto a todas sus propiedades (con la posible excepción de las que tienen un tipo opcional). En ese caso, podemos no definir ningún inicializador y usar el inicializador por defecto que proporciona Swift.

```

struct Punto2D {
    var x = 0.0
    var y = 0.0
}
class Segmento {
    var p1 = Punto2D()
    var p2 = Punto2D()
}

```

```

}

var s = Segmento()

```

También es posible en las estructuras utilizar el inicializador *memberwise*, en el que especificamos todos los valores de las propiedades:

```

var p = Punto2D(x: 10.0, y: 10.0)

```

Los inicializadores por defecto y *memberwise* desaparecen en el momento en que definimos algún inicializador con la palabra `init`. Veamos cómo definir inicializadores.

### Inicialización de propiedades almacenadas

Como hemos dicho, las clases y estructuras deben definir todas sus propiedades almacenadas a un valor inicial en el tiempo en la instancia se crea, a no ser que éstas sean opcionales, en cuyo caso quedarían inicializadas a `nil`.

Podemos definir el valor inicial para una propiedad en un inicializador o asignándole un valor por defecto como parte de la definición de la propiedad.

Un *inicializador*, en su forma más simple se escribe con la palabra clave `init`:

```

init() {
    // realizar alguna inicialización aquí
}

```

Por ejemplo, podemos definir la estructura `Farenheit` que almacena una temperatura en grados Farenheit. Tiene una propiedad almacenada de tipo `Double`. Definimos un inicializador que inicializa las instancias a 32.0 (equivalente a 0.0 grados Celsius).

```

struct Farenheit {
    var temperatura: Double
    init() {
        temperatura = 32.0
    }
}
var f = Farenheit()
print("La temperatura por defecto es \(f.temperatura) Farenheit")
// Imprime "La temperatura por defecto es 32.0° Farenheit"

```

La implementación anterior es equivalente a la que ya hemos visto con el inicializador por defecto:

```
struct Fahrenheit {
    var temperatura = 32.0
}
```

## Inicializadores personalizados

Podemos proporcionar parámetros de inicialización como parte de la definición de un inicializador, para definir los tipos y los nombres de los valores que personalizan el proceso de inicialización. Los parámetros de inicialización tienen las mismas capacidades y sintaxis que los parámetros de funciones y métodos.

```
struct Celsius {
    var temperaturaEnCelsius: Double
    init(desdeFahrenheit fahrenheit: Double) {
        temperaturaEnCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(desdeKelvin kelvin: Double) {
        temperaturaEnCelsius = kelvin - 273.15
    }
}

let puntoDeEbullicionDelAgua = Celsius(desdeFahrenheit: 212.0)
// puntoDeEbullicionDelAgua.temperaturaEnCelsius es 100.0
let puntoDeCongelacionDelAgua = Celsius(desdeKelvin: 273.15)
// puntoDeCongelacionDelAgua.temperaturaEnCelsius es 0.0
```

Vemos que dependiendo del nombre de parámetro proporcionado se escoge un inicializador u otro. En los inicializadores es obligatorio proporcionar los nombres de todos los parámetros:

```
struct Color {
    let rojo, verde, azul: Double
    init(rojo: Double, verde: Double, azul: Double) {
        self.rojo = rojo
        self.verde = verde
        self.azul = azul
    }
    init(blanco: Double) {
        rojo = blanco
        verde = blanco
        azul = blanco
    }
}
let magenta = Color(rojo: 1.0, verde: 0.0, azul: 1.0)
let medioGris = Color(blanco: 0.5)
```

Podemos evitar proporcionar nombres externos usando un subrayado. Por ejemplo, podemos añadir al struct anterior `Celsius` un inicializador sin nombre externo para el caso en que pasemos la temperatura inicial

precisamente en Celsius:

```
struct Celsius {
    var temperaturaEnCelsius: Double
    init(desdeFahrenheit fahrenheit: Double) {
        temperaturaEnCelsius = (fahrenheit - 32.0) / 1.8
    }
    init(desdeKelvin kelvin: Double) {
        temperaturaEnCelsius = kelvin - 273.15
    }
    init(_ celsius: Double) {
        temperaturaEnCelsius = celsius
    }
}

let temperaturaCuerpo = Celsius(37.0)
// temperaturaCuerpo.temperaturaEnCelsius es 37.0
```

Por último, es posible inicializar propiedades constantes definidas con `let`. Sólo toman valor en el momento de la inicialización y después no pueden modificarse.

```
class PreguntaEncuesta {
    let texto: String
    var respuesta: String?
    init(texto: String) {
        self.texto = texto
    }
    func pregunta() {
        print(texto)
    }
}
let preguntaQueso = PreguntaEncuesta(texto: "¿Te gusta el queso?")
preguntaQueso.pregunta() // -> "¿Te gusta el queso?
preguntaQueso.respuesta // -> nil
```

La propiedad `respuesta` se inicializa a `nil` al ser un opcional y no inicializarla en el inicializador.

Por último, es posible definir más de un inicializador, así como invocar a inicializadores más básicos desde otros.

```
struct Rectangulo {
    var origen = Punto()
    var tamaño = Tamaño()
    init(){}
    init(origen: Punto, tamaño: Tamaño) {
        self.origen = origen
        self.tamaño = tamaño
    }
}
```

```

init(centro: Punto, tamaño: Tamaño) {
    let origenX = centro.x - (tamaño.ancho / 2)
    let origenY = centro.y - (tamaño.ancho / 2)
    self.init(origen: Punto(x: origenX, y: origenY), tamaño: tamaño)
}
}

let basicRectangulo = Rectangulo()
// el origen de basicRectangulo es (0.0, 0.0) y su tamaño (0.0, 0.0)
let origenRectangulo = Rectangulo(origen: Punto(x: 2.0, y: 2.0),
                                    tamaño: Tamaño(ancho: 5.0, alto: 5.0))
// el origene de origenRectangulo es (2.0, 2.0) y su tamaño (5.0, 5.0)
let centroRectangulo = Rectangulo(centro: Punto(x: 4.0, y: 4.0),
                                    tamaño: Tamaño(ancho: 3.0, alto: 3.0))
// el origen de centroRectangulo es (2.5, 2.5) y su tamaño (3.0, 3.0)

```

El inicializador `init(){}` permite inicializar el `Rectangulo` a los valores por defecto definidos en las propiedades. Proporciona la misma funcionalidad que el inicializador por defecto, que tal y como hemos comentado, no se crea en una estructura o clase en la que definimos sus propios inicializadores.

## Herencia

Una clase puede *heredar* métodos, propiedades y otras características de otra clase. Cuando una clase hereda de otra, la clase que hereda se denomina *subclase*, y la clase de la que se hereda se denomina su *superclase*. La herencia es una conducta fundamental que diferencia las clases de otros tipos en Swift.

Las clases en Swift pueden llamar y acceder a métodos y propiedades que pertenecen a su superclase y pueden proporcionar sus propias versiones que sobreescreiben esos métodos y propiedades. Para sobreescribir un método o una propiedad es necesario cumplir con la definición proporcionada por la superclase.

Las clases también pueden añadir observadores a las propiedades heredadas para ser notificadas cuando cambia el valor de una propiedad. A cualquier propiedad se le puede añadir un observador de propiedad, independientemente de si es originalmente una propiedad almacenada o calculada.

### Definición de una clase base

Una clase que no hereda de ninguna otra se denomina una *clase base (base class)*. A diferencia de otros lenguajes orientados a objetos, las clases en Swift no heredan de una clase base universal.

También a diferencia de otros lenguajes orientados a objetos Swift no permite definir clases *abstractas* que no permiten crear instancias. Cualquier clase en Swift puede ser instanciada.

El siguiente ejemplo define una clase base llamada `Vehiculo`. Esta clase base define una propiedad almacenada llamada `velocidadActual` con un valor por defecto de 0.0. Esta propiedad se utiliza por una propiedad `String` calculada llamada `descripcion` que crea una descripción del vehículo.

La clase base `Vehiculo` también define un método llamdo `hazRuido`. Este método no hace nada realmente para una instancia de un vehículo base, pero será modificado más adelante por las subclases de `Vehiculo`.

```
class Vehiculo {
    var velocidadActual = 0.0
    var descripcion: String {
        return "viajando a \$(velocidadActual) kilómetros por hora"
    }
    func hazRuido() {
        // no hace nada - un vehículo arbitrario no hace ruido necesariamente
    }
}
```

Creamos una instancia nueva de `Vehiculo` con la sintaxis de inicialización que hemos visto, en la que se escribe el nombre del tipo de la clase seguido por paréntesis vacíos:

```
let unVehiculo = Vehiculo()
```

Habiendo creado una instancia nueva de `Vehiculo`, podemos acceder a su descripción:

```
print("Vehículo: \$(unVehiculo.descripcion)")
// Vehículo: viajando a 0.0 kilómetros por hora
```

La clase `Vehiculo` define características comunes para un vehículo arbitrario, pero no es de mucha utilidad por si misma. Para hacerla más útil, tenemos que refinarla para describir tipos de vehículos más específicos.

## Construcción de subclases

La construcción de una subclase (*subcassing*) es la acción de basar una nueva clase en una clase existente. La subclase hereda características de la clase existente, que después podemos refinar. También podemos añadir nuevas características a la subclase.

Para indicar que una subclase tiene una superclase hay que escribir el nombre de la subclase antes de el de la superclase, separadas por dos puntos (:):

```
class UnaSubclase: UnaSuperClase {
    // definición de la subclase
}
```

En el ejemplo anterior del `Vehiculo` podemos definir una subclase `Bicicleta`:

```
class Bicicleta: Vehiculo {
    var tieneCesta = false
}
```

La nueva clase **Bicicleta** obtiene automáticamente todas las características del **Vehiculo**, como sus propiedades **velocidadActual** y **descripcion** y su método **haceRuido()**.

Además de las características que hereda, la clase **Bicicleta** define una nueva propiedad almacenada, **tieneCesta**, con un valor por defecto de **false**.

Por defecto, cualquier instancia nueva de **Bicicleta** no tendrá una cesta. Puedes establecer la propiedad **tieneCesta** a **true** para una instancia particular de **Bicicleta** después de crearla:

```
let bicicleta = Bicicleta()
bicicleta.tieneCesta = true
```

Podemos también modificar la propiedad heredada **velocidadActual** y preguntar por la propiedad **descripcion**:

```
bicicleta.velocidadActual = 10.0
print("Bicicleta: \(bicicleta.descripcion)")
// Bicicleta: viajando a 10.0 kilómetros por hora
```

Podemos construir subclases a partir de otras subclases. El siguiente ejemplo crea una subclase de **Bicicleta** que representa una bicicleta de dos sillines (un "tandem"):

```
class Tandem: Bicicleta {
    var numeroActualDePasajeros = 0
}
```

**Tandem** hereda todas las propiedades y métodos de **Bicicleta**, que a su vez hereda todas sus propiedades y métodos de **Vehiculo**. La subclase **Tandem** también añade una nueva propiedad almacenada llamada **numeroActualDePasajeros**, con un valor por defecto de 0.

Si creamos una instancia de **Tandem** podremos trabajar con cualquiera de sus propiedades nuevas y heredadas, y preguntar a la descripción de solo lectura que hereda de **Vehiculo**:

```
let tandem = Tandem()
tandem.tieneCesta = true
tandem.numeroActualDePasajeros = 2
tandem.velocidadActual = 18.0
print("Tandem: \(tandem.descripcion)")
// Tandem: viajando a 18.0 kilómetros por hora
```

## Sobreescritura

Una subclase puede proporcionar su propia implementación de un método de la instancia, método del tipo, propiedad de la instancia o propiedad del tipo que hereda de su superclase. Esto se conoce como

sobreescritura (*overriding*).

Para sobreescribir una característica que sería de otra forma heredada debemos usar el prefijo `override`. De esta forma se clarifica que intentamos proporcionar una sobreescritura y que no lo hacemos por error. Esta palabra clave también hace que el compilador comprueba que la superclase (o una de sus clases padre) tiene una declaración que empareja con la que proporcionamos en la sobreescritura.

Cuando proporcionamos una sobreescritura puede ser útil acceder a los valores proporcionados por la clase padre. Para acceder a ellos podemos usar el prefijo `super`.

El siguiente ejemplo define una nueva subclase de `Vehiculo` llamada `Tren`, que sobreescribe el método `hazRuido()`:

```
class Tren: Vehiculo {
    override func hazRuido() {
        print("Chuu Chuu")
    }
}
```

Si creamos una nueva instancia de `Tren` y llamamos al método `hazRuido` podemos comprobar que se llama a la versión del método de la subclase:

```
let tren = Tren()
tren.hazRuido()
// Imprime "Chuu Chuu"
```

Podemos sobreescribir cualquier propiedad heredada del tipo o de la instancia y proporcionar nuestros propios *getters* y *setters* para esa propiedad, o añadir observadores de propiedades para observar cuando cambian los valores subyacentes de la propiedad.

Podemos proporcionar un *getter* (o *setter*, si es apropiado) para sobreescribir cualquier propiedad heredada, independientemente de si la propiedad heredada se implementa como una propiedad almacenada o calculada. La naturaleza almacenada o calculada no se conoce por la subclase, que solo conoce su nombre y su tipo. Es posible convertir una propiedad heredada de solo-lectura en una de lectura-escritura. No es posible presentar una propiedad heredada de lectura-escritura como de solo-lectura.

El siguiente ejemplo define una nueva clase llamada `Coche`, que es una subclase de `Vehiculo`. La clase `Coche` introduce una nueva propiedad almacenada llamada `marcha`, con un valor por defecto de 1. También sobreescribe la propiedad heredada `descripcion`, incluyendo la marcha actual en la descripción:

```
class Coche: Vehiculo {
    var marcha = 1
    override var descripcion: String {
        return super.descripcion + " con la marcha \\" + (marcha)
    }
}
```

Podemos ver el funcionamiento en el siguiente ejemplo:

```
let coche = Coche()
coche.velocidadActual = 50.0
coche.marcha = 3
print("Coche: \(coche.descripcion)")
// Coche: viajando a 50.0 kilómetros por hora con la marcha 3
```

Por último, podemos añadir observadores a propiedades heredadas. Esto nos permite ser notificados cuando el valor de una propiedad heredada cambia, independientemente de si esa propiedad se ha implementado en la subclase o en la superclase.

El siguiente ejemplo define una nueva clase llamada `CocheAutomatico` que es una subclase de `Coche`. La clase `CocheAutomatico` representa un coche con una caja de cambios automática, que selecciona automáticamente la marcha basándose en la velocidad actual:

```
class CocheAutomatico: Coche {
    override var velocidadActual: Double {
        didSet {
            marcha = min(Int(velocidadActual / 25.0) + 1, 5)
        }
    }
}
```

En cualquier momento que se modifica la propiedad `velocidadActual` de una instancia de `CocheAutomatico`, el observador `didSet` establece la propiedad `marcha` a un valor apropiado para la nueva velocidad. Un ejemplo de ejecución:

```
let automatico = CocheAutomatico()
automatico.velocidadActual = 100.0
print("CocheAutomatico: \(automatico.descripcion)")
// CocheAutomatico: viajando a 100.0 kilómetros por hora con la marcha 5
```

Por último, es posible prevenir un método o propiedad de ser sobreescrito declarándolo como *final*. Para ello, hay que escribir el modificador `final` antes del nombre de la palabra clave que introduce el método o la propiedad (como `final var`, `final func`).

También es posible marcar la clase completa como *final*, escribiendo el modificador antes de `class` (`final class`).

## Protocolos

Un *protocolo* (*protocol*) define un esquema de métodos, propiedades y otros requisitos que encajan en una tarea particular o un trozo de funcionalidad. El protocolo puede luego ser *adoptado* (*adopted*) por una clase,

estructura o enumaración para proporcionar una implementación real de esos requisitos. Cualquier tipo que satisface los requerimientos de un protocolo se dice que *se ajusta o cumple (conform)* ese protocolo.

Además de especificar los requisitos de los tipos que cumplen el protocolo, también se puede **extender un protocolo** (lo veremos más adelante, cuando hablemos de **extensiones**) para proporcionar una implementación de algunos de los métodos requeridos por el protocolo.

## Sintaxis

Los protocolos se definen de forma similar a las clases, estructuras y enumeraciones:

```
protocol UnProtocolo {  
    // definición del protocolo  
}
```

Para definir un tipo que se ajusta a un protocolo particular se debe poner el nombre del protocolo tras el nombre del tipo, separado por dos puntos. Podemos listar más de un protocolo, y se separan por comas:

```
struct UnStruct: PrimerProtocolo, OtroProtocolo {  
    // definición del struct  
}
```

Si una clase tiene una superclase, se escribe el nombre de la superclase antes de los protocolos, seguido por una coma:

```
class UnaClase: UnaSuperClase, PrimerProtocolo, OtroProto {  
    // definición de la clase  
}
```

## Requisitos de propiedades

Un protocolo puede requerir a cualquier tipo que se ajuste a él que proporcione una propiedad de instancia o de tipo con un nombre y tipo particular. El protocolo no especifica si la propiedad es una propiedad calculada o almacenada, sólo especifica el nombre y el tipo de la propiedad requerida. El protocolo también especifica si la propiedad debe ser de lectura y escritura o sólo de lectura.

Si un protocolo requiere que una propiedad sea de lectura y escritura, el requisito no puede ser satisfecho por una propiedad constante almacenada o por una propiedad calculada de sólo lectura. Si el protocolo sólo requiere que la propiedad sea de lectura, el requisito puede ser satisfecho por cualquier tipo de propiedad, y es válido que la propiedad sea también de escritura si es útil para nuestro propio código.

Los requisitos de la propiedad se declaran siempre como propiedades variables, precedido por la palabra clave **var**. Las propiedades de lectura y escritura se indican escribiendo **{ get set }** después de la declaración de su tipo, y las propiedades de sólo lectura se indican escribiendo **{ get }**.

```
protocol UnProtocolo {
    var debeSerEscribible: Int { get set }
    var noTienePorQueSerEscribible: Int { get }
}
```

Para definir una propiedad de tipo hay que precederla en el protocolo con la palabra clave **static**:

```
protocol OtroProtocolo {
    static var unaPropiedadDeTipo: Int { get set }
}
```

Veamos un ejemplo. Definimos el protocolo **TieneNombre** en el que se requiere que cualquier clase que se ajuste a él debe tener una propiedad de instancia de lectura de tipo **String** que se llame **nombreCompleto**:

```
protocol TieneNombre {
    var nombreCompleto: String { get }
}
```

Un ejemplo de una sencilla estructura que adopta el protocolo:

```
struct Persona: TieneNombre {
    var edad: Int
    var nombreCompleto: String
}

let john = Persona(edad: 35, nombreCompleto: "John Appleseed")
// john.nombreCompleto es "John Appleseed"
```

Este ejemplo define una estructura llamada **Persona**, que representa una persona con una edad y un nombre específico. En la primera línea se declara que se adopta el protocolo **TieneNombre**. Cada instancia de **Persona** tiene la propiedad almacenada llamada **nombreCompleto**, que es de tipo **String**. Esto cumple el único requisito del protocolo **TieneNombre**, y significa que **Persona** se ajusta correctamente al protocolo (Swift informa de un error en tiempo de compilación si un requisito de un protocolo no se cumple).

Otro ejemplo de una clase más compleja, que también adopta el protocolo:

```
class NaveEstelar: TieneNombre {
    var prefijo: String?
    var nombre: String
    init(nombre: String, prefijo: String? = nil) {
        self.nombre = nombre
        self.prefijo = prefijo
    }
    var nombreCompleto: String {
```

```

        return (prefijo != nil ? prefijo! + " " : "") + nombre
    }
}

var ncc1701 = NaveEstelar(nombre: "Enterprise", prefijo: "USS")
// ncc1701.nombreCompleto es "USS Enterprise"

```

Esta clase implementa el requisito de la propiedad `nombreCompleto` como una propiedad calculada de solo lectura para una nave estelar. Cada instancia de `NaveEstelar` almacena un nombre obligatorio y un prefijo opcional. La propiedad `nombreCompleto` usa el valor del prefijo si existe, y la añade al comienzo del nombre para crear un nombre completo de la nave estelar.

## Requisitos de métodos

Los protocolos pueden requerir que los tipos que se ajusten a ellos implementen métodos de instancia y de tipos específicos. Estos métodos se escriben como parte de la definición del protocolo de la misma forma que los métodos normales, pero sin sus cuerpos:

```

protocol UnProtocolo {
    func unMetodo() -> Int
}

```

Los métodos del tipo en el protocolo deben indicarse con la palabra clave `static`:

```

protocol UnProtocolo {
    static func unMetodoDelTipo()
}

```

Un ejemplo:

```

protocol GeneradorNumerosAleatorios {
    func random() -> Double
}

```

Este protocolo, `GeneradorNumerosAleatorios`, requiere que cualquier tipo que se ajuste a él tenga un método de instancia llamado `random`, que devuelve un valor `Double` cada vez que se llama. Aunque no está especificado en el protocolo, se asume que este valor será un número entre 0.0 y 1.0 (sin incluirlo). El protocolo `GeneradorNumerosAleatorios` no hace ninguna suposición sobre cómo será generado cada número aleatorio, simplemente requiere al generador que proporcione una forma estándar de generarlos.

Una implementación de una clase que adopta el protocolo:

```

class GeneradorLinealCongruente: GeneradorNumerosAleatorios {
    var ultimoRandom = 42.0
    let m = 139968.0
}

```

```

let a = 3877.0
let c = 29573.0
func random() -> Double {
    let number = ultimoRandom * a + c
    ultimoRandom = number.truncatingRemainder(dividingBy: m)
    return ultimoRandom / m
}
let generador = GeneradorLinealCongruente()
print("Un número aleatorio: \(generador.random())")
// Imprime "Un número aleatorio: 0.37464991998171"
print("Y otro: \(generador.random())")
// Imprime "Y otro: 0.729023776863283"

```

## Requisito de método `mutating`

Si definimos un protocolo con un requisito de método de instancia que pretenda mutar las instancias del tipo que adopte el protocolo, se debe marcar el método con la palabra `mutating`. Esto permite a las estructuras y enumeraciones que adopten el protocolo definir ese método como `mutating`. No es necesario hacerlo con las clases, porque la palabra `mutating` solo es necesaria en estructuras y enumeraciones.

Un ejemplo:

```

protocol Conmutable {
    mutating func conmutar()
}

enum Interruptor: Conmutable {
    case apagado, encendido
    mutating func conmutar() {
        switch self {
            case .apagado:
                self = .encendido
            case .encendido:
                self = .apagado
        }
    }
}

var interruptorLampara = Interruptor.apagado
interruptorLampara.conmutar()
// interruptorLampara es ahora igual a .encendido

```

## Protocolos como tipos

Los protocolos no implementan realmente ninguna funcionalidad por ellos mismos. Sin embargo, cualquier protocolo que definamos se convierte automáticamente en un tipo con todas sus propiedades que podemos usar en nuestro código.

Podemos entonces usar el protocolo en cualquier sitio donde permitamos otros tipos, incluyendo:

- El tipo de un parámetro de una función, método o inicializador o de sus valores devueltos.
- El tipo de una constante, variable o propiedad
- El tipo de los ítems de un array, diccionario u otro contenedor

```
class Dado {
    let caras: Int
    let generador: GeneradorNumerosAleatorios
    init(caras: Int, generador: GeneradorNumerosAleatorios) {
        self.caras = caras
        self.generador = generador
    }
    func tirar() -> Int {
        return Int(generador.random() * Double(caras)) + 1
    }
}
```

Este ejemplo define una nueva clase llamada `Dado`, que representa un dado de  $n$  caras que se puede usar en un juego de tablero. Las instancias de dados tienen una propiedad llamada `caras`, que representa cuántas caras tienen, y una propiedad llamada `generador`, que proporciona un generador a partir del cual crear valores de tiradas.

La propiedad `generador` es del tipo `GeneradorNumerosAleatorios`. Podemos asignarle una instancia de cualquier tipo que adopte el protocolo `GeneradorNumerosAleatorios`.

`Dado` tiene también un inicializador, para configurar sus estado inicial. El inicializador tiene un parámetro llamado `generador`, que también es del tipo `GeneradorNumerosAleatorios`. Podemos pasarle un valor de cualquier instancia que se ajuste a este tipo. Y también proporciona un método de instancia llamado `tirar`, que devuelve un valor entero entre 1 y el número de caras del dado. Este método llama al método `random()` del generador para crear un nuevo número aleatorio entre 0.0 y 1.0 y usa este número aleatorio para crear un valor de tirada que esté dentro del rango correcto. Debido a que sabemos que el generador se ajusta al protocolo `GeneradorNumerosAleatorios` tenemos la garantía de que va a existir un método `random()` al que llamar.

Un ejemplo de uso del código:

```
var d6 = Dado(caras: 6, generador: GeneradorLinealCongruente())
for _ in 1...5 {
    print("La tirada del dado es \(d6.tirar())")
}
// La tirada del dado es 3
// La tirada del dado es 5
// La tirada del dado es 4
// La tirada del dado es 5
// La tirada del dado es 4
```

## Colecciones de tipos protocolo

Como hemos comentado anteriormente, un protocolo puede usarse como el tipo que se almacena en una colección (array, diccionario, etc.). Veamos un ejemplo:

```
var peterParker = Persona(edad: 24, nombreCompleto: "Peter Parker")
var ncc1701 = NaveEstelar(nombre: "Enterprise", prefijo: "USS")

let cosasConNombre: [TieneNombre] = [peterParker, ncc1701]

for cosa in cosasConNombre {
    print(cosa.nombreCompleto)
}
// Peter Parker
// USS Enterprise
```

Hay que hacer notar que la constante `cosa` que itera sobre los elementos del array es de tipo `TieneNombre`, no es de tipo `Persona` ni de tipo `NaveEstelar`, incluso aunque las instancias que hay tras de escena son de esos tipos. Por ser del tipo `TieneNombre` sabemos que tiene una propiedad `nombreCompleto` que podemos usar sobre la variable iteradora.

## Protocolo `Equatable`

En la [biblioteca estándar de Swift](#) se definen distintos protocolos como `Collection` y `Equatable` que describen abstracciones comunes. Muchos de estos protocolos incorporan implementaciones por defecto de algunos de sus métodos mediante extensiones definidas en la propia biblioteca estándar.

Veamos por ejemplo el protocolo `Equatable`. Se trata de un protocolo importante que define las operaciones de igualdad (`==`) y diferencia (`!=`). Debemos implementar la operación de igualdad en cualquier clase que se ajuste al protocolo, pero la operación de diferencia ya tiene una implementación por defecto.

Un ejemplo:

```
class Punto3D: Equatable {
    let x, y, z: Double

    init(x: Double, y: Double, z: Double) {
        self.x = x
        self.y = y
        self.z = z
    }

    static func == (izquierda: Punto3D, derecha: Punto3D) -> Bool {
        return
            izquierda.x == derecha.x &&
            izquierda.y == derecha.y &&
            izquierda.z == derecha.z
    }
}
```

```

let p1 = Punto3D(x: 0.0, y: 0.0, z: 0.0)
let p2 = Punto3D(x: 0.0, y: 0.0, z: 0.0)

print(p1 == p2)
// Imprime true
print(p1 != p2)
// Imprime false

```

El operador `==` se define en la propia estructura. Se utiliza la palabra `static` para indicar que se trata de un operador que estamos sobrecargando (hablaremos más adelante de los operadores).

El operador `!=` que se usa en la última instrucción se define en una implementación por defecto.

En **Swift 5** el compilador define una **implementación automática del operador `==` en las estructuras y enumeraciones** al añadir el protocolo `Equatable`, siempre que las propiedades almacenadas y los valores asociados cumplan ese protocolo.

Por ejemplo, si en lugar de una clase definimos un `struct Punto3D` el código quedaría como sigue (no es necesario definir ni el inicializador por defecto ni el operador `==`):

```

struct Punto3D: Equatable {
    let x, y, z: Double
}

let p1 = Punto3D(x: 0.0, y: 0.0, z: 0.0)
let p2 = Punto3D(x: 0.0, y: 0.0, z: 0.0)

print(p1 == p2)
// Imprime true
print(p1 != p2)
// Imprime false

```

## Casting de tipos

El *casting* de tipos es una forma de comprobar el tipo de una instancia o de tratar esa instancia como de una superclase distinta o conseguir una subclase de algún otro sitio en la propia jerarquía de clase. La forma de implementarlo es utilizando los operadores `is` y `as`. Estos operadores proporcionan una forma simple y expresiva de comprobar el tipo de un valor o transformar un valor en uno de otro tipo. También se puede usar el *casting* de tipos para comprobar si un tipo se ajusta a un protocolo.

### Una jerarquía de clases para el casting de tipos

Vamos a comenzar construyendo una jerarquía de clases y subclases con las que trabajar. Utilizaremos el *casting* de tipos para comprobar el tipo de una instancia particular de una clase y para convertir esa instancia en otra clase dentro de la misma jerarquía.

En el primer fragmento de código definimos una clase nueva llamada `MediaItem`. Esta clase proporciona la funcionalidad básica de cualquier tipo de ítem que aparece en una biblioteca de medios digitales.

Específicamente, declara una propiedad `nombre` de tipo `String` y un inicializador `init nombre` (suponemos que todos los ítems, incluyendo películas y canciones, tendrán un nombre).

```
class MediaItem {
    var nombre: String
    init(nombre: String) {
        self.nombre = nombre
    }
}
```

El siguiente fragmento define dos subclases de `MediaItem`. La primera subclase, `Pelicula`, encapsula información adicional sobre una película. Añade una propiedad `director` a la clase base `MediaItem`, con su correspondiente inicializador. La segunda subclase, `Cancion`, añade una propiedad `artista` y un inicializador a la clase base:

```
class Pelicula: MediaItem {
    var director: String
    init(nombre: String, director: String) {
        self.director = director
        super.init(nombre: nombre)
    }
}

class Cancion: MediaItem {
    var artista: String
    init(nombre: String, artista: String) {
        self.artista = artista
        super.init(nombre: nombre)
    }
}
```

Por último, creamos un array constante llamado `biblioteca`, que contiene dos instancias de `Pelicula` y tres instancias de `Cancion`.

```
let biblioteca: [MediaItem] = [
    Pelicula(nombre: "El Señor de los Anillos", director: "Peter Jackson"),
    Cancion(nombre: "Child in Time", artista: "Deep Purple"),
    Pelicula(nombre: "El Puente de los Espías", director: "Steven Spielberg"),
    Cancion(nombre: "I Wish You Were Here", artista: "Pink Floyd"),
    Cancion(nombre: "Yellow", artista: "Coldplay")
]
```

Podríamos también dejar que el compilador infiera el tipo del array. Es capaz de deducir que `Pelicula` y `Cancion` tienen una superclase común `MediaItem`, por lo que **infiere que el tipo del array es `[MediaItem]`**:

```
// Declaración equivalente a la anterior
let biblioteca = [
    Pelicula(nombre: "El Señor de los Anillos", director: "Peter Jackson"),
    Cancion(nombre: "Child in Time", artista: "Deep Purple"),
    Pelicula(nombre: "El Puente de los Espías", director: "Steven Spielberg"),
    Cancion(nombre: "I Wish You Were Here", artista: "Pink Floyd"),
    Cancion(nombre: "Yellow", artista: "Coldplay")
]
```

Los ítems almacenados en la biblioteca son todavía instancias de `Pelicula` y `Cancion`. Sin embargo, si iteramos sobre los contenidos de este array, los ítems que recibiremos tendrán el tipo `MediaItem` y no `Pelicula` o `Cancion`. Para trabajar con ellos como su tipo nativo, debemos chequear su tipo, y hacer un *downcast* a su tipo concreto.

## Comprobación del tipo

Podemos usar el *operador de comprobación (check operator)* `is` para comprobar si una instancia es de un cierto tipo subclase. El operador de comprobación devuelve `true` si la instancia es del tipo de la subclase y `false` si no.

Lo podemos comprobar en el siguiente ejemplo, en el que contamos las instancias de películas y canciones en el array `biblioteca`:

```
var contadorPeliculas = 0
var contadorCanciones = 0

for item in biblioteca {
    if item is Pelicula {
        contadorPeliculas += 1
    } else if item is Cancion {
        contadorCanciones += 1
    }
}

print("La biblioteca contiene \n(contadorCanciones) películas y \
      \n(contadorPeliculas) canciones")
// Imprime "La biblioteca contiene 3 películas y 2 canciones"
```

El ejemplo itera por todos los ítems del array `biblioteca`. En cada paso, el bucle `for-in` guarda en la constante `item` el siguiente `MediaItem` del array.

La instrucción `item is Pelicula` devuelve `true` si el `MediaItem` actual es una instancia de `Pelicula` y `false` en otro caso. De forma similar, `item is Cancion` comprueba si el ítem es una instancia de `Cancion`. Al final del bucle `for-in`, los valores de `contadorPeliculas` y `contadorCanciones` contendrán una cuenta de cuantas instancias `MediaItem` de cada tipo se han encontrado.

## Downcasting

Una constante o variable de un cierto tipo de clase puede referirse (contener) a una instancia de una subclase. Cuando creemos que sucede esto, podemos intentar hacer un *downcast* al tipo de la subclase con un operador de *cast* (`as?` o `as!`). Como el *downcast* puede fallar, la versión condicional, `as?`, devuelve un valor opcional del tipo al que estamos intentando hacer el *downcasting*. La versión forzosa, `as!`, intenta el *downcast* y fuerza la desenvoltura del resultado en un única acción compuesta.

Debemos usar la versión condicional (`as?`) cuando no estamos seguros si el *downcast* tendrá éxito. Se devolverá un valor opcional y el valor será `nil` si no es posible hacer el *downcast*. Esto permitirá comprobar si ha habido un *downcast* con éxito.

La otra versión (`as!`) se usa sólo cuando estamos seguros de que el *downcast* tendrá éxito. Esta versión del operador lanzará un error en tiempo de ejecución si intentamos hacer un *downcast* a un tipo incorrecto.

El siguiente ejemplo itera sobre cada `MediaItem` en `biblioteca`, e imprime una descripción apropiada para cada ítem. Para hacerlo, necesita acceder a cada ítem como una `Pelicula` o `Cancion` y no sólo como una `MediaItem`. Esto es necesario para poder acceder a la propiedad `director` o `artista` de una instancia de `Pelicula` o `Cancion`.

En este ejemplo, cada ítem en el array podría ser un `Pelicula` o podría ser una `Cancion`. No sabemos por anticipado la clase verdadera de cada ítem, por lo que es apropiado usar la versión condicional (`as?`) para comprobar el *downcast* cada vez a lo largo del bucle:

```
for item in biblioteca {
    if let pelicula = item as? Pelicula {
        print("Película: \(pelicula.nombre), dir. \(pelicula.director)")
    } else if let cancion = item as? Cancion {
        print("Cancion: \(cancion.nombre), de \(cancion.artista)")
    }
}

// Película: El Señor de los Anillos, dir. Peter Jackson
// Cancion: Child in Time, de Deep Purple
// Película: El Puente de los Espías, dir. Steven Spielberg
// Cancion: I Wish You Were Here, de Pink Floyd
// Cancion: Yellow, de Coldplay
```

El ejemplo comienza intentando hacer *downcast* del ítem a una `Pelicula`. Debido a que es una instancia de `MediaItem`, es posible que sea un `Pelicula` o una `Cancion`, o incluso el tipo base `MediaItem`. Debido a esta incertidumbre, debemos usar la versión `as?` para devolver un valor opcional. El resultado será una "Pelicula opcional". Podemos desenvolver el valor `Pelicula` usando un `if let` como vimos en el apartado de opcionales. Si tiene éxito el *downcasting*, las propiedades de la película se pueden usar para imprimir una descripción de la película llamando a los correspondientes métodos de la clase `Pelicula`. Igual con `Cancion`.

## El tipo `Any`

El tipo `Any` puede representar una instancia de cualquier tipo, incluyendo tipos función:

```
var array = [Any]()

array.append(0)
array.append(0.0)
array.append(42)
array.append(3.14159)
array.append("hola")
array.append((3.0, 5.0))
array.append(Pelicula(nombre: "Ghostbusters", director: "Ivan Reitman"))
array.append({ (name: String) -> String in "Hola, \(name)" })
```

El array contiene dos valores `Int`, dos valores `Double`, un valor `String`, una tupla del tipo `(Double, Double)`, la película "Ghostbusters", y una clausura que toma un `String` y devuelve otro `String`.

Puedes usar los operadores `is` y `as` en una sentencia `switch` para descubrir en tiempo de ejecución el tipo específico de una constante o variable de la que sólo se sabe que es de tipo `Any`:

```
for item in array {
    switch item {
        case 0 as Int:
            print("cero como un Int")
        case 0 as Double:
            print("cero como un Double")
        case let someInt as Int:
            print("un valor entero de \(someInt)")
        case let unDouble as Double where unDouble > 0:
            print("a valor positivo de \(unDouble)")
        case is Double:
            print("algún otro valor double que no quiero imprimir")
        case let someString as String:
            print("una cadena con valor de \"\(someString)\"")
        case let (x, y) as (Double, Double):
            print("un punto (x, y) en \(x), \(y)")
        case let pelicula as Pelicula:
            print("una película: \(pelicula.nombre), dir. \(pelicula.director)")
        case let stringConverter as (String) -> String:
            print(stringConverter("Michael"))
        default:
            print("alguna otra cosa")
    }
}

// cero como un Int
// cero como un Double
// un valor entero de 42
// a valor positivo de 3.14159
// una cadena con valor de "hola"
// un punto (x, y) en 3.0, 5.0
// una película: Ghostbusters, dir. Ivan Reitman
// Hola, Michael
```

## Comprobación de ajustarse a un protocolo

Podemos usar también los operadores anteriores `is` y `as` (`y as?` y `as!`) para comprobar si una instancia se ajusta a un protocolo y para hacer un *cast* a un protocolo específico.

Veamos un ejemplo. Definimos el protocolo `TieneArea` con el único requisito de una propiedad de lectura llamada `area` de tipo `Double`:

```
protocol TieneArea {
    var area: Double { get }
}
```

Definimos dos clases `Circulo` y `Pais` que se ajustan ambos al protocolo:

```
class Circulo: TieneArea {
    let pi = 3.1415927
    var radio: Double
    var area: Double { return pi * radio * radio }
    init(radio: Double) { self.radio = radio }
}

class Pais: TieneArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

La clase `Circulo` implementa el requisito como una propiedad calculada, basada en la propiedad almacenada `radio`. La clase `Pais` implementa el requisito directamente como una propiedad almacenada. Ambas clases se ajustan correctamente al protocolo `TieneArea`.

Definimos una clase `Animal` que no se ajusta al protocolo:

```
class Animal {
    var patas: Int
    init(patas: Int) { self.patas = patas }
}
```

Las clases `Circulo`, `Pais` y `Animal` no tienen ninguna clase base compartida. Sin embargo, todas son clases, por lo que las instancias de los tres tipos pueden usarse para inicializar un array que almacena valores de tipo `Any`:

```
let objetos: [Any] = [
    Circulo(radio: 2.0),
    Pais(area: 243_610),
```

```
    Animal(patas: 4)
]
```

Y ahora podemos iterar sobre el array de objetos, comprobando para cada ítem si la instancia se ajusta al protocolo `TieneArea`:

```
for objeto in objetos {
    if let objetoConArea = objeto as? TieneArea {
        print("El área es \(objetoConArea.area)")
    } else {
        print("Algo que no tiene un área")
    }
}

// El área es 12.5663708
// El área es 243610.0
// Algo que no tiene un área
```

Cuando un objeto en el array se ajusta al protocolo `TieneArea`, el valor opcional devuelto por el operador `as?` se desenvuelve con un ligado opcional en una constante llamada `objetoConArea`. Esta constante tiene el tipo `TieneArea`, por lo que su propiedad `area` podrá ser accedida e impresa.

Hay que notar que los objetos subyacentes no cambian en el proceso de *casting*. Siguen siendo un `Circulo`, un `Pais` y un `Animal`. Sin embargo, en el momento en se almacenan en la constante `objetoConArea`, sólo se sabe que son del tipo `TieneArea`, por lo que sólo podremos acceder a su propiedad `area`.

## Extensiones

Las *extensiones* añaden nueva funcionalidad a una clase, estructura, enumeración o protocolo. Esto incluye la posibilidad de extender tipos para los que no tenemos acceso al código fuente original (esto se conoce como *modelado retroactivo*).

Entre otras cosas, las extensiones pueden:

- Añadir **propiedades calculadas** de instancia y de tipo
- Definir métodos de instancia y de tipo
- Proporcionar nuevos inicializadores
- Hacer que un tipo existente se ajuste a un protocolo

## Sintaxis

Para declarar una extensión hay que usar la palabra clave `extension`, indicando después el tipo que se quiere extender (enumeración, clase, estructura o protocolo)

```
extension UnTipoExistente {
    // nueva funcionalidad para añadir a UnTipo
}
```

## Propiedades calculadas

Las extensiones pueden añadir propiedades calculadas de instancias y de tipos. Como primer ejemplo, recordemos el tipo **Persona**:

```
protocol TieneNombre {
    var nombreCompleto: String { get }
}

struct Persona: TieneNombre {
    var edad: Int
    var nombreCompleto: String
}
```

Vamos a añadir a la estructura la propiedad calculada **mayorEdad**, un **Bool** que indica si la edad de la persona es mayor o igual de 18:

```
extension Persona {
    var mayorEdad: Bool {
        return edad >= 18
    }
}
```

Una vez definida esta extensión, hemos ampliado la clase con esta nueva propiedad, sin modificar el código inicial con la definición de la clase.

Podemos preguntar si una persona es mayor de edad:

```
var p = Persona(edad: 15, nombreCompleto: "Lucía")
p.mayorEdad // false
```

Es posible incluso extender clases de las librerías estándar de Swift, como **Int**, **Double**, **Array**, etc.

Por ejemplo, podemos añadir propiedades calculadas a la clase **Double** para trabajar con unidades de distancia. Las siguientes propiedades convierten una cantidad en las unidades correspondientes a su equivalente en metros:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
```

Una vez definida la extensión, podemos usarla en cualquier variable `Double`. Incluso la podemos usar en literales:

```
let distancia = 11.km
// En distancia tendremos 11000 metros
let unaPulgada = 25.4.mm
print("Una pulgada es \(unaPulgada) metros")
// Una pulgada es 0.0254 metros
let tresPies = 3.ft
print("Tres pies son \(tresPies) metros")
// Tres pies son 0.914399970739201 metros
```

Por ejemplo, cuando se escribe `11.km` se pide el valor de la propiedad calculada `km` de la instancia `11`. La propiedad calculada devuelve el resultado de multiplicar `11` por `1000`, esto es, los metros correspondientes a 11 kilómetros.

De forma similar, hay 3.28084 pies en un metro, por lo que la propiedad calculada `ft` divide el valor `Double` subyacente por 3.28084, para convertirlo de pies a metros.

Estas propiedades son propiedades calculadas de solo lectura, por lo que se expresan sin la palabra clave `get`, por brevedad. Sus valores devueltos son de tipo `Double`, y pueden usarse en cálculos matemáticos en cualquier sitio que se acepte un `Double`:

```
let unMaraton = 42.km + 195.m
print("Un maratón tiene una longitud de \(unMaraton) metros")
// Un maratón tiene una longitud de 42195.0 metros
```

## Inicializadores

Las extensiones pueden añadir nuevos inicializadores a tipos existentes. Esto nos permite extender otros tipos para aceptar nuestros propios tipos como parámetros de la inicialización, o para proporcionar opciones adicionales que no estaban incluidos en la implementación original del tipo.

Recordemos la estructura `Rectangulo`, definida por un `Punto` y un `Tamaño`. Supongamos que la definimos sin inicializadores:

```
struct Tamaño {
    var ancho = 0.0, alto = 0.0
}
struct Punto {
    var x = 0.0, y = 0.0
}
struct Rectangulo {
    var origen = Punto()
    var tamaño = Tamaño()
}
```

Recordemos que debido a que la estructura `Rectangulo` proporciona valores por defecto para todas sus propiedades, tiene un inicializador por defecto que puede utilizarse para crear nuevas instancias. También podemos inicializarlo asignando todas sus propiedades:

```
let rectanguloPorDefecto = Rectangulo()
let rectanguloInicializado = Rectangulo(origen: Punto(x: 2.0, y: 2.0),
   tamaño: Tamaño(ancho: 5.0, alto: 5.0))
```

Podemos ahora extender la estructura `Rectangulo` para proporcionar un inicializador adicional que toma un punto específico del centro y un tamaño:

```
extension Rectangulo {
    init(centro: Punto, tamaño: Tamaño) {
        let origenX = centro.x - (tamaño.ancho / 2)
        let origenY = centro.y - (tamaño.alto / 2)
        self.init(origen: Punto(x: origenX, y: origenY), tamaño: tamaño)
    }
}
```

Este nuevo inicializador empieza por calcular un punto de origen basado en el centro propuesto y en el tamaño. El inicializador llama después al inicializador automático de la estructura `init(origen:tamaño:)`, que almacena los nuevos valores de las propiedades:

```
let rectanguloCentro = Rectangulo(centro: Punto(x: 4.0, y: 4.0),
                                   tamaño: Tamaño(ancho: 3.0, alto: 3.0))
// el origen del rectanguloCentro es (2.5, 2.5) y su tamaño es (3.0, 3.0)
```

## Métodos

Las extensiones pueden añadir nuevos métodos de instancia y nuevos métodos del tipo.

Por ejemplo, podemos añadir el método `descripcion()` a la estructura `Persona`:

```
extension Persona {
    func descripción() -> String {
        return "Me llamo \(nombreCompleto) y tengo \(edad) años"
    }
}

let reedRichards = Persona(edad: 40, nombreCompleto: "Reed Richards")
print(reedRichards.descripcion())
```

También podemos añadir métodos a clases y estructuras importadas. Por ejemplo podemos añadir un nuevo método de instancia llamado `repeticiones` al tipo `Int`:

```
extension Int {
    func repeticiones(_ tarea: () -> Void) {
        for _ in 0..

```

El método `repeticiones(_:_)` toma un único argumento de tipo `() -> Void`, que indica una función que no tiene parámetros y no devuelve ningún valor. Después de definir esta extensión, podemos llamar al método `repeticiones(_:_)` en cualquier número entero para ejecutar una tarea un cierto número de veces:

```
3.repeticiones({
    print("Hola!")
})
// Hola!
// Hola!
// Hola!
```

Usando clausuras por la cola podemos hacer la llamada más concisa:

```
3.repeticiones {
    print("Adios!")
}
// Adios!
// Adios!
// Adios!
```

## Métodos de instancia mutadores

Los métodos de instancia añadidos con una extensión también pueden modificar (o mutar) la propia instancia. Los métodos de las estructuras y los enumerados que modifican `self` o sus propiedades deben marcarse como `mutating`.

Por ejemplo:

```
extension Int {
    mutating func cuadrado() {
        self = self * self
    }
}
var unInt = 3
```

```
unInt.cuadrado()
// unInt es ahora 9
```

## Ajustar un tipo a un protocolo mediante una extensión

Una extensión puede extender un tipo existente para hacer que se ajuste a uno o más protocolos. En este caso, los nombres de los protocolos se escriben exactamente de la misma forma que en una clase o estructura:

```
extension UnTipo: UnProtocolo, OtroProtocolo {
    // implementación de los requisitos del protocolo
}
```

Las instancias del tipo adoptarán automáticamente el protocolo y se ajustarán al protocolo con las propiedades y métodos añadidos por la extensión.

Por ejemplo, este protocolo, llamado `RepresentableComoTexto` puede ser implementado por cualquier tipo que tenga una forma de representarse como texto. Esto puede ser una descripción de si mismo o una versión de texto de su estado actual:

```
protocol RepresentableComoTexto {
    var descripcionTextual: String { get }
}
```

La clase `Dado` que vimos anteriormente puede extenderse para ajustarse al protocolo:

```
extension Dado: RepresentableComoTexto {
    var descripcionTextual: String {
        return "Un dado de \(caras) caras"
    }
}
```

Esta extensión adopta el nuevo protocolo exactamente como si el `Dado` lo hubiera proporcionado en su implementación original. El nombre del protocolo se indica tras el nombre del tipo, separado por dos puntos, y se proporciona una implementación entre llaves.

Cualquier instancia de un dado se puede tratar ahora como `RepresentableComoTexto`:

```
let d12 = Dado(caras: 12, generador: GeneradorLinealCongruente())
print(d12.descripcionTextual)
// Un dado de 12 caras
```

De forma similar, un `Rectangulo` también puede extenderse para adoptar y ajustarse al protocolo:

```

extension Rectangulo: RepresentableComoTexto {
    var descripcionTextual: String {
        return "Un rectángulo situado en (\(origen.x), \(origen.y))"
    }
}
let rectanguloInicializado = Rectangulo(origen: Punto(x: 2.0, y: 2.0),
   tamaño: Tamaño(ancho: 5.0, alto: 5.0))

print(rectanguloInicializado.descripcionTextual)
// Un rectángulo situado en (2.0, 2.0)

```

## Declaración de la adopción de un protocolo con una extensión

Si un tipo ya se ajusta a todos los requisitos de un protocolo, pero todavía no se ha declarado que se ajusta al protocolo, podemos hacer que lo adopte con una extensión vacía:

```

struct Hamster {
    var nombre: String
    var descripcionTextual: String {
        return "Un hamster llamado \(nombre)"
    }
}
extension Hamster: RepresentableComoTexto {}

```

Las instancias de `Hamster` pueden ahora usarse en cualquier sitio que se requiera un tipo `RepresentableComoTexto`:

```

let simonElHamster = Hamster(nombre: "Simon")
let algoRepresentableComoTexto: RepresentableComoTexto = simonElHamster
print(algoRepresentableComoTexto.descripcionTextual)
// Un hamster llamado Simon

```

## Implementación de métodos de un protocolo

Podemos definir extensiones en los protocolos para proporcionar implementaciones de métodos y propiedades a todos los tipos que se ajustan a él. Esto permite definir conductas en los propios protocolos, más que en cada tipo individual o en una función global.

Por ejemplo, el protocolo `GeneradorNumerosAleatorios` puede ser extendido para proporcionar un método `boolAleatorio()`, que usa el resultado del `random()` requerido para devolver un valor `Bool` aleatorio:

```

extension GeneradorNumerosAleatorios {
    func randomBool() -> Bool {
        return random() > 0.5
    }
}

```

```

    }
}
```

Al crear una extensión en el protocolo, todos los tipos que se ajustan a él adquieren automáticamente esta implementación sin ninguna modificación adicional.

```

let generator = GeneradorLinealCongruente()
print("Un número aleatorio: \(generator.random())")
// Imprime "Un número aleatorio: 0.37464991998171"
print("Y un booleano aleatorio: \(generator.randomBool())")
// Imprime "Un booleano aleatorio: true"
```

El tipo que se ajusta al protocolo puede proporcionar su propia implementación, que se usará en lugar de la proporcionada por la extensión.

## Funciones operadoras

Las clases y las estructuras pueden proporcionar sus propias implementaciones de operadores existentes. Esto se conoce como *sobrecarga* de los operadores existentes.

En el siguiente ejemplo se muestra cómo implementar el operador de suma (+) para una estructura. El operador suma es un operador binario (tiene dos operandos) e infijo (aparece junto entre los dos operandos). Definimos una estructura `Vector2D` para un vector de posición de dos dimensiones:

```

struct Vector2D {
    var x = 0.0, y = 0.0
    static func + (izquierdo: Vector2D, derecho: Vector2D) -> Vector2D {
        return Vector2D(x: izquierdo.x + derecho.x, y: izquierdo.y + derecho.y)
    }
}
```

La función operador se define como una función estática con un nombre de función que empareja con el operador a sobrecargar (+). Debido a que la suma aritmética se define como un operador binario, esta función operador toma dos parámetros de entrada de tipo `Vector2D` y devuelve un único valor de salida, también de tipo `Vector2D`.

En esta implementación, llamamos a los parámetros de entrada `izquierdo` y `derecho` para representar las instancias de `Vector2D` que estarán a la izquierda y a la derecha del operador +. Son nombres arbitrarios, lo importante es la posición. El primer parámetro de la función es el que hace de primer operador.

La función devuelve una nueva instancia de `Vector2D`, cuyas propiedades `x` e `y` se inicializan con la suma de las propiedades `x` e `y` de las instancias de `Vector2D` que se están sumando.

La función se define globalmente, más que como un método en la estructura `Vector2D`, para que pueda usarse como un operador infijo entre instancias existentes de `Vector2D`:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let otroVector = Vector2D(x: 2.0, y: 4.0)
let vectorSuma = vector + otroVector
// vectorSuma es una instancia de Vector2D con valores de (5.0, 5.0)
```

## Operadores prefijos y postfijos

El ejemplo anterior demuestra una implementación propia de un operador binario infijo. Las clases y las estructuras pueden también proporcionar implementaciones de los operadores unarios estándar. Los operadores unarios operan sobre un único objetivo. Son prefijos se preceden el objetivo (como `-a`) y postfijos si siguen su objetivo (como en `b!`).

Para implementar un operador unario prefijo o postfijo se debe escribir el modificador `prefix` o `postfix` antes de la palabra clave `func` en la declaración de la función operador:

```
struct Vector2D {
    ...
    static prefix func - (vector: Vector2D) -> Vector2D {
        return Vector2D(x: -vector.x, y: -vector.y)
    }
}
```

El ejemplo anterior implementa el operador unario negación (`-a`) para instancias de `Vector2D`.

Por ejemplo:

```
let positivo = Vector2D(x: 3.0, y: 4.0)
let negativo = -positivo
// negativo es una instancia de Vector2D con valores de (-3.0, -4.0)
let tambienPositivo = -negativo
// tambienPositivo es una instancia de Vector2D con valores de (3.0, 4.0)
```

## Operadores de equivalencia

Como ya hemos visto, las clases no tienen una implementación por defecto de los operadores "igual a" (`==`) y "no igual a" (`!=`), pero las estructuras sí (a partir de Swift 5). Para comparar dos instancias de una clase debemos proporcionar una implementación del operador `==`.

A partir de Swift 5 si declaramos que una estructura cumple el protocolo `Equatable` el compilador de Swift implementará una comparación por defecto, siempre que todas las propiedades de la estructura sean a su vez `Equatable`.

En Swift 4 para implementar la igualdad en la estructura `Vector2D` tenemos que definir una extensión que cumple el protocolo `Equatable`:

```
// Swift 4
extension Vector2D: Equatable {
    static func == (izquierdo: Vector2D, derecho: Vector2D) -> Bool {
        return (izquierdo.x == derecho.x) && (izquierdo.y == derecho.y)
    }
}
```

En el ejemplo anterior se implementa un operador "igual a" (`==`) que comprueba si dos instancias de `Vector2D` tienen valores equivalentes. En el contexto del `Vector2D` tiene sentido considerar "igual" como "ambas instancias tienen los mismos valores x e y", por lo que esta es la lógica usada por la implementación.

La implementación del operador "no igual a" (`!=`) se realiza por defecto como una negación del anterior.

En Swift 5 basta con declarar con la extensión que el `Vector2D` cumple el protocolo:

```
// Swift 5
extension Vector2D: Equatable { }
```

Una vez definido el operador `==` podemos comparar dos instancias de `Vector2D`:

```
let dosTres = Vector2D(x: 2.0, y: 3.0)
let otroDosTres = Vector2D(x: 2.0, y: 3.0)
let unoDos = Vector2D(x: 1.0, y: 2.0)
if dosTres == otroDosTres {
    print("Los vectores \(dosTres) y \(otroDosTres) son equivalentes.")
}
if (unoDos != dosTres) {
    print("Los vectores \(unoDos) y \(dosTres) son distintos.")
}
```

## Genéricos

Veamos cómo podemos utilizar los genéricos con clases y estructuras.

Vamos a utilizar como ejemplo un tipo de dato muy sencillo: una pila (*stack*) en la que se podrán añadir (*push*) y retirar (*pop*) elementos.

La versión no genérica del tipo de dato es la siguiente, en la que se implementa una pila de enteros.

```
struct IntStack {
    var items = [Int]()
    mutating func push(_ item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

```

    }
}

```

La estructura usa un array para guardar los ítems y los métodos `push` y `pop` añaden y retiran los elementos.

El problema de esta estructura es su falta de genericidad; sólo puede almacenar enteros.

Aquí está una versión genérica del mismo código:

```

struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}

```

El parámetro del tipo `Element` define un tipo genérico que se utiliza como *placeholder* del tipo real del que se declare la estructura. Podemos ver que se utiliza en la definición de los distintos elementos de la estructura.

Por ejemplo, el array de ítems es un array de `Elements`. Y los ítems añadidos y retirados de la pila son también objetos de tipo `Element`.

Por ejemplo, podemos crear una pila de cadenas:

```

var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// la pila contiene ahora 4 cadenas

```

Y podemos retirar la última cadena de la pila:

```

let fromTheTop = stackOfStrings.pop()

```

## Extensión de un tipo genérico

Cuando se extiende un tipo genérico, no hace falta añadir el parámetro del tipo entre `<>`. El tipo genérico está disponible a partir de la definición original y se puede usar tal cual en el cuerpo de la extensión.

Por ejemplo, podemos extender el tipo genérico `Stack` para añadir una propiedad computable de sólo lectura que devuelva el tope de la pila:

```
extension Stack {
    var topItem: Element? {
        return items.isEmpty ? nil : items[items.count - 1]
    }
}
```

En la extensión se utiliza el nombre genérico `Element` sin tener que declararlo después de `Stack`, porque está definido en la estructura original.

Ahora ya se puede acceder al tope de la pila sin retirar el elemento:

```
if let topItem = stackOfStrings.topItem {
    print("El ítem en el tope de la pila es \(topItem).")
}
// Imprime "El ítem en el tope de la pila es tres."
```

## Restricción en las extensiones de un protocolo

En una extensión de un protocolo es posible definir una restricción indicando una condición que se debe cumplir para que la extensión se pueda aplicar. Los métodos y propiedades de la extensión sólo estarán disponibles en aquellos tipos que se ajusten al protocolo y cumplan el requisito.

El requisito se definen usando una cláusula genérica `where` en la que se indica una condición sobre algún *tipo asociado* definido en el protocolo.

!!!Note "Nota" No hemos visto el concepto de *tipo asociado* en un protocolo. Es una especie de tipo genérico que se define en el protocolo y que se convierte en un tipo concreto en la clase que se ajusta al protocolo.

Por ejemplo, podemos definir una extensión al protocolo `Collection` que se aplique a cualquier colección cuyos elementos cumplen el protocolo `Equatable`. De esta forma nos aseguramos de que los operadores `==` y `!=` están definidos y podemos usarlos en la extensión:

```
extension Collection where Element: Equatable {
    func allEqual() -> Bool {
        for element in self {
            if element != self.first {
                return false
            }
        }
        return true
    }
}
```

El nombre `Element` es un nombre definido en el protocolo `Collection` que se refiere al tipo de los elementos de la colección. Es un *tipo asociado*.

El método `allEqual()` devuelve `true` si y sólo si todos los elementos en la colección son iguales.

Un ejemplo:

```
let numerosIguales = [100, 100, 100, 100, 100]
let numerosDiferentes = [100, 100, 200, 100, 200]
```

Dado que ambos arrays cumplen el protocolo `Collection` y los enteros cumplen el protocolo `Equatable` podemos usar el método `allEqual()`:

```
print(numerosIguales.allEqual())
// Prints "true"
print(numerosDiferentes.allEqual())
// Prints "false"
```

En una colección cuyos elementos no cumplen el protocolo `Equatable` no se puede utilizar el método de la extensión:

```
let tormenta = Persona(edad: 32, nombreCompleto: "Ororo Munroe")
let superHeroes = [tormenta, peterParker, reedRichards]
print(superHeroes.allEqual())
//error: type 'Persona' does not conform to protocol 'Equatable'
```

## Bibliografía

- Swift Language Guide
  - [Classes and Structures](#)
  - [Properties](#)
  - [Methods](#)
  - [Inheritance](#)
  - [Initialization](#)
  - [Protocolos](#)
  - [Casting de tipos](#)
  - [Extensiones](#)
  - [Funciones operador](#)
  - [Genéricos](#)

|                               |     |
|-------------------------------|-----|
| 2003 - Septiembre             | 3   |
| 2004 - Diciembre              | 6   |
| 2004 - Junio                  | 8   |
| 2004 - Septiembre             | 13  |
| 2005 - Diciembre              | 16  |
| 2005 - Junio                  | 18  |
| 2005 - Septiembre             | 22  |
| 2006 - Junio                  | 25  |
| 2006 - Septiembre (Enunciado) | 29  |
| 2006 - Septiembre (Solucion)  | 33  |
| 2007 - Junio (Enunciado)      | 36  |
| 2007 - Junio (Solucion)       | 41  |
| 2007 - Septiembre (Enunciado) | 43  |
| 2007 - Septiembre (Solucion)  | 47  |
| 2008 - Junio                  | 51  |
| 2008 - Septiembre             | 54  |
| 2011 - Junio                  | 57  |
| examen-diciembre2003          | 61  |
| examen-diciembre2004          | 63  |
| examen-diciembre2005          | 65  |
| examen-junio2002              | 67  |
| examen-junio2003              | 69  |
| examen-junio2004              | 73  |
| examen-junio2005              | 78  |
| examen-junio2007              | 82  |
| ExamenLPPfinalJunio2011       | 87  |
| examen-septiembre2003         | 91  |
| examen-septiembre2004         | 94  |
| examen-septiembre2005         | 97  |
| JulioFinalLPP2015-16.pdf (1)  | 100 |

|                                           |     |
|-------------------------------------------|-----|
| JulioLPP2013                              | 112 |
| soluc-septiembre                          | 125 |
| 2002 - Junio                              | 131 |
| 2003 - Diciembre                          | 133 |
| 2003 - Junio                              | 135 |
| Recuperación Julio 2013                   | 139 |
| solucion_1                                | 152 |
| Examen_Julio_LPP_2016-17_con_solucion (1) | 157 |
| ExamenLPPjulio2011-2012                   | 170 |

# **Lenguajes y Paradigmas de Programación**

## **Curso 2002-2003**

### **Examen de la Convocatoria de Septiembre**

#### **Normas importantes**

- La puntuación total del examen son 42 puntos que sumados a los 18 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener más de 18 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- **Las notas estarán disponibles el viernes 12 y las revisiones se realizarán el martes 16 de septiembre de 10:00 a 12:00.**

#### **Pregunta 1 (6 puntos)**

Define un procedimiento llamado set-double! que reciba un árbol numérico de cualquier profundidad como parámetro y reemplace cada hoja por el doble de cada número. El método debe recorrer cada elemento del árbol.

```
>(define arbol (list 2 (list 3 4) (list 2 3 (list 3 3))))  
>arbol  
>(2 (3 4) (2 3 (3 3)))  
>(set-double! arbol)  
>arbol  
>(4 (6 8) (4 6 (6 6)))
```

#### **Pregunta 2 (6 puntos)**

Representa mediante diagramas box & pointer el resultado de evaluar las siguientes expresiones:

a.- (cadadr '(a (b (cd ef) d)))  
b.- (define lista (list (cons (append '(a) '(b)) '(c))))  
c.- (set-car! (cdar lista) (caar lista))

#### **Pregunta 3 (6 puntos)**

Escribe una función multicompose que tome como argumento una lista de cualquier longitud, cuyos elementos son funciones de un argumento. Debería devolver una función de un argumento que realiza la composición de todas las funciones dadas.

Nota: No usar mutación (set!)

```
>((multicompose (list sqrt sqrt 1+)) 80)  
3  
  
>((multicompose (list first bf)) '(la sinuosa carretera))  
sinuosa
```

## Pregunta 4 (6 puntos)

Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3 PM es la hora 15.

La implementación se hará usando dos representaciones internas diferentes.

- El constructor para las horas del día es `time`. Toma dos argumentos: las horas y los minutos. 4:12 debería ser (`time 4 12`)

- También queremos los siguientes tres operadores:

`(hour t) → devuelve la parte de hora de un time dado`

`(minute t) → devuelve la parte de minutos de un time dado`

`(hour? t) → devuelve #t si el time t es una hora exacta (como 6:00); devuelve #f en cualquier otro caso`

a/ Implementa `time`, `hour`, `minute` y `hour?` usando una representación interna en la que la hora se representa como una lista de dos números. Esto es, 4:12 debería representarse internamente como `(4 12)`

b/ Implementa `time`, `hour`, `minute` y `hour?` usando una representación interna en la que la hora se representa como un entero, a saber, el número de minutos transcurridos desde medianoche. Esto es, 4:12 debería representarse internamente como el número 252 ( $(4 \times 60) + 12$ )

## Pregunta 5 (6 puntos)

Define un procedimiento llamado (sufijo wd char) que admita como argumento una palabra wd y un carácter char y que devuelva el resto de la palabra wd desde que se encuentra el carácter char. En el caso en que la palabra no contenga a char, entonces la función devuelve la cadena vacía.

Indica si el procedimiento que has escrito es recursivo o iterativo y explica por qué.

```
> (sufijo 'hola 'o)
'la
> (sufijo 'resto 'e)
'sto
> (sufijo 'pepita 's)
""
```

### **Pregunta 6 (6 puntos)**

Explica qué valor devuelven las siguientes expresiones:

a/

```
(define foo (lambda (x)
  (if (equal? x 0)
      1
      (* x (foo (- x 1))))))

(foo 3)
```

b/

```
(define (foo3 x)
  (lambda (y) (+ y x)))

(define foo4 (foo3 1))
foo4
```

c/

```
(let ((x 3) (y 4))
  (let ((x 5) (y (+ x 3)))
    (let ((x (+ x 7)))
      (+ x y))))
```

### **Pregunta 7 (6 puntos)**

Explica la evaluación de las siguientes expresiones y dibuja el entorno resultante.

```
(define (foo y)
  (lambda (x)
    (set! y (+ x y))))

(define w (foo 5))
(w 30)
```

# Lenguajes y Paradigmas de Programación

## Curso 2003-2004

### Examen de la Convocatoria de Diciembre

#### Normas importantes

- La puntuación total del examen es de 40 puntos que sumados a los 20 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 16 puntos en este examen.**
- La duración del examen es de 2 horas.

#### Pregunta 1 (8 puntos)

(6 puntos) Escribe un procedimiento llamado `power-close-to` que tome dos enteros positivos (`b` y `n`) como argumento y devuelva la potencia más pequeña de `b` que es mayor que `n`. Esto es, debería devolver el entero más pequeño `i` tal que  $b^i > n$ .

`(power-close-to 2 100) -> 8`

(2 puntos) ¿El procedimiento que has escrito es recursivo o iterativo?

#### Pregunta 2 (8 puntos)

Suponiendo que `tree` es un árbol cuyos datos son números, define el procedimiento (`suma tree`) que devuelva la suma de todos los datos del árbol `tree`. Debes usar los selectores definidos en clase:

```
(define (make-tree dato hijos) (cons dato hijos))
(define (dato tree) (car tree))
(define (hijos tree) (cdr tree))
```

Por ejemplo, `(suma '(1 (2) (3 (4 (5)))) )` debe devolver 15

#### Pregunta 3 (8 puntos)

oHe aquí un procedimiento para construir un objeto abstracto persona. El procedimiento está incompleto.

```
(define (make-persona nombre edad profesion dni)
  (let ((persona (list nombre edad profesion dni)))
    (lambda (m)
      (cond ((eq? m 'nombre)      <a>)
            ((eq? m 'e-mail)     <b>)
            ((eq? m 'empresa)    <c>)
            ((eq? m 'tfno)       <d>)
            (else (error "petición desconocida" m))))))
```

**(continúa detrás)**

Un ejemplo de uso del procedimiento anterior serían las siguientes instrucciones:

```
(define persona-1
  (make-persona 'lucia 'lgl@yahoo.com 'ua '966454321))
(define persona-2
  (make-persona 'david 'dvg@hotmail.com 'dccia '953441234))
```

1. (4 puntos) Escribe qué sentencias Scheme deberían ir en los lugares marcados con <a>, <b>, <c>, <d>.
2. (4 puntos) ¿Cómo se preguntaría por el dni de una persona? Pon un ejemplo con el objeto persona-2.

#### Pregunta 4 (8 puntos)

Consideramos que las siguientes expresiones se evalúan en el orden que aparecen:

```
(define a (list (list 'q) 'r 's))
(define b (list (list 'q) 'r 's))
(define c a)
(define d (cons 'p a))
(define e (list 'p (list 'q) 'r 's))
```

1. (4 puntos) Dibuja un diagrama box-and-pointer con las estructuras resultantes
2. (4 puntos) Indica cuál sería el resultado de las siguientes expresiones:

|                        |                           |
|------------------------|---------------------------|
| (eq? a c)              | (equal? a c)              |
| (eq? a b)              | (equal? a b)              |
| (eq? a (cdr d))        | (equal? a (cdr d))        |
| (eq? (car a) (cadr e)) | (equal? (car a) (cadr e)) |

#### Pregunta 5 (8 puntos)

Supongamos los siguientes programas Scheme.

|                                                                                |                                                                                    |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| (define z 3)<br>(define h (let ((x 2))<br>(lambda (y)<br>(+ x y z))))<br>(h 4) | (define h<br>(let ((x 2))<br>(lambda (y)<br>(let ((z 3))<br>(+ x y z))))<br>(h 4)) |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|

1. (4 puntos) Dibuja los entornos resultantes después de evaluarse el programa 1 y el 2 (ambos se evalúan por separado)
2. (1 punto) ¿Cuál es el valor que devuelve la última expresión en ambos casos?
3. (3 puntos) Dos entornos son equivalentes cuando todos los programas Scheme dan el mismo resultado en ambos. ¿Son equivalentes los entornos resultantes de evaluar el programa 1 y el 2? Si piensas que los entornos no son equivalentes, encuentra algún programa Scheme que se comporte de forma distinta (devuelva un valor distinto) en uno y otro entorno.

# Lenguajes y Paradigmas de Programación

## Curso 2003-2004

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 40 puntos que sumados a los 20 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 16 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.

#### Pregunta 1 (5 puntos)

Queremos un procedimiento (`haz-palindromo wd`) que tome una palabra `wd` y que devuelva un palíndromo formando a partir de la palabra `wd`. Escribe el procedimiento e indica si es recursivo o iterativo.

Ejemplos:

```
(haz-palindromo "hola") -> "holaaloh"  
(haz-palindromo "arroz")-> "arrozzorra"  
(haz-palindromo "") -> ""
```

#### Pregunta 2 (5 puntos)

Rellena los huecos en las siguientes expresiones de Scheme para que su evaluación devuelva los resultados indicados. No es necesario que en los huecos haya un único dato, puede haber una expresión compuesta.

a)

```
(define g  
  (lambda (_____)  
    (lambda (_____ b) (_____ b)))  
  
  ((g 3) - 5)  
-2
```

b)

```
(define object
  (let ((init 0))
    (_____ (new)
      (let ((temp init))
        (_____ temp)))))

(object 7)
0
(object 1)
7
(object 4)
1
```

c) Indica cuál es el resultado de evaluar las siguientes expresiones de Scheme.

```
(define (echo f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x) (f ((echo f (- n 1)) x))))))

((echo (lambda (x) (* x x)) 2) 2)
```

---

### Pregunta 3 (6 puntos)

Escribe un procedimiento (producto-diagonal matriz) que calcule el producto de la diagonal principal de una matriz cuadrada matriz. Suponemos que la matriz cuadrada se representa como una lista de listas planas en la que cada lista plana representa una fila de la matriz. Puedes definir procedimientos auxiliares que uses en el procedimiento principal.

Ejemplos:

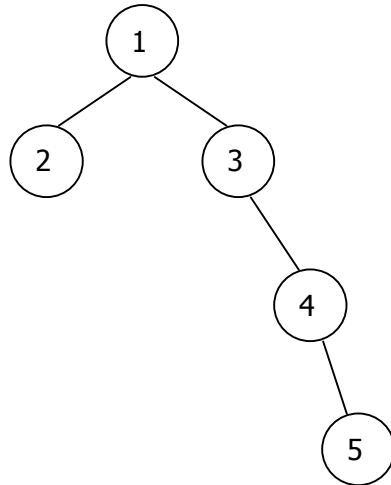
```
(producto-diagonal '((2 3)
                     (4 5)))
10

(producto-diagonal '((2 4 6)
                     (8 10 12)
                     (14 16 18)))
360
```

#### Pregunta 4 (6 puntos)

Define el procedimiento (`max-hijos tree`) que vaya recorriendo los nodos del árbol `tree`, calculando el número de hijos de cada nodo y que devuelva el máximo número de hijos encontrado en un único nodo.

Por ejemplo, el árbol `'(1 (2) (3 (4 (5))))'` se corresponde con la siguiente figura:



En este árbol tenemos los siguientes números de hijos:

Nodo '1' tiene 2 hijos

Nodo '2' tiene 0 hijos

Nodo '3' tiene 1 hijo

Nodo '4' tiene 1 hijo

Nodo '5' tiene 0 hijos

Por lo tanto, el máximo número de hijos corresponde al nodo '1', con 2 hijos. Éste número es el que debe devolverse.

Ejemplos:

```
(max-hijos '(1 (2) (3) (4)))
3
(max-hijos '(1 (2) (3 (4 (5))))))
2
```

#### Pregunta 5 (6 puntos)

Queremos definir con la extensión de Programación Orientada a Objetos de Scheme una jerarquía de clases que nos permita mantener la información sobre el reparto de paquetes de una empresa de mensajería a contra-reembolso.

Definimos una clase llamada **Paquete** que tendrá como variables de instanciacción el **nombre del mensajero** que se encargará de su reparto y el **precio del objeto** a entregar.

Además se guardará la información del estado en que se encuentra un paquete. Existirán tres posibles valores:

- 0: perfecto
- 1: leve deterioro
- 2: deteriorado

Por último, existen dos tipos distintos de paquetes: **Sobre** y **Caja**. El transporte de un sobre tiene un **coste de transporte** de 2 euros y el de una caja de 6 euros siempre que el paquete se entregue en estado perfecto.

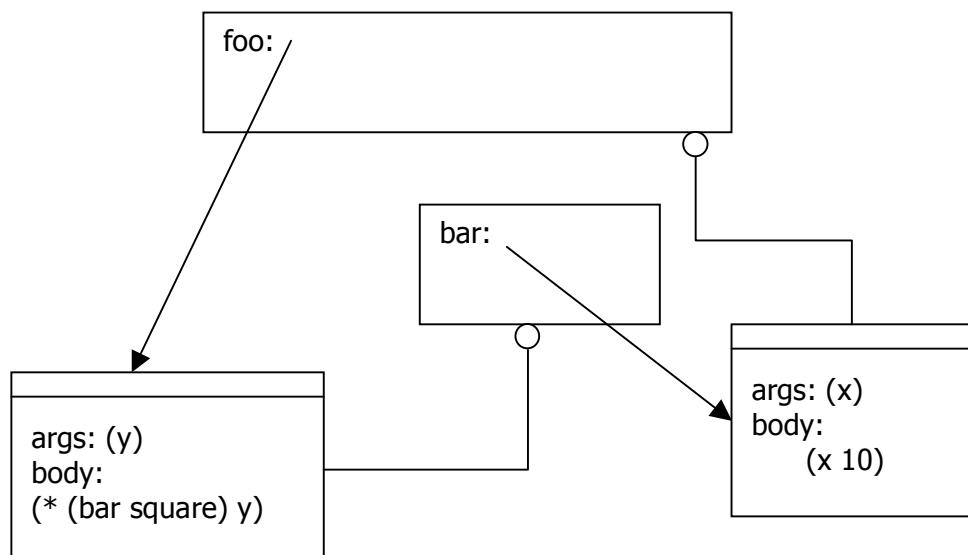
Necesitamos los siguientes métodos:

- num-paquetes, que devuelve el número de paquetes creados
- num-sobres, que devuelve el número de sobres
- num-cajas, que devuelve el número de cajas
- cambia-mensajero, que permite cambiar el mensajero asignado a un paquete
- cambia-estado, que permite cambiar el estado de un paquete. Cuando un paquete está deteriorado ese será su estado definitivo y el estado no se podrá modificar.
- importe, que devuelve el precio a cobrar por la entrega de un paquete. El precio a cobrar es el coste de transporte más el precio del objeto, si el estado es 0 (perfecto). Si el estado es 1, el coste de transporte se reduce en un 50% y si el estado es 2 el coste de transporte es 0 y el importe a cobrar es el precio del objeto.

Usando la extensión de POO de Scheme, implementa un conjunto de clases que resuelvan este problema y escribe un ejemplo de uso.

### Pregunta 6 (6 puntos)

Escribe una única instrucción en Scheme, que no use mutación, que genere el siguiente modelo de entorno:



### Pregunta 7 (6 puntos)

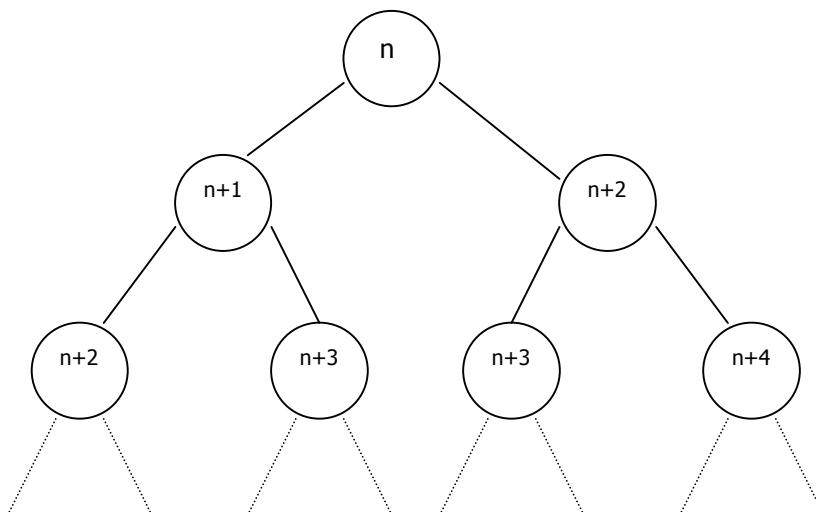
Un árbol binario infinito es una estructura que contiene un dato, un hijo a la izquierda y un hijo a la derecha, los cuales son a su vez árboles binarios infinitos. Es decir, un árbol binario infinito es un árbol cuyos nodos tienen dos hijos y un profundidad infinita (infinitos niveles).

- a) Rellena los huecos para completar la definición del árbol binario infinito y de sus selectores. Supongamos que la llamada a `def-macro` en la siguiente expresión permite definir una nueva macro de Scheme. En este caso es necesario definir `make-inf-tree` como una macro y no como una función porque, de forma similar a lo que sucede con los streams, *no queremos que se evalúen los argumentos* cuando se haga la llamada a `make-inf-tree` para construir el árbol infinito.

```
(def-macro (make-inf-tree dato izq der)
           (list dato _____))

(define (dato tree) (_____))
(define (hijo-izq tree) (_____))
(define (hijo-der tree) (_____))
```

- b) Utilizando la estructura de árbol binario infinito, define el procedimiento `(make-n-tree n)` que reciba un número `n` como parámetro y devuelva el siguiente árbol binario infinito:



# **Lenguajes y Paradigmas de Programación**

## **Curso 2003-2004**

### **Examen de la Convocatoria de Septiembre**

#### **Normas importantes**

- La puntuación total del examen es de 40 puntos que sumados a los 20 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 16 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 2 horas.

#### **Pregunta 1 (8 puntos)**

Escribe una función (palabras-repetidas frase1 frase2) que tome dos frases y devuelva el número de palabras repetidas en ellas. En el caso de que una palabra esté repetida en la misma frase se debe contar en las múltiples ocasiones que aparece

Ejemplos:

```
(palabras-repetidas '(me llamo juan) '(juan es alto))  
1  
(palabras-repetidas '(hola que me me dices) '(me))  
2  
(palabras-repetidas '(hola hola hola adios) '(hola hola))  
6
```

#### **Pregunta 2 (8 puntos)**

Escribe una función denominada (perfecto? numero) que reciba como parámetro un número entero y devuelva true si y sólo si el número es perfecto. Un número es perfecto si es igual a la suma de sus divisores inferiores a él. Por ejemplo, el número 28 es perfecto porque  $28 = 1+7+4+1$  y los divisores de 28 son (exceptuando el propio 28) 14, 7, 4 y 1.

Intenta hacer el ejercicio lo más modular posible, definiendo en su caso funciones auxiliares.

### Pregunta 3 (8 puntos)

Suponemos árboles binarios definidos con la siguiente interfaz

```
(define (make-tree dato izq der)
      (list dato izq der))

(define (dato tree) (car tree))
(define (hijo-izq tree) (car (cdr tree)))
(define (hijo-der tree) (car (cdr (cdr tree))))
```

Define el procedimiento (swap-hijos! tree) que intercambie los hijos del árbol binario pasado como parámetro. El procedimiento debe modificar el árbol mutando sus hijos.

```
(define tree '(1 (2 (3) (4)) (4 (3) (4))))
(swap-hijos! tree)
tree
(1 (4 (3) (4)) (2 (3) (4)))
```

### Pregunta 4 (8 puntos)

Dibuja y explica el modelo de entorno resultante de las siguientes instrucciones Scheme. Numera los entornos en el orden en que se van creando.

```
(define (mystery)
  (let ((baz 1000))
    (define (dispatch msg)
      (cond
        ((eq? msg 'one)
         (begin
           (set! baz (/ baz 2))
           baz))
        ((eq? msg 'two)
         (let ((m (dispatch 'one)))
           (set! baz (* m 4))
           baz)))
        (else
         dispatch)))
  (x 'two))
```

**(EL EXAMEN CONTINÚA DETRÁS)**

### Pregunta 5 (8 puntos)

- a) Define un procedimiento (`scale-stream s f`) que recibe un stream de números `s` y devuelve otro stream con los números de `s` escalados por el factor `f`

```
(display-stream (scale-stream pairs 3))  
6  
12  
18  
24  
30  
...
```

- b) Necesitamos generar un stream de potencias de `x`. Define el procedimiento (`powers x`) para que genere dicho stream. **Pista:** puedes usar el procedimiento anterior `scale-stream`.

```
(display-stream (powers 2))  
1  
2  
4  
8  
16  
32  
...
```

# Lenguajes y Paradigmas de Programación

## Curso 2005-2006

### Examen de la Convocatoria de Diciembre

#### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 2 horas.
- Las fechas de revisión y las notas estarán disponibles en la web de la asignatura el próximo día 16 de diciembre.

#### Pregunta 1 (10 puntos)

Define un procedimiento (`mcd n1 n2`) que calcule el máximo común divisor de dos números:

```
(mcd 60 40)  -> 20
(mcd 5 15)   -> 5
```

#### Pregunta 2 (10 puntos)

Define un procedimiento (`mismo-dato? tree`) que tome un árbol como argumento y devuelva `#t` si existe algún nodo en el árbol que tenga el mismo dato que uno de sus hijos directos (no nieto ni sobrino) y `#f` si no existe.

```
(define t1 (make-tree 2 nil))
(define t2 (make-tree 3 nil))
(define t3 (make-tree 2 (list t1 t2)))
(define t4 (make-tree 1 (list t3)))
(mismo-dato? t4) -> #t

(define t5 (make-tree 1 nil))
(define t6 (make-tree 2 (list t2 t5)))
(define t7 (make-tree 1 (list t6)))
(mismo-dato? t7) -> #f
```

### **Pregunta 3 (10 puntos)**

Define un conjunto de funciones que permitan trabajar con una agenda de contactos. Ten especial cuidado en hacer un código modular, mantenable y reusable.

1. (3 puntos) Diseña las funciones.
2. (4 puntos) Implementalas en Scheme puro (no usar las macros de POO) usando alguna de las técnicas de abstracción de datos vistas en clase.
3. (3 puntos) Explica por qué consideras que las funciones que has desarrollado son modulares, mantenibles y reusables.

### **Pregunta 4 (10 puntos)**

Considera la siguiente función:

```
(define (foo a b c)
  (lambda (m)
    (if (a m) (b m) (c m))))
```

1. (2 puntos) Escribe un ejemplo de llamada a foo en el que el resultado de su evaluación sea 5.
2. (4 puntos) Explica el proceso de evaluación de la llamada anterior usando el **modelo de evaluación de sustitución**.
3. (4 puntos) Explica el proceso de evaluación de la llamada anterior usando el **modelo de evaluación basado en entornos**.

# Lenguajes y Paradigmas de Programación

## Curso 2004-2005

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas estarán disponibles en la web de la asignatura el próximo día 1 de Julio.

#### Pregunta 1 (8 puntos)

a) (5 puntos)

Supongamos el siguiente procedimiento:

```
(define (h n)
  (lambda (x y)
    (set! x (+ (car n) (cdr n) y))
    (set-car! n x)
    (set-cdr! n y)
    x))
```

Escribe un ejemplo completo (en el que se llegue a evaluar el cuerpo del lambda) de uso del procedimiento, incluyendo cuáles serían los resultados devueltos por el intérprete.

b) (3 puntos)

Supongamos las siguientes expresiones:

```
(h (g 3) 5)  ->  8
(h (g 12) 5)  -> 17
```

Completa las siguientes definiciones de forma que la llamada (g x) devuelva un procedimiento:

```
(define (g x)
  _____)
```

```
(define (h f z)
  _____)
```

## Pregunta 2 (8 puntos)

Define un procedimiento (calculadora expr) que tome una lista con la estructura (número op número op... número), donde - y / son las únicas operaciones permitidas, y calcule el resultado de la operación. Ten en cuenta que la división tiene mayor precedencia que la resta. Por ejemplo, (10 - 9 / 3) se evalúa como 10 - (9 / 3) y no como (10 - 9) / 3. Se asume que la lista nunca estará vacía.

Ejemplos:

```
(calculadora '(4 - 2)) -> 2  
(calculadora '(10 - 20 / 2 / 5)) -> 8
```

## Pregunta 3 (8 puntos)

Hemos definido la siguientes clases utilizando la extensión de Scheme para POO:

```
(define-class (electrodomestico marca precio)  
  (class-vars (cantidad 0))  
  (initialize (set! cantidad (1+ cantidad)))  
  (method (apagar) ...)  
)  
  
(define-class (vitroceramica marca precio)  
  (parent (electrodomestico marca precio))  
  (method (cambiarPrecio p) (set! precio p))  
  (method (enciendeFogon numeroFuego intensidad) ....)  
  (method (apagar tiempo) ...)  
 ;este método apaga el aparato con un retardo de tiempo minutos  
 ...  
)
```

a) (3 puntos) Supongamos que se evalúan las siguientes expresiones:

```
(define vitro (instantiate vitroceramica 'aeg 200))  
(ask vitro 'cambiarPrecio 500)
```

Se desea obtener el precio original (200) de *vitro*. ¿Cuáles serían las respuestas correctas?

- a) Definiendo un método que devuelva la variable de instanciacción *precio*
- b) Almacenando el precio original en una variable de instancia que posteriormente devolveremos
- c) Definiendo el método: (method (precio-original) (usual 'precio))
- d) Ninguna de las anteriores

b) (3 puntos) Indica cuales de las siguientes afirmaciones son ciertas y cuales falsas:

- a) Cada instancia de una clase tiene su propia versión de las variables de instanciacción
- b) Las variables de instancia se definen como argumentos cuando la instancia se crea
- c) Cuando se envía un mensaje a un objeto para el que no hay definido un

método ejecuta el del hijo si éste existiera

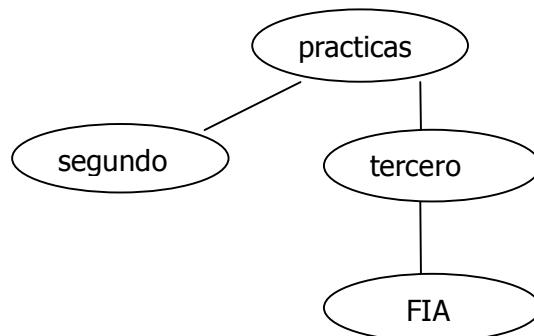
d) La cláusula *initialize* permite modificar las variables de instancia cada vez que se crea una nueva instancia

c) (2 puntos) Se desea agregar un método a la clase `vitrocerámica` para que se apague instantáneamente. Ya dispone de un método con un retardo de x minutos. ¿Qué opciones serían las correctas?

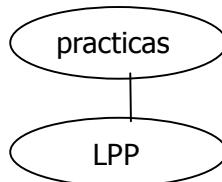
- a) (method (apagateYa) (ask self 'apagar 0))
- b) (method (apagateYa) (ask self 'apagar))
- c) (method (apagateYa) (usual 'apagar))
- d) Las respuestas a) y c) son correctas.

#### Pregunta 4 (9 puntos)

Vamos a representar estructuras de archivos mediante árboles genéricos. Por ejemplo:



a) (5 puntos) Para construir esta estructura, definimos la función `make-rama` que toma una lista con una ruta como argumento y devuelve el árbol que la representa. Por ejemplo, la llamada `(make-rama '(prácticas LPP))` debería devolver el árbol siguiente:



El procedimiento `make-rama` podría definirse como:

```
(define (make-rama ruta)
1 (if (null? ruta)
2   '()
3   (make-tree      ; en esta implementación, make-tree = cons
4         (car ruta)
5         (make-rama (cdr ruta))))
```

Una llamada a (make-rama `(*practicas* segundo LPP)) debería devolver (*practicas* (segundo (LPP))), sin embargo devuelve (*practicas* segundo LPP). Haz los cambios necesarios para que el procedimiento funcione correctamente. Puede que no sea necesario que cambies todas las líneas.

Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_

b) (4 puntos) Define una función (find-ruta fichero) que devuelva una lista plana con la ruta en la que se encuentra el fichero. Por ejemplo:

```
(find-ruta 'FIA) -> '(practicas tercero FIA)
(find-ruta 'tercero) -> '(practicas tercero)
(find-ruta 'GRAFICOS) -> '()
```

### Pregunta 5 (8 puntos)

Define el procedimiento (intercalar! 11 12) que reciba dos listas como argumento y, utilizando mutadores, modifique la lista original 11 e intercale los elementos de ambas. Obviamente, la lista 12 también quedará modificada.

Ejemplo:

```
(define 11 '(1 2 3 4 5))
(define 12 '(a b c d e))
(intercalar! 11 12)
11 -> (1 a 2 b 3 c 4 d 5 e)
12 -> (a 2 b 3 c 4 d 5 e)
```

### Pregunta 6 (9 puntos)

Supongamos las siguientes expresiones:

```
(define (h z)
  (let ((x z) (y (* 2 z)))
    (lambda (v)
      (set! z v)
      (+ v z x y))))
(define g (h 3))
(g 2)
```

- a) (4 puntos) Dibuja el entorno resultante de evaluar las expresiones anteriores  
b) (5 puntos) Explica paso a paso cómo se han evaluado las expresiones anteriores

# Lenguajes y Paradigmas de Programación

Curso 2004-2005

## Examen de la Convocatoria de Septiembre

### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen**.
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 2 horas y 30 minutos.
- Las notas estarán disponibles en la web de la asignatura el próximo día 1 de Julio.

### Pregunta 1 (10 puntos)

A) **(5 puntos)** Define una función (`cumplen-todos pred list`) que devuelva `#t` o `#f` dependiendo de si todos los elementos de la lista `list` cumplen el predicado `pred`. La lista `list` puede ser un pseudo-arbol y contener otras listas. Por ejemplo:

```
(cumplen-todos par? '(2 4 (6 8) 10)) -> #t
(cumplen-todos impar? '(1 3 5 6)) -> #f
(cumplen-todos impar? '()) -> #t
```

B) **(2 puntos)** La función que has definido en el apartado anterior ¿es iterativa o recursiva? ¿Por qué?

C) **(3 puntos)** Supongamos que uno de los elementos de la lista no cumple el predicado ¿se sigue procesando la lista en tu solución? En el caso de que no, perfecto, **ya puedes sumar los 3 puntos de este apartado**. En el caso de que se siga procesando la lista hasta el final, modifica la función para que no lo haga y devuelva `#f` justo en ese instante.

### Pregunta 2 (10 puntos)

Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3:00 PM es la hora 15:00. El constructor para las horas del día es `make-time`. Toma dos argumentos: las horas y los minutos. `4:12` debería ser `(make-time 4 12)` También queremos los siguientes tres operadores:

```
(hour t) → devuelve la parte de hora de un time dado
(minute t) → devuelve la parte de minutos de un time dado
(hour? t) → devuelve #t si el time t es una hora exacta (como 6:00); devuelve #f en cualquier otro caso
```

Ejemplo:

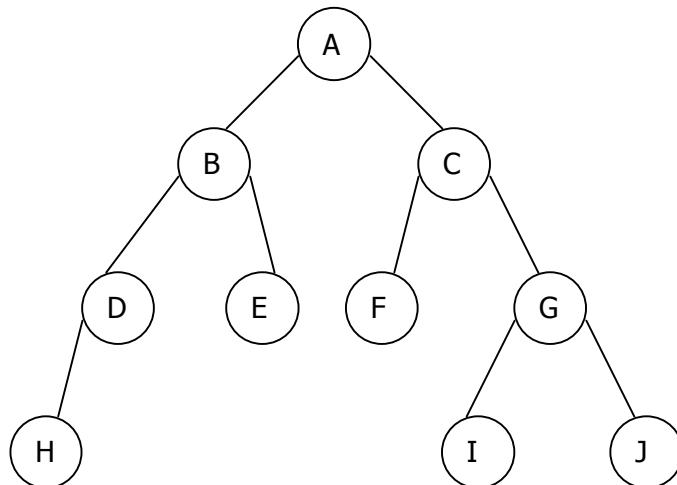
```
> (define t (make-time 4 12))
```

```
> (minute t)
12
> (hour t)
4
> (hour? t)
#f
```

Implementa el constructor `make-time` y los operadores `hour`, `minute` y `hour?` usando la técnica del paso de mensajes, en la que el objeto devuelto por el constructor es una función que admite mensajes.

### Pregunta 3 (10 puntos)

El procedimiento `(prof_hojas tree)` devuelve una lista plana con la profundidad de cada hoja de un árbol binario. Ejemplo:



`(prof_hojas tree)` devolverá (3 2 2 3 3)

Rellena los huecos para que `prof_hojas` funcione correctamente:

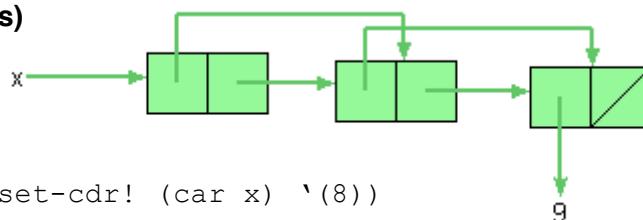
```
(define (prof_hojas tree)
  (define (help tree depth)
    (cond ((null? tree) _____)
          ((leaf? tree) _____)
          (else ( _____
                    (help tree 0))))))
```

### Pregunta 4 (10 puntos)

Para cada apartado:

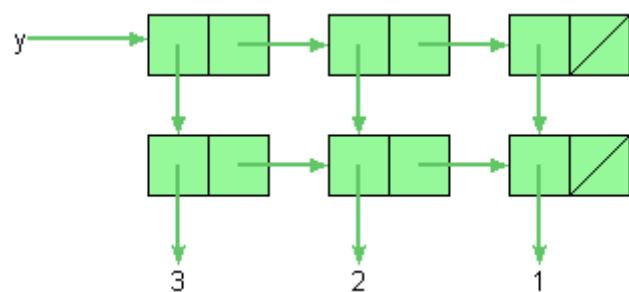
- Escribe una expresión Scheme que construya el diagrama caja y puntero de la figura.
- Dibuja el diagrama caja y puntero resultante de la mutación indicada.

#### A) (5 puntos)



Mutación: `(set-cdr! (car x) '(8))`

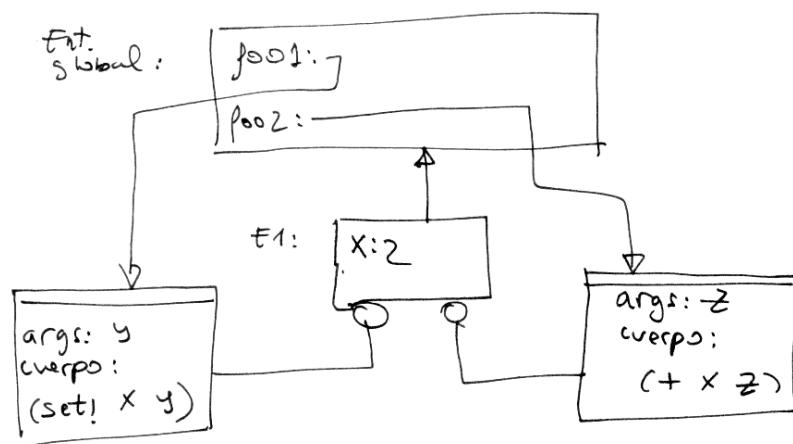
B) (5 puntos)



Mutación: (set-car! (cdr (cadr y)) 4)

Pregunta 5 (10 puntos)

Supongamos el entorno definido por la siguiente figura



A) (5 puntos) Completa el siguiente programa para que genere el entorno anterior.

```
(define c  (_____))
(define foo1 (_____) )
(define foo2 (_____) )
```

Nota: La variable auxiliar  $c$  no aparece en la figura.

B) (4 puntos) Explica paso a paso cómo se han evaluado las expresiones del programa que acabas de completar para generar el entorno de la figura, según el modelo de evaluación de Scheme basado en entornos.

C) (1 punto) Escribe un ejemplo de invocación de `foo1` y `foo2`, indicando el resultado devuelto.

# Lenguajes y Paradigmas de Programación

## Curso 2005-2006

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas estarán disponibles en la web de la asignatura el próximo día 1 de Julio.

#### Pregunta 1 (8 puntos)

Define un procedimiento llamado (transformar plantilla lista) que reciba dos listas como argumento: la lista plantilla estará compuesta por números enteros positivos con una estructura jerárquica, como (2 (3 1) 0 (4)). La segunda lista será una lista plana con tantos elementos como indique el mayor número de plantilla más uno (en el caso anterior serían 5 elementos). El procedimiento deberá devolver una lista donde los elementos de la segunda lista se sitúen (en situación y en estructura) según indique la plantilla.

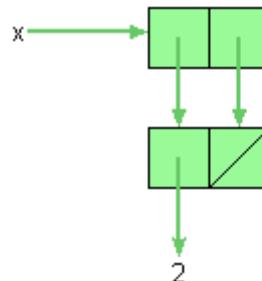
Ejemplos:

```
(transformar '((0 1) 4 (2 3)) '(hola que tal estas hoy))
(hola que) hoy (tal estas)

(transformar '(1 4 3 2 5 (0)) '(vamos todos a aprobar este examen))
(todos este aprobar a examen (vamos))
```

#### Pregunta 2 (8 puntos)

a) (4 puntos) Tenemos el siguiente box-and-pointer:



Para cada una de las siguientes expresiones que lo gene correctas y cuáles no. En caso de que se incorrecta, debes explicar por qué.

| Expresión                                                                                                   | Verdadero / Falso<br>(indicar por qué) |
|-------------------------------------------------------------------------------------------------------------|----------------------------------------|
| (define x `((2) 2))                                                                                         |                                        |
| (define x<br>(let ((temp (list 2)))<br>(cons temp temp)))                                                   |                                        |
| (define x<br>(let ((temp (list 2)))<br>(let ((foo (cons (list 2) temp)))<br>foo)))                          |                                        |
| (define x<br>(let ((temp (list 2)))<br>(let ((foo (cons (list 2) temp)))<br>(set-car! foo temp)<br>foo)))   |                                        |
| (define x<br>(let ((temp (list 2)))<br>(let ((foo (cons (list 2) temp)))<br>(set! (car foo) temp)<br>foo))) |                                        |

b) (4 puntos) El siguiente programa tiene errores. Corrígelos e indica cuáles son y el tipo de error (los errores no son sintácticos ni están en el número de paréntesis)

```

1. (define attach-type cons)
2. (define type car)
3. (define contents cdr)

4. (define (make-euro amt)
5.   (attach-type 'euro amt))

6. (define (make-dolar amt)
7.   (attach-type 'dolar amt))

8. (define (make-libra amt)
9.   (attach-type 'libra amt))

10. (define (+money amt1 amt2)
11.   (make-euro (+ (contents (toeuro amt1))
12.                 (contents (toeuro amt2)))))

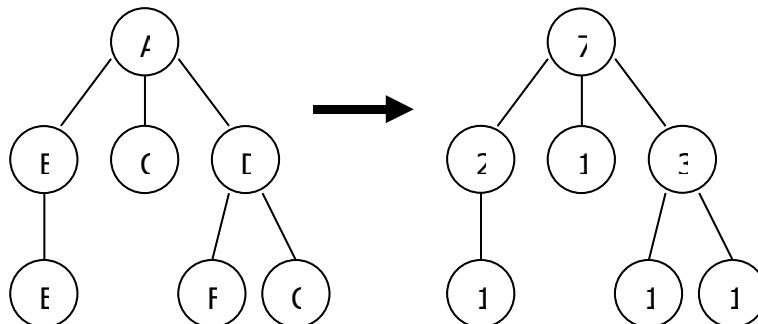
13. (define (toeuro amt)
14.   (* (conversion (car amt)) (cdr amt)))

15. (define (conversion type)
16.   (cond ((eq? type 'euro) 1.0)
17.         ((eq? type 'dolar) 0.7)
18.         ((eq? type 'libra) 1.2)))

```

### Pregunta 3 (9 puntos)

Define un procedimiento llamado (`set-nodes! tree`) que reciba un árbol genérico como argumento y mute el contenido de los nodos del árbol de forma que cada uno contenga el número de nodos del subárbol del que es raíz. Ejemplo:



### Pregunta 4 (9 puntos)

Recuerda la función `map` que toma como argumento una función y una lista y devuelve la lista resultante de aplicar la función a cada elemento de la lista. Veamos una variante, la función `mapfun`, que toma una lista de funciones y un valor y devuelve una lista resultante de aplicar cada una de las funciones al valor. Por ejemplo:

```
(mapfun (list (lambda (x) (+ x 3)) (lambda (x) (+ x 10))) 5) -> (8 15)
```

Supongamos ahora que llamamos `suma-k` a la función de un parámetro que suma el número `k` a ese parámetro. Por ejemplo `suma-5` sería una función que le suma 5 al parámetro que se le pasa:

```
(suma-5 10) -> 15
```

Definimos entonces la función (`sumadores n k`) como la función que devuelve una lista de `n` funciones sumadoras que suman los números `k`, `2k`, ..., `nk` (esto es, la lista de funciones `suma-k`, `suma-2k`, ..., `suma-nk`). Por ejemplo, (`sumadores 4 3`) devuelve una lista con las funciones `suma-3`, `suma-6`, `suma-9` y `suma-12`. De esta forma, si unimos la llamada a `mapfun` con lo que nos devuelve `sumadores`, obtendremos el siguiente ejemplo:

```
(mapfun (sumadores 4 3) 10) -> (13 16 19 22)
```

- a) (4 puntos) Implementa la función `mapfun`
- b) (5 puntos) Implementa la función `sumadores`

### Pregunta 5 (8 puntos)

Supongamos las siguientes expresiones:

```
(define (f y)
  (let ((x 10))
    (lambda (y)
      (set! x (+ x y))
      x)))
(define x 2)
```

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

- a) **(4 puntos)** Dibuja el entorno resultante de evaluar las expresiones anteriores  
b) **(4 puntos)** Explica paso a paso cómo se han evaluado las expresiones anteriores

## Pregunta 6 (8 puntos)

Lee el siguiente problema a programar:

### Explorando las minas de Moria.

Supongamos una aventura en la que un personaje puede moverse por las minas de Moira. Las minas de Moira son un conjunto de estancias comunicadas entre si. Cada estancia tiene un identificador (su nombre). Las estancias se conectan entre si mediante puertas situadas en el norte, sur, este u oeste (una estancia puede tener un número variable de puertas: de una a cuatro). Cada puerta tiene un identificador único.

El personaje podrá moverse de una habitación a otra realizando uno de los siguientes cuatro movimientos: ir-norte, ir-sur, ir-este e ir-oeste. Si la puerta correspondiente existe y está abierta en la estancia actual, el personaje pasará a la estancia situada en esa posición. Si la puerta no existe o está cerrada, el persona permanecerá en la misma habitación.

En las habitaciones puede haber distintos objetos, entre ellos llaves. Todos los objetos tienen un identificador que los distingue. Los identificadores de las llaves coincidirán con un identificador de alguna puerta, en cuyo caso la llave puede abrir la puerta. El personaje puede coger los objetos y guardárselos (el objeto deja entonces de estar en la habitación). Si el objeto es una llave, podrá usarla para abrir y cerrar la puerta correspondiente a su identificador.

Una vez leído el problema, contesta a las siguientes cuestiones:

- a) **(4 puntos)** Diseña un conjunto de **clases, interfaces y relaciones de herencia** con las que especificar el problema. Para cada clase debes definir sus campos y sus métodos, explicando qué hace cada uno de ellos **pero sin implementarlos** (si quieres alguna pista de posibles clases y métodos, sigue leyendo). Puedes dibujar diagramas UML como los vistos en clase (pero explicando además cada componente).
- b) **(2 puntos)** Implementa en la clase **personaje** el método **(abre-puerta dirección)** (siendo **dirección** uno de los valores: **'norte**, **'sur**, **'este**, **'oeste**) que pruebe a abrir la puerta correspondiente de la estancia actual con todas las llaves que lleva el personaje. Debes usar siempre métodos que hayas definido en el apartado anterior.
- c) **(2 puntos)** Implementa en la clase **personaje** el método **(coge-objeto identificador)** que realiza la acción de coger de la estancia actual el objeto especificado por el identificador.

# Lenguajes y Paradigmas de Programación

## Curso 2005-2006

### Examen de la Convocatoria de Septiembre

#### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas estarán disponibles en la web de la asignatura el próximo viernes 15 de Septiembre.

#### Pregunta 1 (10 puntos)

Escribe un procedimiento (*diferencias-mayor-que* lista n) que tome una lista con números y un número n como argumento. Deberá devolver el número de veces que el valor absoluto de la diferencia entre dos números consecutivos en la lista es mayor que el número n. Explica si tu procedimiento es recursivo o iterativo.

Ejemplos:

```
(diferencias-mayor-que '(1 0 4 8 9 0 6) 2)
4
(diferencias-mayor-que '(2 4 6 8 10) 3)
0
```

#### Pregunta 2 (10 puntos)

Supongamos n estudiantes haciendo un examen de LPP. Podemos representar la nota de cada estudiante en cada elemento de una lista (de longitud n). Escribe un procedimiento llamado (*histograma* notas) que tome una lista de notas como parámetro y devuelva su histograma. Es decir, la lista resultante deberá ser de longitud M, donde M es la máxima puntuación obtenida por un alumno en el examen y cada elemento i-ésimo de la lista representa el número de estudiantes que han obtenido la puntuación i en el examen.

Ejemplos:

```
(histograma '(3 2 2 3 2))
(0 0 3 2) ;; ningún estudiante obtuvo 0 puntos
              ;; ningún estudiante obtuvo 1 punto
              ;; 3 estudiantes obtuvieron 2 puntos
              ;; 2 estudiantes obtuvieron 3 puntos
```

```
(histograma '(0 1 0 2))
(2 1 1) ;; 2 estudiantes obtuvieron 0 puntos
          ;; 1 estudiante obtuvo 1 punto
          ;; 1 estudiante obtuvo 2 puntos
```

**Pista:** te puede ser de ayuda comenzar por implementar la función auxiliar `(inc-nth lista n)` que incrementa en 1 la posición n-ésima de la lista, o añade ceros a la derecha (seguidos de un 1) si la posición n es mayor o igual que la longitud de la lista:

```
(define lista '(0 0 0))
(inc-nth lista 0) ; lista = (1 0 0)
(inc-nth lista 1) ; lista = (1 1 0)
(inc-nth lista 0) ; lista = (2 1 0)
(inc-nth lista 4) ; lista = (2 1 0 0 1)
lista
(2 1 0 0 1)
```

Para implementar esta función puedes usar la función de Scheme `(list-tail lista n)` que devuelve la pareja n-ésima de la lista (o sea, la sublista que comienza en la posición n) y debes usar algún mutador para modificar la lista sin tener que crear parejas nuevas.

Ejemplos de `list-tail`:

```
(list-tail '(1 2 3 4) 1) -> (2 3 4)
(list-tail '(1 2 3 4) 4) -> ()
(list-tail '(1 2 3 4) 0) -> (1 2 3 4)
(list-tail '(1 2 3 4) 5) -> error!!
```

### Pregunta 3 (10 puntos)

Supongamos este programa para ordenar listas:

```
(define (sort lst)
  (if (null? lst)
      '()
      (insert (car lst) (sort (cdr lst)))))

(define (insert value sorted)
  (cond ((null? sorted) (list value))
        ((< value (car sorted)) (cons value sorted))
        (else (cons (car sorted)
                    (insert value (cdr sorted))))))
```

Vamos a reescribirlo utilizando mutación, reordenando el orden de las parejas y sin llamar a `cons` para crear nuevas parejas:

```

(define (sort! lst)
  (if (null? lst) '()
      (insert! lst (sort! (cdr lst)))))

(define (insert! value-pair sorted)
  (cond ((null? sorted)
         (set-cdr! value-pair '())
         value-pair)
        ((< (car value-pair) (car sorted))
         ...)
        (else
         ...)))

```

Ejemplo de uso:

```

(sort! (list 7 3 87 5))
(3 5 7 87)

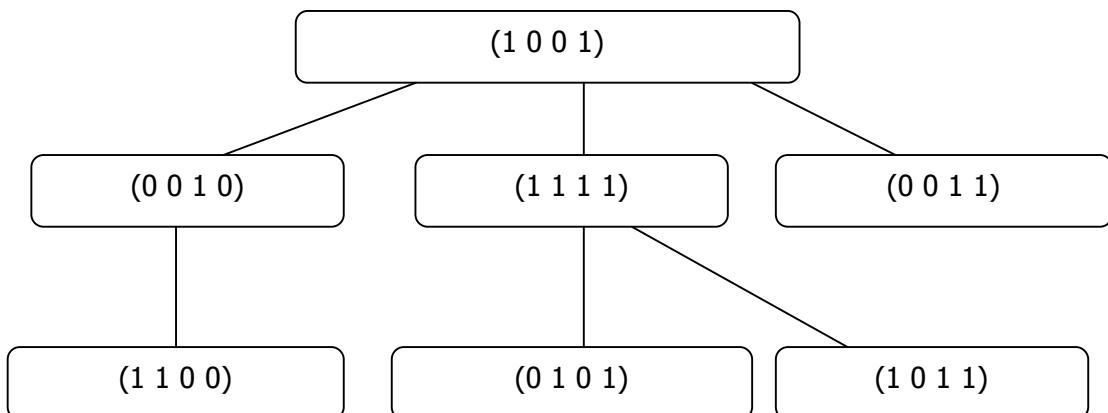
```

a) **(7 puntos)** Completa la definición de `insert!`.

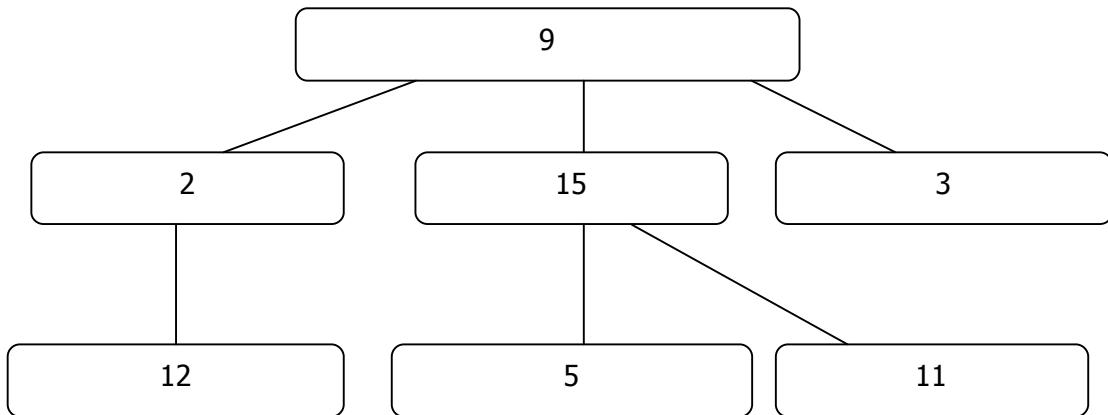
b) **(3 puntos)** Una vez implementado `sort!` e `insert!` te das cuenta de que tal y como están definidos sólo se pueden aplicar a ordenar listas de números. ¿Cómo podrías definirlos de una forma más genérica, para que se pudiera aplicar a cualquier tipo de dato para el que existiera una relación de orden (relación que indica si un dato es menor, mayor o igual que otro).? Explícalo, indicando cómo cambiar las funciones `sort!` e `insert!` y pon un ejemplo.

#### Pregunta 4 (10 puntos)

Un compañero está haciendo una práctica en la que utiliza árboles genéricos que contienen números binarios del 0 al 15 (4 bits). Para representar los números binarios utiliza listas de 4 elementos. De esta forma, su programa maneja árboles como el siguiente:



Preocupado por el consumo de memoria de esta representación, le comentas que sería mejor que representara los números binarios utilizando enteros decimales, con lo que quedaría un árbol como el siguiente:



**a) (3 puntos)** Implementa las funciones (`lista-binaria-a-numero lista`) y (`numero-a-lista-binaria numero`) que transformen un tipo de datos en otro.

**Pista:** Recuerda que el valor decimal de un número binario ( $a_3 a_2 a_1 a_0$ ) es  $a_3*2^3+a_2*2^2+a_1*2^1+a_0*2^0$ .

**b) (5 puntos)** Utilizando las funciones anteriores, implementa los procedimientos (`reduce-tree tree`) y (`amplia-tree tree`) que reciba un árbol genérico como argumento y devuelva el árbol con el tipo de dato reducido (número) o ampliado (lista binaria).

**c) (2 puntos)** Una vez que has implementado las funciones anteriores, te das cuenta de que podríamos mejorar la abstracción si simplemente implementáramos una función (`transforma-tree tree`) que ampliara el árbol si está reducido y lo redujese si está ampliado. Explica cómo implementarías esta función.

### Pregunta 5 (10 puntos)

Supongamos las siguientes expresiones:

```
(define (sumador-x)
  (lambda (y)
    (+ x y)))
(define x 10)
(define foo
  (let ((x 5))
    (sumador-x)))
(foo 3)
```

**a) (2 puntos)** ¿Qué valor devuelve la última expresión?

**b) (5 puntos)** Dibuja el diagrama de entornos resultante de evaluar las expresiones.

**c) (3 puntos)** Explica la evaluación de las expresiones usando el modelo de entorno.

SOLUCIONES:

**Ejercicio 1:**

```
(define (diferencias-mayor-que lista n)
  (cond
    ((empty? lista) 0)
    ((empty? (cdr lista)) 0)
    ((> (abs (- (car lista) (cadr lista))) n)
     (+ 1 (diferencias-mayor-que (cdr lista) n)))
    (else
      (diferencias-mayor-que (cdr lista) n))))
```

**Ejercicio 2:**

```
(define (anyade-0 lista)
  (begin
    (set-cdr! (list-tail lista (- (length lista) 1))
              (list 0))
    lista))
```

```
(define (inc-nth lista n)
  (if (> n (- (length lista) 1))
      (inc-nth (anyade-0 lista) n)
      (let ((pos-n (list-tail lista n)))
        (set-car! pos-n (+ (car pos-n) 1)))))
```

```
(define (histograma lista)
  (let ((lista-aux '(0)))
    (histograma-aux lista lista-aux)
    lista-aux))
```

```
(define (histograma-aux lista list-aux)
  (if (null? lista)
      list-aux
      (begin
        (inc-nth list-aux (car lista))
        (histograma-aux (cdr lista) list-aux))))
```

**Ejercicio 3:**

```
(define (sort! lst)
  (if (null? lst) '()
      (insert! lst (sort! (cdr lst)))))

(define (insert! value-pair sorted)
  (cond
    ((null? sorted)
     (set-cdr! value-pair '())
     value-pair)
    ((< (car value-pair) (car sorted))
     (set-cdr! value-pair sorted)
     value-pair)
    (else
     (set-cdr! sorted (insert! value-pair (cdr sorted)))
     sorted))))
```

**Ejercicio 4:**

a)

```
(define (lista-binaria-a-num lista)
  (if (null? lista) 0
      (+ (lista-binaria-a-num (cdr lista))
          (* (car lista) (aux (- (length lista) 1))))))

(define (aux n)
  (if (= n 0) 1
      (* 2 (aux (- n 1)))))

.....
```

```
(define (num-a-lista-bin num)
  (cond
    ((= num 1) (list num))
    (else (append (num-a-lista-bin (parte-entera num 2)) (list (remainder num 2))))))
```

```
(define (parte-entera n m)
  (let ((resto (remainder n m)))
    (/ (- n resto) m)))
```

b)

```
(define (reduce-tree tree)
  (make-tree (lista-binaria-a-num (dato tree))
            (map reduce-tree (hijos tree)) ))
```

```
(define (amplia-tree tree)
  (make-tree (num-a-lista-bin (dato tree))
            (map amplia-tree (hijos tree)) ))
```

## **Ejercicio 5**

Devuelve 13

# Lenguajes y Paradigmas de Programación

## Curso 2006-2007

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 60 puntos. Para obtener la nota final (en escala 0-10) se suman los puntos de prácticas y se divide por 6.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 24 puntos en este examen.**
- Se debe contestar cada pregunta **en una hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas (y la fecha de revisión) estarán disponibles en la web de la asignatura el próximo día 2 de Julio.

#### Pregunta 1 (5 puntos)

Contesta a las siguientes preguntas, rodeando la respuesta que consideres correcta (sólo una es cierta). Cada pregunta vale 1 punto.

1. En programación declarativa ¿es posible definir una referencia?
  - (a) Depende del lenguaje de programación.
  - (b) No, porque en programación declarativa las variables sólo son nombres que representan valores, similares a constantes.
  - (c) No, porque la programación declarativa está basada en la declaración de funciones.
  - (d) Sí, por ejemplo cuando damos valor a un parámetro de una función.
2. Un modelo de computación proporciona:
  - (a) Una explicación del funcionamiento de los programas de un lenguaje de programación.
  - (b) Una forma de determinar si un programa es correcto o no.
  - (c) Unas reglas que analizan sintácticamente un programa.
  - (d) Un mecanismo de ejecución virtual de cualquier lenguaje de programación de un paradigma.
3. En el modelo de computación basado en entornos, se crea un entorno nuevo:
  - (a) Cada vez que se llama a una función primitiva.
  - (b) Cada vez que se llama a un procedimiento definido por el usuario.
  - (c) Cada vez que se llama a un “define”.
  - (d) Cada vez que se llama a un “lambda”.
4. Una de las ventajas principales de la utilización de barreras de abstracción es que:
  - (a) Los programas son más rápidos.
  - (b) Los programas se pueden ejecutar en distintos sistemas operativos.

- (c) Se oculta la implementación y se acerca el lenguaje de programación al dominio que se está programando.
- (d) El funcionamiento de los programas se puede comprobar automáticamente.

5. Una de las ventajas principales de los lenguajes fuertemente tipados es que:

- (a) Los entornos de programación pueden detectar más errores mientras el programador va escribiendo el programa.
- (b) Los programas son más eficientes.
- (c) Es posible definir polimorfismo.
- (d) Una función puede devolver más de un tipo de datos.

## Pregunta 2 (5 puntos)

1. Dadas la siguiente expresión:

a) Dibuja el diagrama *box & pointer* correspondiente.

b) Escribe la salida por pantalla que produce Scheme después de la evaluación.

```
(define z
  (let ((y '(7)))
    (let ((y '(8))
          (x (list 'a '(b c) (car y))))
      (set-car! x (car y))
      (set-cdr! x (car (cdr x)))
      x)))
```

## Pregunta 3 (10 puntos)

a) Implementa la función (`split lista n`) que divide una lista en dos por el elemento  $n$ -ésimo **sin utilizar mutadores**. El resultado es una pareja cuyo `car` es la primera sublistas y cuyo `cdr` es la segunda. Ejemplo:

```
(define lista '(1 2 3 4 5))
(define pareja (split lista 3)
(car pareja) -> '(1 2 3)
(cdr pareja) -> '(4 5)
lista -> '(1 2 3 4 5))
```

b) Implementa la función (`split! lista n`) que haga lo mismo que `split` pero **utilizando mutadores**. Ejemplo:

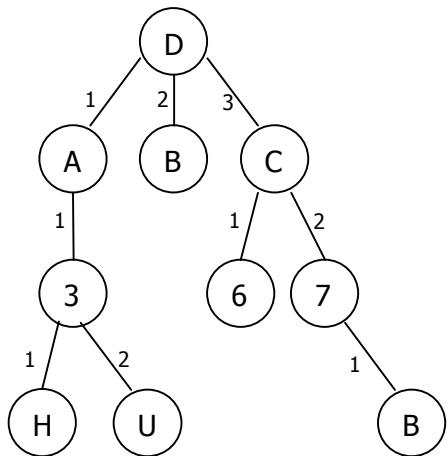
```
(define lista '(1 2 3 4 5))
(define pareja (split lista 3)
(car pareja) -> '(1 2 3)
(cdr pareja) -> '(4 5)
lista -> '(1 2 3))
```

#### Pregunta 4 (10 puntos)

Queremos un procedimiento que nos permita acceder de forma indexada a los elementos de un árbol.

Escribe un procedimiento `tree-ref` que tome un árbol genérico y una lista de índices (números enteros mayores o igual a 1) como argumentos. La secuencia de índices muestra el **camino** a seguir para encontrar un elemento en el árbol.

Por ejemplo, considerando el árbol que se muestra a continuación, la llamada a `(tree-ref tree '(3 1))` devuelve 6. El primer índice, el 3, selecciona el subárbol que se encuentra en la tercera rama del árbol, y el índice 1 selecciona el subárbol que está en la primera rama de ese subárbol. Devuelve el dato que se encuentra en el subárbol indexado (el 6).



Más ejemplos:

```
(tree-ref tree '(1 1 1))
H
(tree-ref tree '(3 2))
7
(tree-ref tree '(1))
A
```

Suponemos que la secuencia de índices siempre es correcta, es decir, proporciona un camino existente en el árbol.

## Pregunta 5 (10 puntos)

Supongamos que necesitamos mantener simultáneamente dos implementaciones del tipo de dato `rectangulo2D`, que llamaremos `rect-base-altura` y `rect-coords`.

En la primera implementación representamos un rectángulo por la posición (x e y) de su esquina inferior izquierda y por su base y altura.

En la segunda implementación lo representamos mediante la posición de su esquina inferior izquierda y de su esquina superior derecha.

a) (5 puntos) Define la barrera de abstracción de los tipos específicos `rect-base-altura` y `rect-coords` (un mínimo de 3 funciones), indicando el nombre de cada función, sus argumentos y su comportamiento. Define una barrera de abstracción común con un mínimo de 3 funciones genéricas (polimórficas) que admitan cualquier tipo específico de rectángulo, describiendo su comportamiento.

b) Define una implementación completa de las funciones anteriores, utilizando el enfoque de paso de mensajes.

## Pregunta 6 (10 puntos)

Estamos diseñando una aventura mediante PDD. Se trata de un laberinto formado por un conjunto de habitaciones unidas por pasadizos. Un ejemplo de utilización (en negrita aparece lo que el jugador escribe por teclado) sería:

### (aventura)

Te encuentras en la entrada de una cueva. Hay un túnel hacia el este.

Movimiento? **este**

Te encuentras en una cueva amplia. Las paredes brillan. Hay túneles al este y al sur. Hay una varita mágica en el suelo.

Movimiento? **coger varita**

Aparece una nube de humo. Te encuentras ahora en una habitación con paredes de oro. Un frasco con una poción verde está en el suelo. Hay unas puertas al norte y al sur.

Movimiento?

...

Y así sucesivamente. El juego está programado de la siguiente forma:

```
(put 'entrada 'descripcion "Te encuentras en la entrada de una cueva.  
Hay un túnel hacia el este")  
(put 'entrada 'este (lambda () (visit 'cueva)))  
(put 'cueva 'descripcion "Te encuentras en una cueva amplia. Las  
paredes brillan. Hay túneles al este y al sur. Hay una varita mágica  
en el suelo")  
(put 'cueva 'este (lambda () (visit 'camara-tortura)))  
(put 'cueva 'sur (lambda () (visit 'biblioteca)))  
(put 'varita 'coger (lambda () (visit 'sala-oro)))  
(put 'varita 'agitar (lambda () (visit 'biblioteca)))  
...  
...
```

Tenemos los siguientes procedimientos (suponemos que `read-line` es una primitiva que devuelve la petición que el jugador escribe en forma de lista de identificadores):

```
(define (aventura)
  (visit 'entrada))

(define (visit habitacion)
  (newline)
  (display (get habitación 'descripcion))
  (newline)
  (display "Movimiento? ")
  (actua habitacion (read-line)))
```

Implementa el procedimiento `actua` que reciba la habitación donde se encuentra el jugador y una petición como argumento.

Se considera que todo lo que el jugador teclea es correcto.

### Pregunta 6 (10 puntos)

Supongamos las siguientes expresiones en Scheme:

```
(define (g x)
  (let ((z 8))
    (lambda (y)
      (set! z (+ x y))
      (set! x (+ z y))
      (+ x y z)))
  (define h (g 4))
  (h 6))
```

- a) (2 puntos) ¿Qué valor devolverá la última expresión?
- b) (5 puntos) Dibuja y explica el diagrama de entornos creado al ejecutar las expresiones.
- c) (3 puntos) ¿Tiene la función `g` estado local? ¿Tiene la función `h` estado local? ¿Qué variables contienen el estado local?

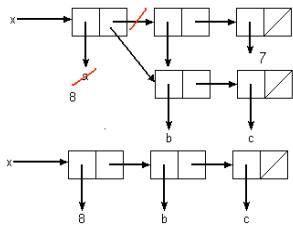
## Soluciones Examen Junio 2007 LPP

### Pregunta 1

1. b
2. a
3. b
4. c
5. a

### Pregunta 2

a)



b)

'(8 b c)

### Pregunta 3

a) (5 puntos)

*Esto es una posible solución: hay bastantes más.*

```
(define (split lista n)
  (if (equal? n 0)
      (cons '() lista)
      (let ((result (split (cdr lista) (- n 1))))
        (cons
          (cons (car lista) (car result))
          (cdr result)))))
```

b) (5 puntos)

```
(define (do-split! lista n)
  (if (equal? n 1)
      (let ((result (cdr lista)))
        (set-cdr! lista '())
        result)
      (do-split! (cdr lista) (- n 1)))

(define (split! lista n)
  (if (equal? n 0)
      (cons '() lista)
      (cons lista (do-split! lista n))))
```

### Pregunta 4

```
(define (tree-ref tree camino)
  (cond ((null? tree) #f)
        ((null? camino) (dato tree))
        (else (forest-ref (hijos tree) (car camino) (cdr camino)))))

(define (forest-ref forest num-hijo camino)
  (if (= num-hijo 1)
      (tree-ref (car forest) camino)
      (forest-ref (cdr forest) (- num-hijo 1) camino)))
```

### Pregunta 5

a) Barrera de abstracción

*Lo siguiente es una posible solución. Hay muchas, pero para ser correctas deben tener:*

- Correcta especificación de las funciones (explicar cuáles son los argumentos y el resultado)
- Nombres correctos (incluir el sufijo con el tipo)
- Deben coincidir las funciones específicas de ambos tipos (la barrera de abstracción debe ser igual en ambos)

Funciones específicas del tipo "rect-ba" (rectángulo base altura):

(make-rect-ba x y base altura): devuelve un rectángulo con la posición (x y) y la base y altura especificada.  
 (base-ba rect-ba): devuelve la base de un rectángulo-ba  
 (altura-ba rect-ba): devuelve la altura de un rectángulo-ba  
 (posx-ba rect-ba): devuelve la posición x de un rectángulo-ba  
 (posy-ba rect-ba): devuelve la posición y de un rectángulo-ba

Funciones específicas del tipo "rect-coords" (rectángulo coordenadas):

(make-rect-coords x1 y1 x2 y2): devuelve un rectángulo con la esquina inf. izqd. situada en la posición (x1 y1) la esquina inf. dcha.en la posición (x2 y2)  
 (base-coords rect-coords): devuelve la base de un rectángulo-coords  
 (altura-coords rect-coords): devuelve la altura de un rectángulo-coords  
 (posx-ba rect-coords): devuelve la posición x de un rectángulo-coords  
 (posy-ba rect-coords): devuelve la posición y de un rectángulo-coords

Funciones genéricas

(make-rect-from-base-altura x y base altura): devuelve un rectángulo con la posición (x y) y la base y altura especificada.  
 (area rect): devuelve el área de un rectángulo  
 (perimetro rect): devuelve el perímetro de un rectángulo

b) Implementación con paso de mensajes:

```
(define (make-rect-from-base-altura x y base altura)
```

```

(lambda (mensaje)
  (cond
    ((equal? mensaje 'base) base)
    ((equal? mensaje 'altura) altura)
    ((equal? mensaje 'x) x)
    ((equal? mensaje 'y) y)
    (else
      (error "operacion desconocida")))))

(define (make-rect-from-coords x1 y1 x2 y2)
  (lambda (mensaje)
    (cond
      ((equal? mensaje 'base) (- x2 x1))
      ((equal? mensaje 'altura) (- y2 y1))
      ((equal? mensaje 'x) x1)
      ((equal? mensaje 'y) y2)
      (else
        (error "operacion desconocida")))))

```

**Pregunta 6**

```

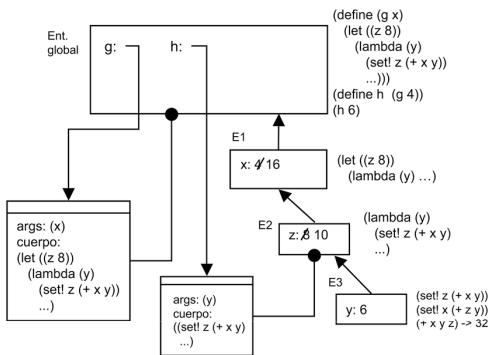
(define (actua habitacion accion)
  (if (= (length accion) 1)
      ((get habitacion (car accion)))
      ((get (cadr accion) (car accion)))))


```

**Pregunta 7**

a) 32

b)



c) Sólo la función h tiene estado local y está representado por las variables x y z.

# **Lenguajes y Paradigmas de Programación**

## **Curso 2006-2007**

### **Examen de la Convocatoria de Septiembre**

## **Normas importantes**

- La puntuación total del examen es de 60 puntos. Para obtener la nota final (en escala 0-10) se suman los puntos de prácticas y se divide por 6.
  - Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 24 puntos en este examen**.
  - Se debe contestar cada pregunta **en una hoja distinta**. No olvides poner el nombre en todas las hojas.
  - La duración del examen es de 3 horas.
  - Las notas (y la fecha de revisión) estarán disponibles en la web de la asignatura el próximo día 18 de Septiembre.

## **Pregunta 1 (10 puntos)**

a) (5 puntos) Vamos a crear un sistema para manipular polinomios:

$$\begin{aligned} p_1(x) &= x^3 + 3x + 2 \\ p_2(x) &= 3x^2 + 2x + 5 \end{aligned}$$

Representaremos estos polinomios por sus coeficientes, almacenándolos en orden incremental de su potencia. Las expresiones anteriores se representarían como:

$$\begin{pmatrix} 2 & 3 & 0 & 1 \\ 5 & 2 & 3 \end{pmatrix}$$

Para sumar dos polinomios, necesitamos sumar sus coeficientes:

$$p_1(x) + p_2(x) = (x^3 + 3x + 2) + (3x^2 + 2x + 5) = (x^3 + 3x^2 + 5x + 7)$$

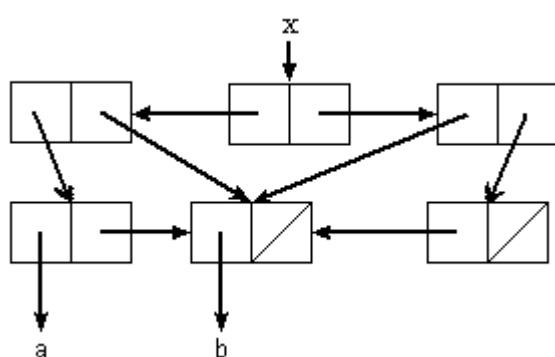
En nuestra representación obtendríamos  $(7 \ 5 \ 3 \ 1)$ .

Implementa el procedimiento suma-polinomios que tome dos polinomio (listas de coeficientes) como argumento y devuelva un nuevo polinomio (lista de coeficientes) resultado de la suma de ambos.

b) (5 puntos) Dado el siguiente diagrama *box & pointer*.

b.) Escribe las instrucciones de Scheme que lo generan.

b.2) Dibuja el diagrama *box & pointer* resultante después de evaluar la mutación: (set-car! (caddr x) (caar x))



## Pregunta 2 (10 puntos)

a) (5 puntos) Implementa la función (`split-k lista n`) que divide una lista original en  $k$  sublistas. Las primeras  $k-1$  sublistas deben tener exactamente  $n$  elementos escogidos secuencialmente de la lista original. La última sublista tendrá los elementos sobrantes (pueden ser también  $n$  si el número de elementos de la lista original es módulo  $n$ ). Debes hacer la implementación **sin utilizar mutadores**. Ejemplo:

```
(define lista '(1 2 3 4 5 6 7 8 9 10))
(split-k lista 4)-> '((1 2 3 4)(5 6 7 8) (9 10))
(split-k lista 3)-> '((1 2 3)(4 5 6)(7 8 9) (10))
```

b) (5 puntos) Implementa la función (`split-k! lista n`) que haga lo mismo que `split-k` pero **utilizando mutadores**. En esta versión no se debe llamar a `cons` más de  $k$  veces.

## Pregunta 3 (10 puntos)

Queremos construir un tipo de datos `matrix` que nos permita manipular matrices representadas como una lista de listas, por ejemplo:

```
((10 11 12 13 14)
 (15 16 17 18 19)
 (20 21 22 23 24))
```

representaría la matriz:

```
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

a) (2 puntos) Define la barrera de abstracción del tipo de dato `matrix` e implementa 2 funciones de esta barrera de abstracción.

b) (4 puntos) Escribe un procedimiento (`make-matrix rows cols start`) que construya una matriz de `rows` filas por `cols` columnas, empezando en `start`, como indica el ejemplo:

```
(make-matrix 3 5 10) devolverá:
 ((10 11 12 13 14)(15 16 17 18 19)(20 21 22 23 24))
```

c) (4 puntos) Escribe un procedimiento (`transpuesta m`) que tome una matriz como argumento y devuelva su matriz transpuesta. Ejemplo:

```
(define m (make-matrix 3 5 10))
(transpuesta m) => ((10 15 20)(11 16 21)(12 17 22)(13 18 23)(14
19 24))
```

#### Pregunta 4 (10 puntos)

Supongamos que estamos implementando una calculadora con la que podemos lanzar operaciones (factorial, cuadrado, etc.) sobre un único número. A la calculadora le pasamos un símbolo que representa el nombre de una función a aplicar y un número al que aplicar esa función. Una posible implementación de este problema sería la siguiente:

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

(define (cuadrado n)
  (* n n))

(define (calculadora funcion n)
  (cond
    ((equal? funcion 'factorial) (factorial n))
    ((equal? funcion 'cuadrado) (cuadrado n)))
    (else (error "funcion desconocida"))))

(calculadora 'factorial 3) -> 6
(calculadora 'doble 3) -> Error funcion desconocida
```

- a) El problema fundamental de esta implementación es que no es posible añadir nuevas funciones a la calculadora sin modificar el procedimiento `calculadora`. Si queremos añadir una función que, por ejemplo, devuelva el doble de un número habría que definir esa función y modificar el condicional que implementa `calculadora`.

(4 puntos) Cambia la implementación del procedimiento `calculadora` para que pueda trabajar con un número indeterminado de funciones. Puedes utilizar estructuras de datos adicionales. Explica qué habría que hacer para añadir nuevas funciones a la calculadora.

(2 puntos) Define e implementa un procedimiento `añade-funcion` para añadir nuevas funciones a la calculadora. ¿Qué parámetros tendría? ¿Cómo se implementaría?

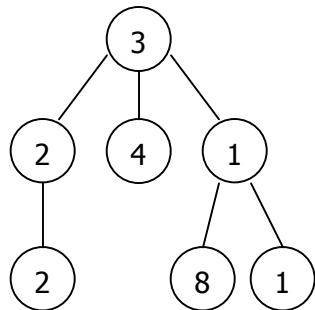
- b) Otro problema de la implementación de `calculadora` es que únicamente se puede calcular con un número.

(4 puntos) Modifica el procedimiento `calculadora` para que admita más de un número. Por ejemplo,

```
(calculadora 'suma 2 3 4) -> 9
(calculadora 'media 2 3 4) -> 3
```

### Pregunta 5 (10 puntos)

Escribe un procedimiento `sumaniveltree` que tome un árbol genérico y un número que indique un nivel como argumentos, y devuelva la suma de los datos de ese nivel del árbol. Consideramos que la raíz tiene nivel 0.



```
(sumaniveltree 0 tree)
3
(sumaniveltree 1 tree)
7
(sumaniveltree 2 tree)
20
```

### Pregunta 6 (10 puntos)

Supongamos las siguientes expresiones en Scheme:

```
(define x 3)
(define z 10)
(define g
  (let ((z 8)
        (x (+ z 3)))
    (lambda (z)
      (set! x (+ x z))
      x)))
(g 5)
```

- (6 puntos) Dibuja y explica el diagrama de entornos creado al ejecutar las expresiones.
- (2 puntos) ¿Qué valor devolverá la última expresión?
- (2 puntos) ¿Cómo modificarías una única línea del programa para que la misma llamada `(g 5)` devolviera 14?

## Pregunta 1

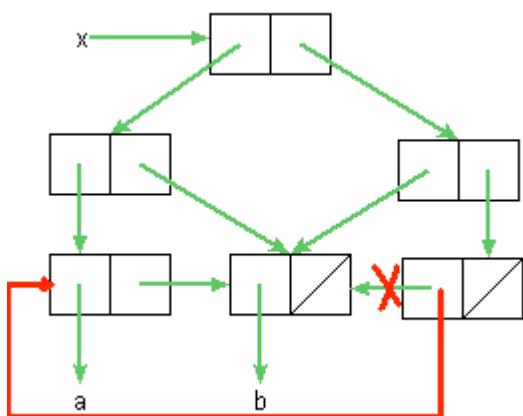
a)

```
(define (suma-polinomios pol1 pol2)
  (cond ((null? pol1) pol2)
        ((null? pol2) pol1)
        (else (cons (+ (car pol1) (car pol2))
                    (suma-polinomios (cdr pol1) (cdr pol2)))))))
```

b)

b.1) (define x  
 (let\* ((y (cons 'b '()))  
 (w (cons 'a y))  
 (z (cons y '())))  
 (cons (cons w y) (cons y z))))

b.2)



## Pregunta 2

a)

```
(define (split-k lista n)
  (if (< (length lista) n)
      (list lista)
      (append (list (primeros-n lista n))
              (split-k (resto-n lista n) n)))))

(define (primeros-n lista n)
  (if (= n 0)
      '()
      (cons (car lista)
            (primeros-n (cdr lista) (- n 1)))))

(define (resto-n lista n)
  (if (= n 0)
      lista
      (resto-n (cdr lista) (- n 1))))
```

```

b)
(define (split-k! lista n)
  (if (< (length lista) n)
      (list lista)
      (let ((lista2 (corta-n lista n)))
        (cons lista
              (split-k! lista2 n)))))

(define (corta-n lista n)
  (if (= n 1)
      (let ((lista2 (cdr lista)))
        (set-cdr! lista '())
        lista2)
      (corta-n (cdr lista) (- n 1))))

```

### Pregunta 3

a) Una posible barrera de abstracción sería:

Constructor:

(make-matriz fil col): devuelve una matriz de fil filas y col columnas.

Selectores:

(num-filas-matriz matriz): devuelve el número de filas que tiene la matriz matriz.

(num-cols-matriz matriz): devuelve el número de columnas que tiene la matriz matriz.

Funciones específicas:

(insert-fila-matriz fila matriz): añade la fila fila a la matriz matriz.

(get-elem-matriz fila col matriz): devuelve el elemento situado en la fila fila y columna col de la matriz matriz.

(get-fila-matriz fila matriz): devuelve la fila indicada por fila de la matriz matriz.

(get-col-matriz col matriz): devuelve la columna indicada por col de la matriz matriz.

Implementación de dos funciones de esta barrera de abstracción:

```

(define (num-filas-matriz m)
  (if (null? m) 0
      (length m)))

(define (num-cols-matriz m)
  (if (null? m) 0
      (length (car m))))

b)
(define (make-matrix rows cols start)
  (if (= rows 0) '()
      (cons (make-fila cols start)
            (make-matrix (- rows 1) cols (+ start cols)))))

(define (make-fila cols start)
  (if (= cols 0) '()
      (cons start (make-fila (- cols 1) (+ start 1)))))

c)
(define (transpuesta m)
  (cond
    ((null? (car m)) '())
    (else (cons (map car m) (transpuesta (map cdr m))))))

```

#### Pregunta 4

a) Utilizamos una tabla hash similar a la definida en el tema de programación dirigida por los datos, que relaciona los identificadores con los procedimientos. La interfaz de esta tabla hash sería:

(**put identificador procedimiento**): añade un procedimiento a la tabla hash, asociándolo al identificador  
(**get identificador**): devuelve el procedimiento asociado al identificador

Ejemplos de uso:

```
(put 'suma +)
(put 'doble (lambda (x) (+ x x))
(get 'doble) -> devuelve el procedimiento
```

Implementación de calculadora:

```
(define (calculadora funcion n)
  (let ((proc (get funcion)))
    (if (null? proc)
        (error "funcion desconocida")
        (proc n))))
```

Para añadir nuevas funciones a la calculadora ya no hay que tocar la función **calculadora**, basta con añadir una nueva pareja (identificador, procedimiento) a la tabla hash usando la función **put**:

```
(define (añade-funcion ident proc)
  (put ident proc))

b)

(define suma +)

(define (media . args)
  (/ (apply suma args) (length args)))

(define (calculadora funcion . args)
  (cond
    ((equal? funcion 'factorial) (factorial (car args)))
    ((equal? funcion 'cuadrado) (cuadrado (car args)))
    ((equal? funcion 'suma) (apply suma args))
    ((equal? funcion 'media) (apply media args)))
    (else (error "funcion desconocida"))))
```

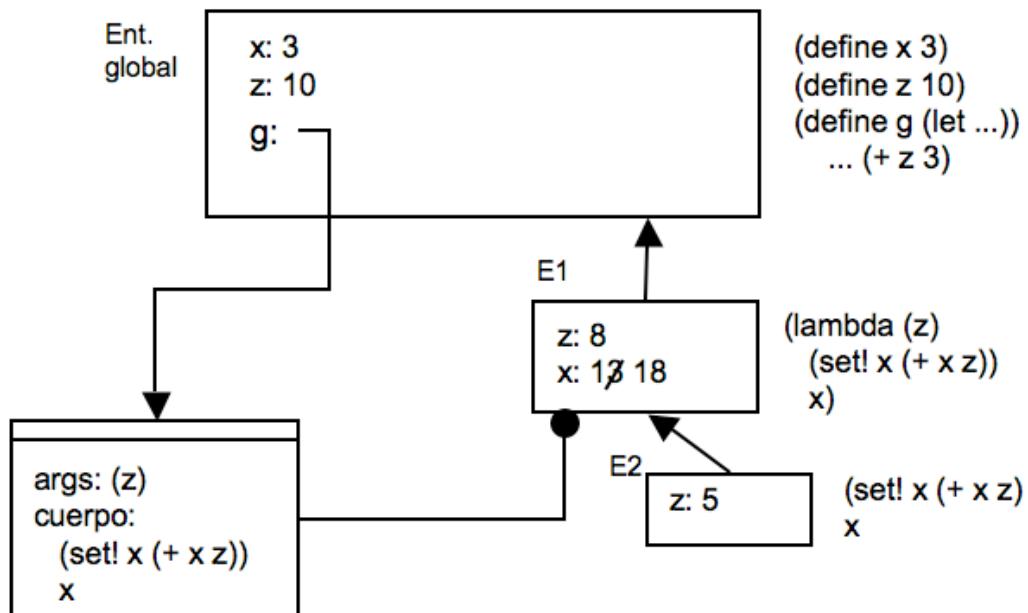
#### Pregunta 5

```
(define (suma-nivel-tree n tree)
  (if (= n 0) (dato tree)
      (suma-nivel-forest n (hijos tree)))))

(define (suma-nivel-forest n forest)
  (if (null? forest) 0
      (+ (suma-nivel-tree (- n 1) (car forest))
          (suma-nivel-forest n (cdr forest)))))
```

**Pregunta 6**

a)



b) 18

c) (define z 6)

# Lenguajes y Paradigmas de Programación

## Curso 2007-2008

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 100 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 40 puntos en este examen.**
- Se debe contestar cada pregunta **en una hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas estarán disponibles en la web de la asignatura el próximo día 1 de Julio

#### Pregunta 1 (16 puntos)

Escribe el procedimiento `alterna-f` que tome dos procedimientos como argumento y devuelva un procedimiento que los aplique alternativamente a los elementos de una lista.

```
((alterna-f square (lambda (x) (+ 1 x)) ' (5 6 7 8))
 (25 7 49 9)
(define updown (alterna-f (lambda (x) (+ x 100))
                           (lambda (x) (- x 100))))
(updown ' (0 1 -1 2 -2 3 -3))
(100 -99 99 -98 98 -97 97)
```

#### Pregunta 2 (16 puntos)

Escribe el procedimiento `diff-pt` que tome como argumentos dos listas de listas (pseudoárboles) con la misma estructura, pero posiblemente con diferentes elementos, y devuelva una lista de parejas que contenga los diferentes elementos. Ejemplo:

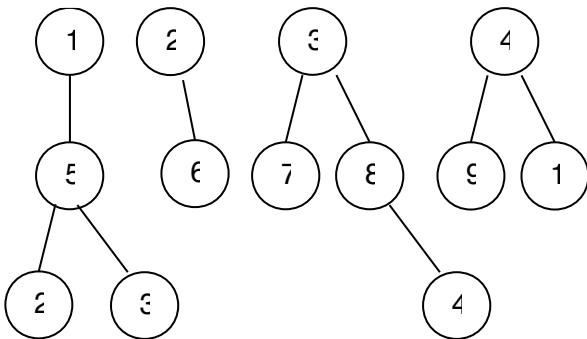
```
(diff-pt '(a (b ((c)) d e) f) '(1 (b ((2)) 3 4) f))
 ((a . 1) (c . 2) (d . 3) (e . 4))
 (diff-pt '() '())
 ()
 (diff-pt '((a b) c) '((a b) c))
 ()
```

#### Pregunta 3 (16 puntos)

Escribe el procedimiento `anchura-forest` que devuelva la máxima anchura de un bosque (lista de árboles). La anchura de un bosque en un nivel determinado se define como la suma del número de nodos que hay en esa nivel en todos los árboles del bosque. Por ejemplo, en la siguiente figura, la anchura del bosque en el nivel 1 sería 4, en el nivel 2 sería 6 y en el nivel 3 sería 3.

Ejemplo:

bosque:



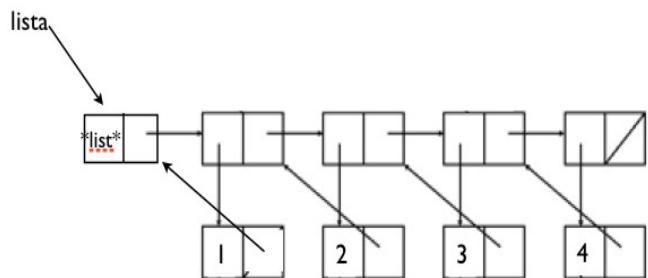
(anchura-forest bosque)

6

La anchura del bosque es 6 por ser la mayor anchura de todos los niveles.

#### Pregunta 4 (20 puntos)

Vamos a construir un nuevo tipo de dato, llamado *lista doblemente enlazada* o *dlist*. Esta estructura de datos tiene la propiedad de que, en cada momento, se puede acceder tanto al siguiente elemento como al anterior. Para implementarlo, definimos los elementos de esta lista como parejas cuya parte izquierda contiene el dato propiamente dicho y la parte derecha el enlace al elemento anterior. La lista va precedida de una pareja que hace el papel de cabecera y que es referenciada por las variables que apuntan a ella. En la figura puedes ver esta implementación:



Tenemos las siguientes funciones de su barrera de abstracción:

```
(define empty-dlist '())
(define (empty-dlist? obj)
  (null? Obj))
(define (first? l)
  (equal? '*list* (car l)))
(define (value dlist)
  (if (empty-dlist? dlist)
    (error "Error dlist vacía")
    (caar dlist)))
(define (next dlist)
  (if (empty-dlist? dlist)
    (error "Error dlist vacía")
    (cdr dlist)))
(define (previous dlist)

  (cond
    ((first? dlist) dlist)

    ((empty? dlist) (error "Error dlist vacía"))
```

```
(else (cdar dlist))))
```

Vamos a completar la barrera de abstracción.

**a) (6 puntos)** Define el procedimiento (`make-dlist`) que construya una lista vacía y el procedimiento (`add-dlist dlist x`) que añada un primer elemento `x` en la primera posición de una lista vacía. La variable `dlist` referencia la cabecera de la lista.

**a) (10 puntos)** Define el procedimiento (`insert-dlist pos-dlist x`) que inserte un elemento `x` en la posición de la lista definida por `pos-dlist`. Esta posición puede ser la posición inicial (`pos-dlist` apuntando a la cabecera) o cualquier posición intermedia resultante de llamadas a `next`. ¡Cuidado con el caso especial del final de la lista!

**b) (4 puntos)** Define el procedimiento (`list-to-dlist lista`) que construya una `dlist` con los elementos de la lista pasada como argumento.

### Pregunta 5 (16 puntos)

Supongamos que representamos las fechas como un tipo de dato con los campos `día`, `mes` y `año`, todos ellos enteros. Queremos definir una función `fecha-to-string` a la que le pasemos un dato de tipo fecha y un símbolo identificando el formato de conversión de esa fecha. La función nos tiene que devolver una cadena con el formato escogido.

Ejemplo:

```
(define fecha (make-fecha 10 11 2007)) ; 10 de noviembre de 2007
(fecha-to-string fecha 'guiones-2-digitos) -> "10-11-07"
(fecha-to-string fecha 'guiones-alfa) -> "10-nov-2007"
(fecha-to-string fecha 'barras-4-digitos) -> "10/11/2007"
```

Queremos implementar la función `fecha-to-string` de forma que podamos añadir nuevos formatos **sin tener que reescribir su código**.

**a) (8 puntos)** Implementa la función `fecha-to-string` de forma que sea posible añadir nuevos formatos de fecha.

**b) (8 puntos)** Define una función para añadir los formatos de fecha, y escribe un ejemplo completo de funcionamiento, por ejemplo el correspondiente al formato '`'guiones-alfa` del ejemplo anterior (recuerda que la función de Scheme para concatenar cadenas es `string-append`)

### Pregunta 6 (16 puntos)

Supongamos las siguientes expresiones en Scheme:

```
(define y 5)
(define x 8)
(let ((x (+ y 3))
      (f (lambda (x)
            (+ x y)))
      (y 10))
  (f x))
```

Dibuja **(10 puntos)** y explica **(6 puntos)** el diagrama de entornos creado al ejecutar las expresiones.

# Lenguajes y Paradigmas de Programación

## Curso 2007-2008

### Examen de la Convocatoria de Septiembre

#### Normas importantes

- La puntuación total del examen es de 100 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 40 puntos en este examen.**
- Se debe contestar cada pregunta **en una hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas estarán disponibles en la web de la asignatura el próximo día 19 de Septiembre. La revisión se realizará la semana siguiente en una fecha que se anunciará en la web el mismo día en que estén disponibles las notas.

#### Pregunta 1 (16 puntos)

Escribe el procedimiento mutador (`split! lista`) que tome una lista como argumento y la divida en dos: la primera con los elementos de las posiciones impares de la lista y la segunda con los de las pares. La función no debe crear ninguna pareja nueva (con la excepción de la que se devuelve), sino que se debe modificar los punteros de la lista original.

```
(define l '(a b c d e f))
(define p (split! l))
(car p) → (a c e)
(cdr p) → (b d f)
l → (a c e)
```

#### Pregunta 2 (16 puntos)

Escribe el procedimiento mutador (`reemplazar! pred proc lista`) que reemplace cada elemento de la lista que cumpla el predicado `pred` por la aplicación del procedimiento `proc` correspondiente a ese elemento.

```
(define l '(1 2 3 4))
(reemplazar! even? square l)
l → '(1 4 3 16)
```

#### Pregunta 3 (18 puntos)

Supongamos el siguiente escenario:

“Necesitamos un sistema informático que gestione el calendario de un profesional que atiende a pacientes y les da citas.

- El sistema debe poder añadir y eliminar citas, así como detectar posibles solapes entre ellas.
- El sistema debe proporcionar los huecos libres de un determinado día.”

Realiza lo siguiente:

1. **(6 puntos)** Escribe la barrera de abstracción en Scheme de todos los tipos de datos que utilizarías para implementar este escenario.
2. **(6 puntos)** Implementa las funciones definidas en la barrera de abstracción (todas las que puedas).

**3.(6 puntos)** Escribe un conjunto de pruebas que validen todas las funcionalidades de la barrera de abstracción. Cada prueba debe consistir en un conjunto de instrucciones en Scheme que realizan una llamada a la barrera de abstracción utilizando datos concretos junto con una indicación de cuál es el resultado esperado.

#### Pregunta 4 (18 puntos)

Vamos a construir un procedimiento `maquina-del-tiempo` que recuerde los valores de los argumentos que se le han pasado. Cuando se llame al procedimiento `maquina-del-tiempo`, se guarda el valor de la llamada actual, y devuelve el valor que se pasó `n` veces antes, donde `n` es un número que se proporciona cuando la máquina del tiempo se construye. Si la máquina es nueva (se han hecho menos de `n` llamadas), se devuelve el valor inicial con el que la máquina se creó.

`(make-maquina-tiempo n valor_ini) -> Crea una máquina del tiempo que devuelve valores de n llamadas anteriores, con valor inicial valor_ini.`

Ejemplos:

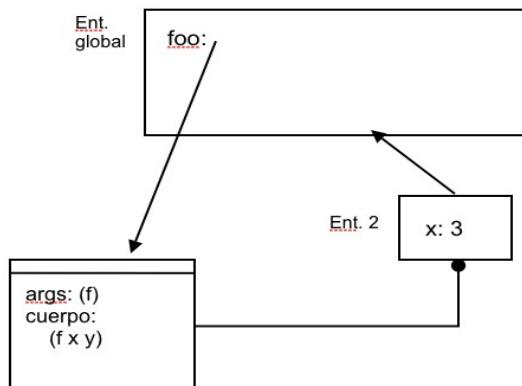
```
(define mt1 (make-maquina-tiempo 1 #f))
(mt1 5) → #f
(mt1 'a) → 5
(mt1 8) → a

(define mt2 (make-maquina-tiempo 2 'holo))
(mt2 5) → hola
(mt2 'a) → hola
(mt2 8) → 5

(define mt0 (make-maquina-tiempo 0 #f))
(mt0 5) → 5
(mt0 'a) → a
(mt0 8) → 8
```

#### Pregunta 5 (16 puntos)

Supongamos el siguiente diagrama de entornos:



**1.(6 puntos)** Escribe un conjunto de instrucciones que genere ese diagrama de entornos.

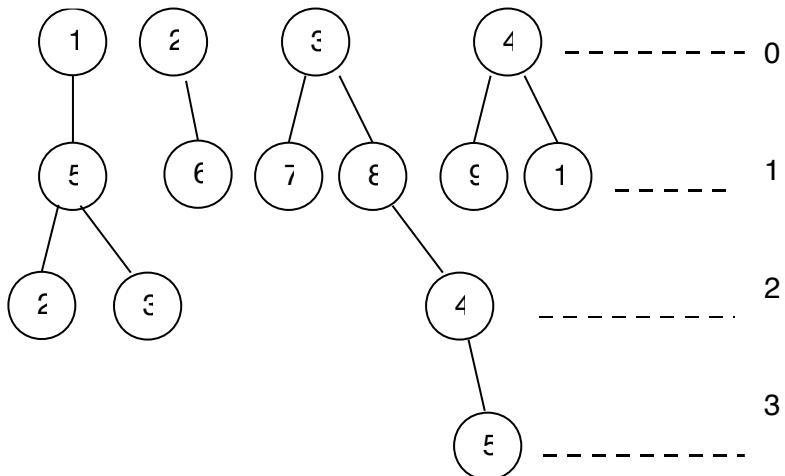
**2.(5 puntos)** Escribe un conjunto de instrucciones que, ejecutadas en el entorno global, tengan como resultado 15 y dibuja el diagrama de entornos resultantes.

**3.(5 puntos)** ¿Sería posible cambiar el valor de x en el entorno 2? Si crees que sí, indica cómo hacerlo escribiendo el código Scheme que haría el cambio.

**Pregunta 6 (16 puntos)**

Escribe el procedimiento profundidad-forest que devuelva la máxima profundidad de un bosque (lista de árboles).

bosque:



(profundidad-forest bosque)  
3

# Lenguajes y Paradigmas de Programación

Curso 2010-2011

Examen Final de la convocatoria de junio

---

## Normas importantes

- La puntuación total del examen es de 50 puntos sobre los 100 que se valora la nota de la asignatura.
  - Se debe contestar **cada pregunta en una hoja distinta**. No olvides poner el nombre en todas las hojas.
  - La duración del examen es de 3 horas.
  - Las notas estarán disponibles en la web de la asignatura el viernes 17 de junio.
  - La revisión será el lunes 20 de junio a las 10:00h en el despacho de Domingo Gallardo.
- 

## Pregunta 1 (8 puntos)

**a) (3 puntos)** Describe las características principales de los siguientes paradigmas de programación, e indica dos lenguajes de programación que se puedan ubicar en cada uno de los paradigmas:

- Paradigma funcional
- Paradigma declarativo
- Paradigma imperativo
- Paradigma orientado a objetos

**b) (3 puntos)** Define y escribe un ejemplo en Scala de los siguientes conceptos:

- Coercion
- Sobrecarga
- Polimorfismo en tiempo de ejecución

**c) (2 puntos)** Dado el siguiente ejemplo:

```
abstract class Bar { def bar(x: Int) : Int }
class Foo extends Bar { def bar(x: Int) = x }

trait Foo1 extends Foo { abstract override def bar(x: Int) = super.bar(x * 3) }
trait Foo2 extends Foo { abstract override def bar(x: Int) = super.bar(x + 3) }
trait Foo3 extends Foo { abstract override def bar(x: Int) = x + super.bar(x) }
trait Foo4 extends Foo { abstract override def bar(x: Int) = x * super.bar(x) }
```

Rellena los huecos con el resultado de las siguientes llamadas y explica tu respuesta:

```
scala> (new Foo with Foo2 with Foo4).bar(5)
scala> _____
```

```
scala> (new Foo with Foo2 with Foo1).bar(5)
scala> _____
```

```
scala> (new Foo with Foo3 with Foo4).bar(5)
scala> _____
```

## Pregunta 2 (7 puntos)

Supongamos que estamos desarrollando en Scheme un programa de estadística y que guardamos una lista de frecuencias (número de veces que se repite un valor) como una lista de parejas (`<valor> <veces>`). Por ejemplo, la lista ((1.3) (2.4) (4.8)) representa que el valor 1 ha aparecido 3 veces, el 2 4 veces y el 4 8 veces.

La media de estos valores se puede expresar como

$$(v_1 * n_1 + v_2 * n_2 + \dots + v_n * n_n) / (n_1 + n_2 + \dots + n_n),$$

siendo  $v_i$  el valor  $i$ -ésimo y  $n_i$  su frecuencia. Por ejemplo, en el caso anterior, la media de los valores es:

$$(1*3+2*4+4*8)/3+4+8 = 2,87$$

Define una función recursiva (`frecuencias lista-freccs`) que devuelva una pareja donde la parte izquierda corresponda a  $(v_1 * n_1 + v_2 * n_2 + \dots + v_n * n_n)$  y la parte derecha a  $(n_1 + n_2 + \dots + n_n)$ . Utiliza esta función para calcular la media. Ejemplo:

```
> lista-frec
((1 . 3) (2 . 4) (4 . 8))
> (frecuencias lista-frec)
(43.15)
> (media lista-frec)
2,87
```

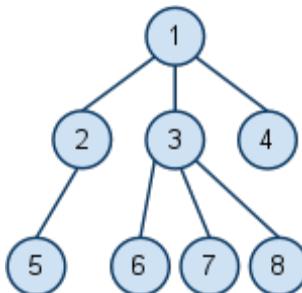
## Pregunta 3 (9 puntos)

**a) (3 puntos)** Define en Scheme la barrera de abstracción del árbol genérico y utilízala para implementar el procedimiento (`equal-tree? tree1 tree2`) que reciba dos árboles genéricos como argumento y devuelva #t si son iguales y #f en caso contrario.

**b) (6 puntos)** Vamos a implementar en Scala el tipo Tree (como un árbol genérico). Su implementación es:

```
class Tree (val dato: Int, val hijos: List[Tree])
```

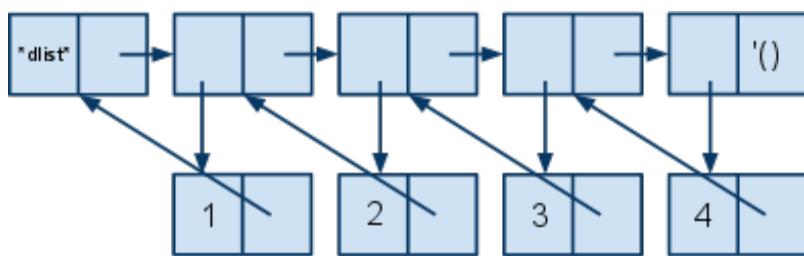
**b.1)** Define el siguiente árbol utilizando el tipo de dato Tree:



**b.2)** Añade al tipo Tree el método `sumaTotalTree` que reciba un Tree como argumento y devuelva la suma de todos sus nodos.

#### Pregunta 4 (9 puntos)

Vamos a construir un nuevo tipo de dato, llamado lista doblemente enlazada o dlist. Esta estructura de datos tiene la propiedad de que, en cada momento, se puede acceder tanto al siguiente elemento como al anterior. Para implementarlo, definimos los elementos de esta lista como parejas cuya parte izquierda contiene el dato propiamente dicho y la parte derecha el enlace al elemento anterior. La lista va precedida de una pareja que hace el papel de cabecera y que es referenciada por las variables que apuntan a ella. En la figura puedes ver esta implementación:



**a) (6 puntos)** Define el procedimiento (`make-dlist`) que construya una lista vacía y el procedimiento mutador (`add-dlist! dlist x`) que añada un elemento `x` en la primera posición de una lista creada por `make-dlist`.

**b) (3 puntos)** Define el procedimiento mutador (`insert-dlist! lista pos-dlist x`) que inserte un elemento `x` en la posición de la lista indicada por `pos-dlist`. Suponemos que la posición está contenida en la lista.

Por ejemplo, el siguiente código genera la figura anterior:

```
(define lista (make-dlist))
(add-dlist! lista 4)
(add-dlist! lista 3)
(add-dlist! lista 1)
(insert-dlist! lista 1 2)
```

#### Pregunta 5 (9 puntos)

Estudia el siguiente código en Scala:

```
def foo(lista: List[(Int) => Boolean]) = {
  (x: Int) => {
    var todos = true
    for (f <- lista)
      if (!lista(f)(x)) todos = false
    todos
  }
}
```

**a) (6 puntos)** Explica qué hace la función: qué recibe y qué devuelve, y escribe un ejemplo de código en Scala en el que se utilice la función

**b) (3 puntos)** Tiene un error de compilación. Encuéntralo y corrígetlo.

**Pregunta 6 (8 puntos)**

Supongamos el siguiente código en Scala:

```
def constructor() = {  
    var x = 0  
    def suma(y:Int) = {  
        x = x+y  
        x  
    }  
    def resta(y:Int) = {  
        x = x-y  
        x  
    }  
    def getFunc(s:String) = {  
        s match {  
            case "suma" => suma _  
            case "resta" => resta _  
        }  
    }  
    getFunc _  
}
```

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

f(20) -> 20  
g(10) -> 10

**a) (4 puntos)** Rellena los huecos (una sentencia en cada uno) para que las dos últimas expresiones devuelvan los valores indicados.

**b) (4 puntos)** Dibuja y explica el diagrama de entornos creado.

**Lenguajes y Paradigmas de Programación**  
**Examen de la convocatoria de Diciembre**  
**Curso 2003-2004**

Notas:

1. La duración del examen es de 3 horas
2. Los puntos máximos que se pueden obtener en el examen son 42
3. La puntuación de las prácticas se acumulará a la obtenida en el examen
4. Para aprobar la asignatura es necesario obtener en este examen más de 18 puntos y, junto con las prácticas, más de 30 puntos

**Pregunta 1 (6 puntos).** Dadas estas definiciones:

```
(define (square x) (* x x))
(define (inc) (set! count (+ count 1)) count)
(define count 5)
```

¿Cuál es el valor de la expresión `(square (inc))` bajo un orden de evaluación aplicativo? ¿Cuál es el valor de la misma expresión bajo un orden de evaluación normal?

**Pregunta 2 (6 puntos).** Escribe una función prefijo-a-infijo que tome una expresión aritmética de Scheme como argumento, en forma de árbol infijo, y devuelva una lista con la expresión en la forma de notación usual infija, con los operadores entre los operandos, como esto:

```
> (prefix-to-infix '(+ (* 2 3) (- 7 4)))
((2 * 3) + (7 - 4))
> (prefix-to-infix '(* (remainder 9 2) 5)
((9 remainder 2) * 5))
```

**Pregunta 3 (10 puntos).** Esta pregunta se refiere al tipo abstracto de datos árbol, definido por estos selectores y constructor:

```
(define datum car)
(define children cdr)
(define make-node cons)
```

Escribe el procedimiento de alto nivel `tree-accumulate`. Sus dos argumentos son una función `fn` y un árbol. La función `fn` tomará dos argumentos y será asociativa (esto es, no tienes que preocuparte del orden en el que encuentres los nodos en el árbol). El procedimiento `tree-accumulate` combinará todos los datos del árbol mediante la aplicación de `fn` a ellos de dos en dos, de forma análoga al procedimiento ordinario `accumulate`. Por ejemplo:

```
> (define my-tree
  (make-node 3 (list (make-node 4 '())
                     (make-node 7 '())
                     (make-node 2 (list (make-node 3 '())
   (make-node 8 '()))))))
> (tree-accumulate + my-tree)
27
> (tree-accumulate max my-tree)
8
```

**Pregunta 4 (10 puntos).** Escribe un procedimiento deep-subst! que tome tres argumentos, dos de los cuales son palabras y el tercero es cualquier estructura de lista (cualquier cosa hecha de parejas). El procedimiento debe mutar la estructura de la lista de forma que cualquier ocurrencia de la primer palabra se substituya por la segunda palabra. Ejemplos:

```
> (deep-subst! 'foo 'baz (list (cons 'hello 'goodbye) (cons 'moby 'foo)))
((hello . goodbye) (moby . baz))

> (deep-subst! 'a 'x (list (list 'a 'b 'c) (list 'b 'a 'd)
                           (list 'f 'a 'b)))
((x b c) (b x d) (f x b))
```

La función no debe crear nuevas parejas.

**Pregunta 5 (10 puntos).** Dibuja los diagramas de entornos resultantes de la evaluación de las siguientes expresiones:

```
(define (kons a b)
  (lambda (m)
    (if (eq? m 'kar) a b)))
(define p (kons (kons 1 2) 3))
```

# Lenguajes y Paradigmas de Programación

## Curso 2003-2004

### Examen de la Convocatoria de Diciembre

#### Normas importantes

- La puntuación total del examen es de 40 puntos que sumados a los 20 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 16 puntos en este examen.**
- La duración del examen es de 2 horas.

#### Pregunta 1 (8 puntos)

(6 puntos) Escribe un procedimiento llamado `power-close-to` que tome dos enteros positivos (`b` y `n`) como argumento y devuelva la potencia más pequeña de `b` que es mayor que `n`. Esto es, debería devolver el entero más pequeño `i` tal que  $b^i > n$ .

`(power-close-to 2 100) -> 8`

(2 puntos) ¿El procedimiento que has escrito es recursivo o iterativo?

#### Pregunta 2 (8 puntos)

Suponiendo que `tree` es un árbol cuyos datos son números, define el procedimiento (`suma tree`) que devuelva la suma de todos los datos del árbol `tree`. Debes usar los selectores definidos en clase:

```
(define (make-tree dato hijos) (cons dato hijos))
(define (dato tree) (car tree))
(define (hijos tree) (cdr tree))
```

Por ejemplo, `(suma '(1 (2) (3 (4 (5)))) )` debe devolver 15

#### Pregunta 3 (8 puntos)

oHe aquí un procedimiento para construir un objeto abstracto persona. El procedimiento está incompleto.

```
(define (make-persona nombre edad profesion dni)
  (let ((persona (list nombre edad profesion dni)))
    (lambda (m)
      (cond ((eq? m 'nombre)      <a>)
            ((eq? m 'e-mail)     <b>)
            ((eq? m 'empresa)    <c>)
            ((eq? m 'tfno)       <d>)
            (else (error "petición desconocida" m))))))
```

**(continúa detrás)**

Un ejemplo de uso del procedimiento anterior serían las siguientes instrucciones:

```
(define persona-1
  (make-persona 'lucia 'lgl@yahoo.com 'ua '966454321))
(define persona-2
  (make-persona 'david 'dvg@hotmail.com 'dccia '953441234))
```

1. (4 puntos) Escribe qué sentencias Scheme deberían ir en los lugares marcados con <a>, <b>, <c>, <d>.
2. (4 puntos) ¿Cómo se preguntaría por el dni de una persona? Pon un ejemplo con el objeto persona-2.

#### Pregunta 4 (8 puntos)

Consideramos que las siguientes expresiones se evalúan en el orden que aparecen:

```
(define a (list (list 'q) 'r 's))
(define b (list (list 'q) 'r 's))
(define c a)
(define d (cons 'p a))
(define e (list 'p (list 'q) 'r 's))
```

1. (4 puntos) Dibuja un diagrama box-and-pointer con las estructuras resultantes
2. (4 puntos) Indica cuál sería el resultado de las siguientes expresiones:

|                        |                           |
|------------------------|---------------------------|
| (eq? a c)              | (equal? a c)              |
| (eq? a b)              | (equal? a b)              |
| (eq? a (cdr d))        | (equal? a (cdr d))        |
| (eq? (car a) (cadr e)) | (equal? (car a) (cadr e)) |

#### Pregunta 5 (8 puntos)

Supongamos los siguientes programas Scheme.

|                                                                                |                                                                                    |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| (define z 3)<br>(define h (let ((x 2))<br>(lambda (y)<br>(+ x y z))))<br>(h 4) | (define h<br>(let ((x 2))<br>(lambda (y)<br>(let ((z 3))<br>(+ x y z))))<br>(h 4)) |
|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------|

1. (4 puntos) Dibuja los entornos resultantes después de evaluarse el programa 1 y el 2 (ambos se evalúan por separado)
2. (1 punto) ¿Cuál es el valor que devuelve la última expresión en ambos casos?
3. (3 puntos) Dos entornos son equivalentes cuando todos los programas Scheme dan el mismo resultado en ambos. ¿Son equivalentes los entornos resultantes de evaluar el programa 1 y el 2? Si piensas que los entornos no son equivalentes, encuentra algún programa Scheme que se comporte de forma distinta (devuelva un valor distinto) en uno y otro entorno.

# Lenguajes y Paradigmas de Programación

## Curso 2005-2006

### Examen de la Convocatoria de Diciembre

#### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 2 horas.
- Las fechas de revisión y las notas estarán disponibles en la web de la asignatura el próximo día 16 de diciembre.

#### Pregunta 1 (10 puntos)

Define un procedimiento (`mcd n1 n2`) que calcule el máximo común divisor de dos números:

```
(mcd 60 40)  -> 20
(mcd 5 15)   -> 5
```

#### Pregunta 2 (10 puntos)

Define un procedimiento (`mismo-dato? tree`) que tome un árbol como argumento y devuelva `#t` si existe algún nodo en el árbol que tenga el mismo dato que uno de sus hijos directos (no nieto ni sobrino) y `#f` si no existe.

```
(define t1 (make-tree 2 nil))
(define t2 (make-tree 3 nil))
(define t3 (make-tree 2 (list t1 t2)))
(define t4 (make-tree 1 (list t3)))
(mismo-dato? t4) -> #t

(define t5 (make-tree 1 nil))
(define t6 (make-tree 2 (list t2 t5)))
(define t7 (make-tree 1 (list t6)))
(mismo-dato? t7) -> #f
```

### **Pregunta 3 (10 puntos)**

Define un conjunto de funciones que permitan trabajar con una agenda de contactos. Ten especial cuidado en hacer un código modular, mantenable y reusable.

1. (3 puntos) Diseña las funciones.
2. (4 puntos) Implementalas en Scheme puro (no usar las macros de POO) usando alguna de las técnicas de abstracción de datos vistas en clase.
3. (3 puntos) Explica por qué consideras que las funciones que has desarrollado son modulares, mantenibles y reusables.

### **Pregunta 4 (10 puntos)**

Considera la siguiente función:

```
(define (foo a b c)
  (lambda (m)
    (if (a m) (b m) (c m))))
```

1. (2 puntos) Escribe un ejemplo de llamada a foo en el que el resultado de su evaluación sea 5.
2. (4 puntos) Explica el proceso de evaluación de la llamada anterior usando el **modelo de evaluación de sustitución**.
3. (4 puntos) Explica el proceso de evaluación de la llamada anterior usando el **modelo de evaluación basado en entornos**.

# Examen de Lenguajes y Paradigmas de Programación

## Convocatoria de Junio de 2002

Nombre y apellidos:

### Normas generales

- ?? El examen consta de 2 partes; la primera parte se deberá hacer con el ordenador apagado y en la segunda se podrá usar el ordenador para probar los procedimientos que haya que programar. En ambas partes se pueden usar tanto los apuntes como el libro de la asignatura.
- ?? La nota total de la asignatura será la media entre la nota de prácticas y la nota del examen, siempre que ésta última sea mayor o igual a 3.
- ?? **La duración del examen es de 2 horas**

### Parte 1 (sin ordenador)

**1. (1 punto)** ¿Qué responderá Scheme a las siguientes expresiones?

```
a.- (word 10 (+ 2 3))  
b.- (let ((a 5) (b 3)) ((lambda (x) (* a x)) b))  
c.- ((lambda (p x) (p 2 x)) * 3)  
d.- (append (list 1 2) (cdr (cons 1 nil)))  
e.- (car (list (list 1 2) 3))
```

**2. (1 punto)** Supongamos el siguiente procedimiento

```
(define (multi x)  
  (if (null? x)  
      (lambda (arg) arg)  
      (lambda (arg)  
        ((car x) ((multi (cdr x)) arg))))))
```

Explica qué hace multi y escribe un ejemplo de una llamada a la función junto con el resultado que devolvería.

**3. (1 punto)** Supongamos el siguiente código

```
(define (pairs x lista)  
  (if (null? lista) lista  
      (append (list (cons x (car lista))) (pairs x (cdr lista)))))
```

Explica qué hace pairs y escribe un ejemplo de una llamada a la función junto con el resultado que devolvería.

**4. (1 punto)** Dibuja diagramas box-and-pointer que representan los resultados de las siguientes expresiones

```
(define x1 (list (cons 'a 'b) (list 'c 'd)))  
(define x2 (cons 'a (cons 'b (cons 'c 'd))))  
(define x3 (list 'a 'b 'c (cons 'd '()))))  
(set-cdr! x3 (cdr x1)))
```

## Parte 2 (con ordenador)

**5. (1,5 puntos)** Escribe una función test que tome como argumentos un número y una lista de números y devuelva #t si y sólo si existe alguna pareja de números de la lista que sume lo mismo que el primer argumento.

Por ejemplo:

```
(test 5 (1 4 5 8 3))  
#t  
(test 5 (2 4 5 8 3))  
#f
```

**6. (1,5 puntos)** Imagina que tienes una versión de Scheme en la que solo están disponibles los números enteros y que, sin embargo, quieras poder representar cantidades de dinero como 3,95 €. Decides crear un tipo abstracto de datos con dos componentes, llamados euros y cents. Escribe los constructores y selectores:

```
(define (make-price e c) (+ (* e 100) c))  
(define (euros p) (quotient p 100))  
(define (cents p) (remainder p 100))
```

- Escribe el procedimiento +price que tome dos precios como argumento y devuelva su suma como un nuevo precio. *Respetá la abstracción de datos.*
- Ahora queremos cambiar la representación interna de forma que en lugar de representar el precio como un número de céntimos lo representemos como una pareja de dos números. Escribe los constructores y los selectores necesarios, respetando la abstracción de datos.

**7. (1'5 puntos)** Recuerda la forma de definir un constructor de contadores en el que se crea una variable “de clase” que mantiene la suma de todos los contadores y una variable “de instancia” que guarda el valor del contador:

```
(define make-count  
  (let ((glob 0))  
    (lambda ()  
      (let ((loc 0))  
        (lambda ()  
          (set! loc (+ loc 1))  
          (set! glob (+ glob 1))  
          (list loc glob))))
```

Escribe una versión modificada de make-count en la que la variable del clase glob cuente el número de contadores que se han creado.

**8. (1,5 puntos)** Implementa, usando variables locales y paso de mensajes, las funciones de manejo de árboles que hemos visto en teoría:

```
(make-arbol dato hijos)  
(dato nodo)  
(hijos nodo)  
(hoja? nodo)
```

Usando esas funciones, escribe una función (contar-hojas arbol) que cuente el número de hojas de un árbol.

# Lenguajes y Paradigmas de Programación

## Curso 2002-2003

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen son 42 puntos que sumados a los 18 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener más de 18 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.

#### Pregunta 1 (3 puntos)

Escribe un procedimiento `add-numbers` que tome una frase como argumento. Algunas de las palabras en la frase pueden ser números. El procedimiento deberá devolver la suma de estos números, por ejemplo:

```
> (add-numbers '(8 dias 1 semana))  
9  
  
> (add-numbers '(76 trombones 4 flautas y 2 trompetas))  
82  
  
> (add-numbers '(all you need is love))  
0
```

#### Pregunta 2 (5 puntos)

Rellena los espacios en blanco para obtener el resultado esperado:

a) (2 puntos)

```
> ((lambda (x y z) (x (_____) z)) * + 3)  
> 15
```

b) (3 puntos)

```
> (((lambda (a b) b) + _____) 3 5)  
> 15
```

### Pregunta 3 (4 puntos)

Escribe un procedimiento (`(pref wd char)`) que tome como argumentos una palabra `wd` y un carácter `char` y que devuelva el prefijo de la palabra `wd` hasta llegar al carácter `char`.

Ejemplo:

```
> (pref "hola" "l")
> "ho"
> (pref "pepito" "i")
> "pep"
> (pref "hola" "i")
> "hola"
```

El procedimiento que has escrito es ¿recursivo o iterativo?. Explica la respuesta.

### Pregunta 4 (3 puntos)

Vamos a crear un TAD para representar una fecha. El constructor (`(make-fecha dia mes año)`) tendrá tres argumentos: día mes año.

Tendremos cuatro operadores:

`(año fec)` -> devuelve el año de la fecha pasada por parámetro.  
`(mes fec)` -> devuelve el mes de la fecha pasada por parámetro.  
`(dia fec)` -> devuelve el dia de la fecha pasada por parámetro.  
`(fin-mes fec)` -> devuelve true si es el último día del mes de la fecha `fec` (no tener en cuenta años bisiestos).

- Implementar `make-fecha`, `año`, `mes`, `dia` y `fin-mes` usando una representación interna de una lista de tres números. Es decir 02/12/2001 (02 12 2001).
- Implementar `make-fecha`, `año`, `mes`, `dia` y `fin-mes` usando una representación interna en la que la fecha se representa como un entero `((añox10000)+(mesx 100)+dia)`

### Pregunta 5 (5 puntos)

Escribe una función (`(max-levels)`) que recorra una lista de listas (un árbol sin datos en los nodos) y que devuelva el número de niveles de la rama que tiene mayor profundidad.

Por ejemplo

```
> (max-levels '(1 2 3 4))
> 1
> (max-levels '((1 (2) 3)))
> 2
> (max-levels '((1) (2 (3)))))
> 3
```

## Pregunta 6 (4 puntos)

Vamos a hacer un conversor de monedas. Imaginariamente podemos tener una progresión lineal con las conversiones de moneda de este tipo:

... -> dólar -> rupia -> euro -> peseta -> ...  
... -> 1 -> 12 -> 62 -> 166 -> ...

Hemos rellenado una tabla con las conversiones:

```
(put 'convert 'dolar (attach-tag 'rupia 12)) ; 12 rupias son 1 dolar  
(put 'convert 'rupia (attach-tag 'euro 62)) ; 62 euros son 1 rupia  
(put 'convert 'euro (attach-tag 'peseta 166)) ; 166 pesetas son 1 euro  
....
```

Sólo hay una conversión para cada moneda. Si se quiere convertir de una moneda a otra que no sea correlativa, se deberá convertir una a una hasta llegar al tipo deseado. Por ejemplo, si queremos pasar de rupias a pesetas, debemos convertir rupias a euros y euros a pesetas.

Escribe el procedimiento `conversion` que tenga dos argumentos, `desde` y `hasta`, y deberá devolver un número que represente la conversión de la moneda `desde` a la moneda `hasta`.

```
>(conversion 'dolar 'rupia)  
>12  
>(conversion 'dolar 'euro)  
>744
```

## Pregunta 7 (7 puntos)

Queremos definir en POO una clase llamada **animal**, que tendrá como variable las vidas que tiene un animal y que por defecto valdrá 1. También definiremos las clases **gato** y **perro**, teniendo en cuenta que una gato tiene 7 vidas. Implementar un método que nos devuelva la cantidad de animales que se han creado, otro que nos devuelva la cantidad de perros y otro para la cantidad de gatos. Además implementaremos un método que se llamará **finVida** que restará una vida al animal al que se le aplique, de forma que cuando no le queden más vidas visualizará un cero. También implementaremos un método **estado** que devuelva el número de vidas de un animal. Por último podremos jugar a ser "DIOS" y redefinir el número de vidas de los animales que queramos, con el método **cambioVidas**. Este método cambia el contador de vidas del animal al número que queramos, siempre que ese número no sea mayor que la cantidad máxima de vidas que tiene el tipo de animal (1 para perro y 7 para gato).

### Pregunta 8 (6 puntos)

Explica la evaluación de las siguientes expresiones y dibuja el entorno resultante.

```
(define m
  (lambda (x)
    (lambda (y)
      (+ x y)))))

(define k (m 5))

(k 7)
```

### Pregunta 9 (5 puntos)

Queremos implementar el procedimiento `swap-cars!` que tome dos listas como parámetros e intercambie entre ellas el primer elemento de cada una. Por ejemplo:

```
>(define pares (list 2 4 6))
>(define impares (list 1 3 5))
>(swap-cars! pares impares)
>pares
>(1 4 6)
>impares
>(2 3 5)
```

(a) (3 puntos) ¿Cuál o cuáles de los siguientes procedimientos funcionan correctamente? Rodea SI o NO en cada uno. Si rodeas el NO, razona por qué en la casilla de la derecha.

|    |                                                                                                                           |  |
|----|---------------------------------------------------------------------------------------------------------------------------|--|
| SI | (define (swap-cars! x y)   (let ((temp x))     (set! x (cons (car y) (cdr x)))     (set! y (cons (car temp) (cdr y )))))) |  |
| NO | (define (swap-cars! x y)   (let ((temp (car x)))     (set! (car x) (car y))     (set! (car y) temp)))                     |  |
| SI | (define (swap-cars! x y)   (set-car! x (car y))   (set-car! y (car x))))                                                  |  |
| NO | (define (swap-cars! x y)   (let ((temp (cons (car x) (cdr x))))     (set-car! x (car y))     (set-car! y (car temp))))    |  |

(b) (2 puntos) Rellena el espacio en blanco del siguiente procedimiento para que `swap-cars!` funcione correctamente:

```
(define (swap-cars! X y)
  (set! x (cons (car y) x))
  (set-car! y (cadr x))
  ;; Rellena el espacio en blanco con una
  ;; llamada a set!, set-car! o set-cdr!
  (_____))
```

# Lenguajes y Paradigmas de Programación

## Curso 2003-2004

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 40 puntos que sumados a los 20 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 16 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.

#### Pregunta 1 (5 puntos)

Queremos un procedimiento (`haz-palindromo wd`) que tome una palabra `wd` y que devuelva un palíndromo formando a partir de la palabra `wd`. Escribe el procedimiento e indica si es recursivo o iterativo.

Ejemplos:

```
(haz-palindromo "hola") -> "holaaloh"  
(haz-palindromo "arroz")-> "arrozzorra"  
(haz-palindromo "") -> ""
```

#### Pregunta 2 (5 puntos)

Rellena los huecos en las siguientes expresiones de Scheme para que su evaluación devuelva los resultados indicados. No es necesario que en los huecos haya un único dato, puede haber una expresión compuesta.

a)

```
(define g  
  (lambda (_____ )  
    (lambda (_____ b) (_____ b)))  
  
((g 3) - 5)  
-2
```

b)

```
(define object
  (let ((init 0))
    (_____ (new)
      (let ((temp init))
        (_____ temp)))))

(object 7)
0
(object 1)
7
(object 4)
1
```

c) Indica cuál es el resultado de evaluar las siguientes expresiones de Scheme.

```
(define (echo f n)
  (if (= n 0)
      (lambda (x) x)
      (lambda (x) (f ((echo f (- n 1)) x)))))

((echo (lambda (x) (* x x)) 2) 2)
```

---

### Pregunta 3 (6 puntos)

Escribe un procedimiento (producto-diagonal matriz) que calcule el producto de la diagonal principal de una matriz cuadrada matriz. Suponemos que la matriz cuadrada se representa como una lista de listas planas en la que cada lista plana representa una fila de la matriz. Puedes definir procedimientos auxiliares que uses en el procedimiento principal.

Ejemplos:

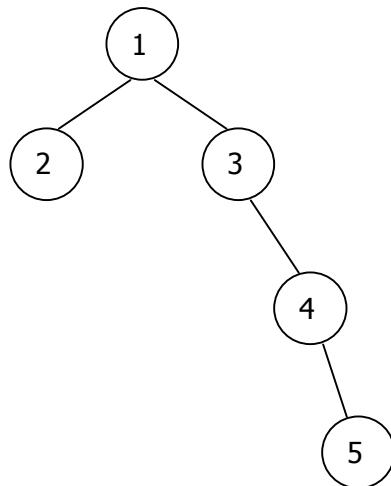
```
(producto-diagonal '((2 3)
                     (4 5)))
10

(producto-diagonal '((2 4 6)
                     (8 10 12)
                     (14 16 18)))
360
```

#### Pregunta 4 (6 puntos)

Define el procedimiento (`max-hijos tree`) que vaya recorriendo los nodos del árbol `tree`, calculando el número de hijos de cada nodo y que devuelva el máximo número de hijos encontrado en un único nodo.

Por ejemplo, el árbol `'(1 (2) (3 (4 (5))))'` se corresponde con la siguiente figura:



En este árbol tenemos los siguientes números de hijos:

Nodo '1' tiene 2 hijos

Nodo '2' tiene 0 hijos

Nodo '3' tiene 1 hijo

Nodo '4' tiene 1 hijo

Nodo '5' tiene 0 hijos

Por lo tanto, el máximo número de hijos corresponde al nodo '1', con 2 hijos. Éste número es el que debe devolverse.

Ejemplos:

```
(max-hijos '(1 (2) (3) (4)))
3
(max-hijos '(1 (2) (3 (4 (5))))))
2
```

#### Pregunta 5 (6 puntos)

Queremos definir con la extensión de Programación Orientada a Objetos de Scheme una jerarquía de clases que nos permita mantener la información sobre el reparto de paquetes de una empresa de mensajería a contra-reembolso.

Definimos una clase llamada **Paquete** que tendrá como variables de instanciacción el **nombre del mensajero** que se encargará de su reparto y el **precio del objeto** a entregar.

Además se guardará la información del estado en que se encuentra un paquete. Existirán tres posibles valores:

- 0: perfecto
- 1: leve deterioro
- 2: deteriorado

Por último, existen dos tipos distintos de paquetes: **Sobre** y **Caja**. El transporte de un sobre tiene un **coste de transporte** de 2 euros y el de una caja de 6 euros siempre que el paquete se entregue en estado perfecto.

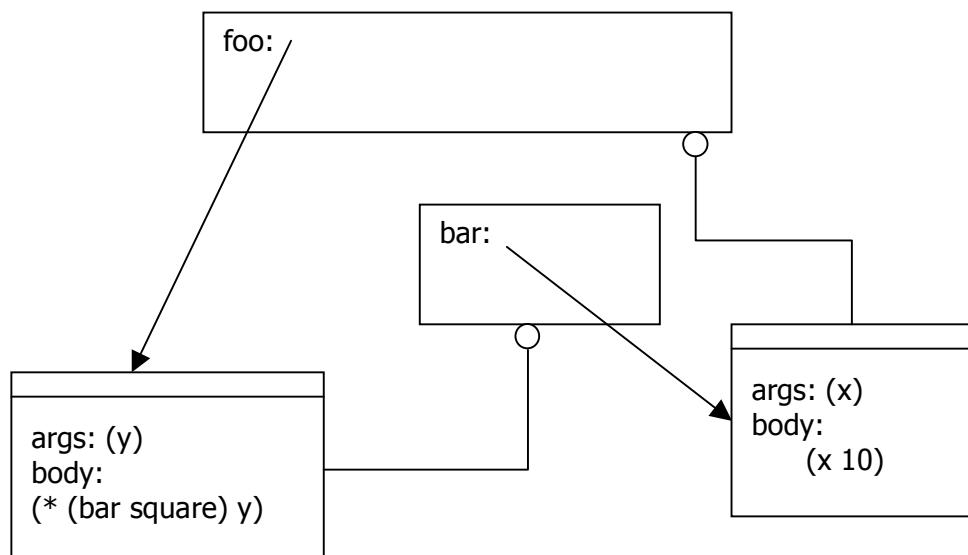
Necesitamos los siguientes métodos:

- num-paquetes, que devuelve el número de paquetes creados
- num-sobres, que devuelve el número de sobres
- num-cajas, que devuelve el número de cajas
- cambia-mensajero, que permite cambiar el mensajero asignado a un paquete
- cambia-estado, que permite cambiar el estado de un paquete. Cuando un paquete está deteriorado ese será su estado definitivo y el estado no se podrá modificar.
- importe, que devuelve el precio a cobrar por la entrega de un paquete. El precio a cobrar es el coste de transporte más el precio del objeto, si el estado es 0 (perfecto). Si el estado es 1, el coste de transporte se reduce en un 50% y si el estado es 2 el coste de transporte es 0 y el importe a cobrar es el precio del objeto.

Usando la extensión de POO de Scheme, implementa un conjunto de clases que resuelvan este problema y escribe un ejemplo de uso.

### Pregunta 6 (6 puntos)

Escribe una única instrucción en Scheme, que no use mutación, que genere el siguiente modelo de entorno:



### Pregunta 7 (6 puntos)

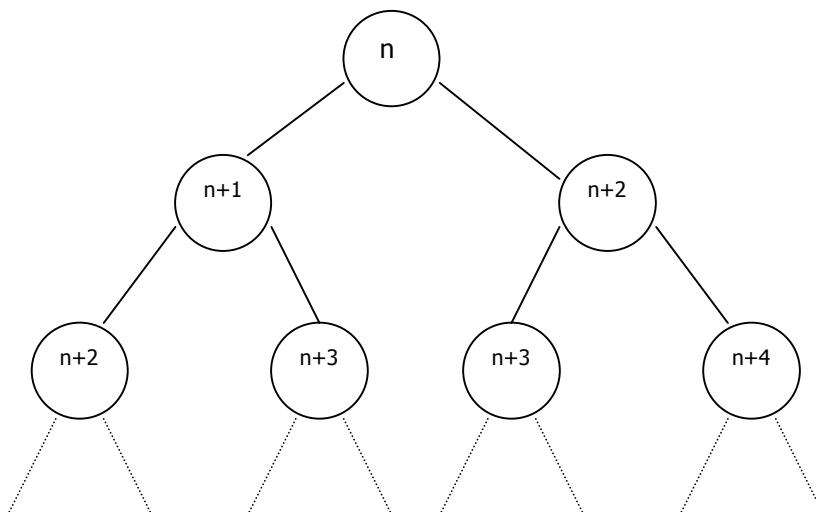
Un árbol binario infinito es una estructura que contiene un dato, un hijo a la izquierda y un hijo a la derecha, los cuales son a su vez árboles binarios infinitos. Es decir, un árbol binario infinito es un árbol cuyos nodos tienen dos hijos y un profundidad infinita (infinitos niveles).

- a) Rellena los huecos para completar la definición del árbol binario infinito y de sus selectores. Supongamos que la llamada a `def-macro` en la siguiente expresión permite definir una nueva macro de Scheme. En este caso es necesario definir `make-inf-tree` como una macro y no como una función porque, de forma similar a lo que sucede con los streams, *no queremos que se evalúen los argumentos* cuando se haga la llamada a `make-inf-tree` para construir el árbol infinito.

```
(def-macro (make-inf-tree dato izq der)
           (list dato _____))

(define (dato tree) (_____))
(define (hijo-izq tree) (_____))
(define (hijo-der tree) (_____))
```

- b) Utilizando la estructura de árbol binario infinito, define el procedimiento `(make-n-tree n)` que reciba un número `n` como parámetro y devuelva el siguiente árbol binario infinito:



# Lenguajes y Paradigmas de Programación

## Curso 2004-2005

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas estarán disponibles en la web de la asignatura el próximo día 1 de Julio.

#### Pregunta 1 (8 puntos)

a) (5 puntos)

Supongamos el siguiente procedimiento:

```
(define (h n)
  (lambda (x y)
    (set! x (+ (car n) (cdr n) y))
    (set-car! n x)
    (set-cdr! n y)
    x))
```

Escribe un ejemplo completo (en el que se llegue a evaluar el cuerpo del lambda) de uso del procedimiento, incluyendo cuáles serían los resultados devueltos por el intérprete.

b) (3 puntos)

Supongamos las siguientes expresiones:

```
(h (g 3) 5)  ->  8
(h (g 12) 5)  -> 17
```

Completa las siguientes definiciones de forma que la llamada (g x) devuelva un procedimiento:

```
(define (g x)
  _____)
```

```
(define (h f z)
  _____)
```

## Pregunta 2 (8 puntos)

Define un procedimiento (calculadora expr) que tome una lista con la estructura (número op número op... número), donde - y / son las únicas operaciones permitidas, y calcule el resultado de la operación. Ten en cuenta que la división tiene mayor precedencia que la resta. Por ejemplo, (10 - 9 / 3) se evalúa como 10 - (9 / 3) y no como (10 - 9) / 3. Se asume que la lista nunca estará vacía.

Ejemplos:

```
(calculadora '(4 - 2)) -> 2  
(calculadora '(10 - 20 / 2 / 5)) -> 8
```

## Pregunta 3 (8 puntos)

Hemos definido la siguientes clases utilizando la extensión de Scheme para POO:

```
(define-class (electrodomestico marca precio)  
  (class-vars (cantidad 0))  
  (initialize (set! cantidad (1+ cantidad)))  
  (method (apagar) ...)  
)  
  
(define-class (vitroceramica marca precio)  
  (parent (electrodomestico marca precio))  
  (method (cambiarPrecio p) (set! precio p))  
  (method (enciendeFogon numeroFuego intensidad) ....)  
  (method (apagar tiempo) ...)  
 ;este método apaga el aparato con un retardo de tiempo minutos  
 ...  
)
```

a) (3 puntos) Supongamos que se evalúan las siguientes expresiones:

```
(define vitro (instantiate vitroceramica 'aeg 200))  
(ask vitro 'cambiarPrecio 500)
```

Se desea obtener el precio original (200) de *vitro*. ¿Cuáles serían las respuestas correctas?

- a) Definiendo un método que devuelva la variable de instanciacción *precio*
- b) Almacenando el precio original en una variable de instancia que posteriormente devolveremos
- c) Definiendo el método: (method (precio-original) (usual 'precio))
- d) Ninguna de las anteriores

b) (3 puntos) Indica cuales de las siguientes afirmaciones son ciertas y cuales falsas:

- a) Cada instancia de una clase tiene su propia versión de las variables de instanciacción
- b) Las variables de instancia se definen como argumentos cuando la instancia se crea
- c) Cuando se envía un mensaje a un objeto para el que no hay definido un

método ejecuta el del hijo si éste existiera

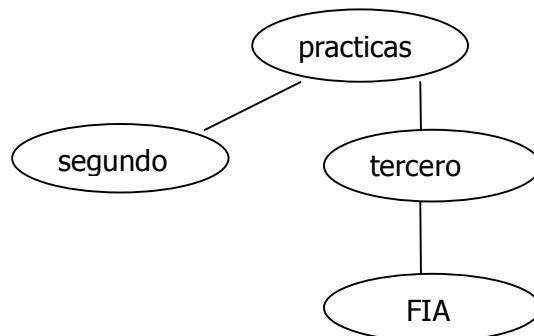
d) La cláusula *initialize* permite modificar las variables de instancia cada vez que se crea una nueva instancia

c) (2 puntos) Se desea agregar un método a la clase `vitrocerámica` para que se apague instantáneamente. Ya dispone de un método con un retardo de x minutos. ¿Qué opciones serían las correctas?

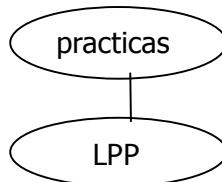
- a) (method (apagateYa) (ask self 'apagar 0))
- b) (method (apagateYa) (ask self 'apagar))
- c) (method (apagateYa) (usual 'apagar))
- d) Las respuestas a) y c) son correctas.

#### Pregunta 4 (9 puntos)

Vamos a representar estructuras de archivos mediante árboles genéricos. Por ejemplo:



a) (5 puntos) Para construir esta estructura, definimos la función `make-rama` que toma una lista con una ruta como argumento y devuelve el árbol que la representa. Por ejemplo, la llamada `(make-rama '(prácticas LPP))` debería devolver el árbol siguiente:



El procedimiento `make-rama` podría definirse como:

```
(define (make-rama ruta)
1 (if (null? ruta)
2   '()
3   (make-tree      ; en esta implementación, make-tree = cons
4         (car ruta)
5         (make-rama (cdr ruta))))
```

Una llamada a (make-rama `(*practicas* segundo LPP)) debería devolver (*practicas* (segundo (LPP))), sin embargo devuelve (*practicas* segundo LPP). Haz los cambios necesarios para que el procedimiento funcione correctamente. Puede que no sea necesario que cambies todas las líneas.

Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_  
Cambio línea\_\_ por \_\_\_\_\_

b) (4 puntos) Define una función (find-ruta fichero) que devuelva una lista plana con la ruta en la que se encuentra el fichero. Por ejemplo:

```
(find-ruta 'FIA) -> '(practicas tercero FIA)
(find-ruta 'tercero) -> '(practicas tercero)
(find-ruta 'GRAFICOS) -> '()
```

### Pregunta 5 (8 puntos)

Define el procedimiento (intercalar! 11 12) que reciba dos listas como argumento y, utilizando mutadores, modifique la lista original 11 e intercale los elementos de ambas. Obviamente, la lista 12 también quedará modificada.

Ejemplo:

```
(define 11 '(1 2 3 4 5))
(define 12 '(a b c d e))
(intercalar! 11 12)
11 -> (1 a 2 b 3 c 4 d 5 e)
12 -> (a 2 b 3 c 4 d 5 e)
```

### Pregunta 6 (9 puntos)

Supongamos las siguientes expresiones:

```
(define (h z)
  (let ((x z) (y (* 2 z)))
    (lambda (v)
      (set! z v)
      (+ v z x y))))
(define g (h 3))
(g 2)
```

- a) (4 puntos) Dibuja el entorno resultante de evaluar las expresiones anteriores  
b) (5 puntos) Explica paso a paso cómo se han evaluado las expresiones anteriores

# Lenguajes y Paradigmas de Programación

## Curso 2006-2007

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen es de 60 puntos. Para obtener la nota final (en escala 0-10) se suman los puntos de prácticas y se divide por 6.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 24 puntos en este examen.**
- Se debe contestar cada pregunta **en una hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas (y la fecha de revisión) estarán disponibles en la web de la asignatura el próximo día 2 de Julio.

#### Pregunta 1 (5 puntos)

Contesta a las siguientes preguntas, rodeando la respuesta que consideres correcta (sólo una es cierta). Cada pregunta vale 1 punto.

1. En programación declarativa ¿es posible definir una referencia?

- (a) Depende del lenguaje de programación.
- (b) No, porque en programación declarativa las variables sólo son nombres que representan valores, similares a constantes.
- (c) No, porque la programación declarativa está basada en la declaración de funciones.
- (d) Sí, por ejemplo cuando damos valor a un parámetro de una función.

2. Un modelo de computación proporciona:

- (a) Una explicación del funcionamiento de los programas de un lenguaje de programación.
- (b) Una forma de determinar si un programa es correcto o no.
- (c) Unas reglas que analizan sintácticamente un programa.
- (d) Un mecanismo de ejecución virtual de cualquier lenguaje de programación de un paradigma.

3. En el modelo de computación basado en entornos, se crea un entorno nuevo:

- (a) Cada vez que se llama a una función primitiva.
- (b) Cada vez que se llama a un procedimiento definido por el usuario.
- (c) Cada vez que se llama a un “define”.
- (d) Cada vez que se llama a un “lambda”.

4. Una de las ventajas principales de la utilización de barreras de abstracción es que:

- (a) Los programas son más rápidos.
- (b) Los programas se pueden ejecutar en distintos sistemas operativos.

- (c) Se oculta la implementación y se acerca el lenguaje de programación al dominio que se está programando.
- (d) El funcionamiento de los programas se puede comprobar automáticamente.

5. Una de las ventajas principales de los lenguajes fuertemente tipados es que:

- (a) Los entornos de programación pueden detectar más errores mientras el programador va escribiendo el programa.
- (b) Los programas son más eficientes.
- (c) Es posible definir polimorfismo.
- (d) Una función puede devolver más de un tipo de datos.

### Pregunta 2 (5 puntos)

1. Dadas la siguiente expresión:

a) Dibuja el diagrama *box & pointer* correspondiente.

b) Escribe la salida por pantalla que produce Scheme después de la evaluación.

```
(define z
  (let ((y '(7)))
    (let ((y '(8))
          (x (list 'a '(b c) (car y))))
      (set-car! x (car y))
      (set-cdr! x (car (cdr x)))
      x)))
```

### Pregunta 3 (10 puntos)

a) Implementa la función (`split lista n`) que divide una lista en dos por el elemento  $n$ -ésimo **sin utilizar mutadores**. El resultado es una pareja cuyo `car` es la primera sublistas y cuyo `cdr` es la segunda. Ejemplo:

```
(define lista '(1 2 3 4 5))
(define pareja (split lista 3)
(car pareja) -> '(1 2 3)
(cdr pareja) -> '(4 5)
lista -> '(1 2 3 4 5))
```

b) Implementa la función (`split! lista n`) que haga lo mismo que `split` pero **utilizando mutadores**. Ejemplo:

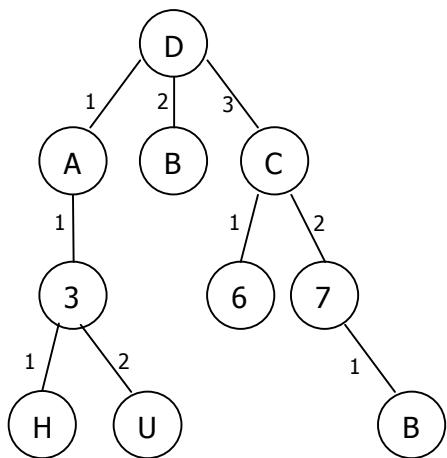
```
(define lista '(1 2 3 4 5))
(define pareja (split lista 3)
(car pareja) -> '(1 2 3)
(cdr pareja) -> '(4 5)
lista -> '(1 2 3))
```

#### Pregunta 4 (10 puntos)

Queremos un procedimiento que nos permita acceder de forma indexada a los elementos de un árbol.

Escribe un procedimiento `tree-ref` que tome un árbol genérico y una lista de índices (números enteros mayores o igual a 1) como argumentos. La secuencia de índices muestra el **camino** a seguir para encontrar un elemento en el árbol.

Por ejemplo, considerando el árbol que se muestra a continuación, la llamada a `(tree-ref tree '(3 1))` devuelve 6. El primer índice, el 3, selecciona el subárbol que se encuentra en la tercera rama del árbol, y el índice 1 selecciona el subárbol que está en la primera rama de ese subárbol. Devuelve el dato que se encuentra en el subárbol indexado (el 6).



Más ejemplos:

```
(tree-ref tree '(1 1 1))
H
(tree-ref tree '(3 2))
7
(tree-ref tree '(1))
A
```

Suponemos que la secuencia de índices siempre es correcta, es decir, proporciona un camino existente en el árbol.

## Pregunta 5 (10 puntos)

Supongamos que necesitamos mantener simultáneamente dos implementaciones del tipo de dato `rectangulo2D`, que llamaremos `rect-base-altura` y `rect-coords`.

En la primera implementación representamos un rectángulo por la posición (x e y) de su esquina inferior izquierda y por su base y altura.

En la segunda implementación lo representamos mediante la posición de su esquina inferior izquierda y de su esquina superior derecha.

a) (5 puntos) Define la barrera de abstracción de los tipos específicos `rect-base-altura` y `rect-coords` (un mínimo de 3 funciones), indicando el nombre de cada función, sus argumentos y su comportamiento. Define una barrera de abstracción común con un mínimo de 3 funciones genéricas (polimórficas) que admitan cualquier tipo específico de rectángulo, describiendo su comportamiento.

b) Define una implementación completa de las funciones anteriores, utilizando el enfoque de paso de mensajes.

## Pregunta 6 (10 puntos)

Estamos diseñando una aventura mediante PDD. Se trata de un laberinto formado por un conjunto de habitaciones unidas por pasadizos. Un ejemplo de utilización (en negrita aparece lo que el jugador escribe por teclado) sería:

### (aventura)

Te encuentras en la entrada de una cueva. Hay un túnel hacia el este.

Movimiento? **este**

Te encuentras en una cueva amplia. Las paredes brillan. Hay túneles al este y al sur. Hay una varita mágica en el suelo.

Movimiento? **coger varita**

Aparece una nube de humo. Te encuentras ahora en una habitación con paredes de oro. Un frasco con una poción verde está en el suelo. Hay unas puertas al norte y al sur.

Movimiento?

...

Y así sucesivamente. El juego está programado de la siguiente forma:

```
(put 'entrada 'descripcion "Te encuentras en la entrada de una cueva.  
Hay un túnel hacia el este")  
(put 'entrada 'este (lambda () (visit 'cueva)))  
(put 'cueva 'descripcion "Te encuentras en una cueva amplia. Las  
paredes brillan. Hay túneles al este y al sur. Hay una varita mágica  
en el suelo")  
(put 'cueva 'este (lambda () (visit 'camara-tortura)))  
(put 'cueva 'sur (lambda () (visit 'biblioteca)))  
(put 'varita 'coger (lambda () (visit 'sala-oro)))  
(put 'varita 'agitar (lambda () (visit 'biblioteca)))  
...  
...
```

Tenemos los siguientes procedimientos (suponemos que `read-line` es una primitiva que devuelve la petición que el jugador escribe en forma de lista de identificadores):

```
(define (aventura)
  (visit 'entrada))

(define (visit habitacion)
  (newline)
  (display (get habitación 'descripcion))
  (newline)
  (display "Movimiento? ")
  (actua habitacion (read-line)))
```

Implementa el procedimiento `actua` que reciba la habitación donde se encuentra el jugador y una petición como argumento.

Se considera que todo lo que el jugador teclea es correcto.

### Pregunta 6 (10 puntos)

Supongamos las siguientes expresiones en Scheme:

```
(define (g x)
  (let ((z 8))
    (lambda (y)
      (set! z (+ x y))
      (set! x (+ z y))
      (+ x y z)))
  (define h (g 4))
  (h 6))
```

- a) (2 puntos) ¿Qué valor devolverá la última expresión?
- b) (5 puntos) Dibuja y explica el diagrama de entornos creado al ejecutar las expresiones.
- c) (3 puntos) ¿Tiene la función `g` estado local? ¿Tiene la función `h` estado local? ¿Qué variables contienen el estado local?

# Lenguajes y Paradigmas de Programación

Curso 2010-2011

Examen Final de la convocatoria de junio

---

## Normas importantes

- La puntuación total del examen es de 50 puntos sobre los 100 que se valora la nota de la asignatura.
  - Se debe contestar **cada pregunta en una hoja distinta**. No olvides poner el nombre en todas las hojas.
  - La duración del examen es de 3 horas.
  - Las notas estarán disponibles en la web de la asignatura el viernes 17 de junio.
  - La revisión será el lunes 20 de junio a las 10:00h en el despacho de Domingo Gallardo.
- 

## Pregunta 1 (8 puntos)

**a) (3 puntos)** Describe las características principales de los siguientes paradigmas de programación, e indica dos lenguajes de programación que se puedan ubicar en cada uno de los paradigmas:

- Paradigma funcional
- Paradigma declarativo
- Paradigma imperativo
- Paradigma orientado a objetos

**b) (3 puntos)** Define y escribe un ejemplo en Scala de los siguientes conceptos:

- Coercion
- Sobrecarga
- Polimorfismo en tiempo de ejecución

**c) (2 puntos)** Dado el siguiente ejemplo:

```
abstract class Bar { def bar(x: Int) : Int }
class Foo extends Bar { def bar(x: Int) = x }

trait Foo1 extends Foo { abstract override def bar(x: Int) = super.bar(x * 3) }
trait Foo2 extends Foo { abstract override def bar(x: Int) = super.bar(x + 3) }
trait Foo3 extends Foo { abstract override def bar(x: Int) = x + super.bar(x) }
trait Foo4 extends Foo { abstract override def bar(x: Int) = x * super.bar(x) }
```

Rellena los huecos con el resultado de las siguientes llamadas y explica tu respuesta:

```
scala> (new Foo with Foo2 with Foo4).bar(5)
scala> _____
```

```
scala> (new Foo with Foo2 with Foo1).bar(5)
scala> _____
```

```
scala> (new Foo with Foo3 with Foo4).bar(5)
scala> _____
```

## Pregunta 2 (7 puntos)

Supongamos que estamos desarrollando en Scheme un programa de estadística y que guardamos una lista de frecuencias (número de veces que se repite un valor) como una lista de parejas (`<valor> <veces>`). Por ejemplo, la lista ((1.3) (2.4) (4.8)) representa que el valor 1 ha aparecido 3 veces, el 2 4 veces y el 4 8 veces.

La media de estos valores se puede expresar como

$$(v_1 * n_1 + v_2 * n_2 + \dots + v_n * n_n) / (n_1 + n_2 + \dots + n_n),$$

siendo  $v_i$  el valor  $i$ -ésimo y  $n_i$  su frecuencia. Por ejemplo, en el caso anterior, la media de los valores es:

$$(1*3+2*4+4*8)/3+4+8 = 2,87$$

Define una función recursiva (`frecuencias lista-freccs`) que devuelva una pareja donde la parte izquierda corresponda a  $(v_1 * n_1 + v_2 * n_2 + \dots + v_n * n_n)$  y la parte derecha a  $(n_1 + n_2 + \dots + n_n)$ . Utiliza esta función para calcular la media. Ejemplo:

```
> lista-frec
((1 . 3) (2 . 4) (4 . 8))
> (frecuencias lista-frec)
(43.15)
> (media lista-frec)
2,87
```

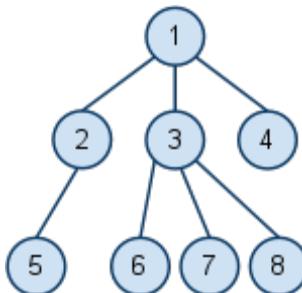
## Pregunta 3 (9 puntos)

**a) (3 puntos)** Define en Scheme la barrera de abstracción del árbol genérico y utilízala para implementar el procedimiento (`equal-tree? tree1 tree2`) que reciba dos árboles genéricos como argumento y devuelva #t si son iguales y #f en caso contrario.

**b) (6 puntos)** Vamos a implementar en Scala el tipo Tree (como un árbol genérico). Su implementación es:

```
class Tree (val dato: Int, val hijos: List[Tree])
```

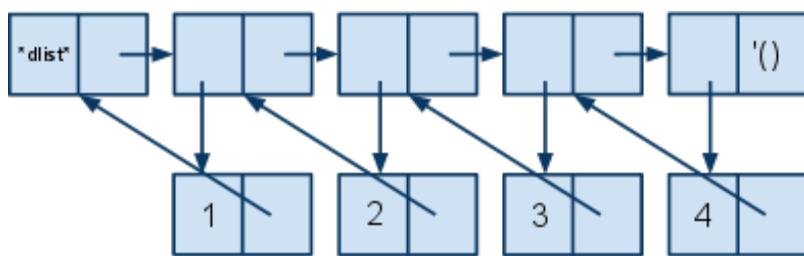
**b.1)** Define el siguiente árbol utilizando el tipo de dato Tree:



**b.2)** Añade al tipo Tree el método `sumaTotalTree` que reciba un Tree como argumento y devuelva la suma de todos sus nodos.

#### Pregunta 4 (9 puntos)

Vamos a construir un nuevo tipo de dato, llamado lista doblemente enlazada o dlist. Esta estructura de datos tiene la propiedad de que, en cada momento, se puede acceder tanto al siguiente elemento como al anterior. Para implementarlo, definimos los elementos de esta lista como parejas cuya parte izquierda contiene el dato propiamente dicho y la parte derecha el enlace al elemento anterior. La lista va precedida de una pareja que hace el papel de cabecera y que es referenciada por las variables que apuntan a ella. En la figura puedes ver esta implementación:



**a) (6 puntos)** Define el procedimiento (`make-dlist`) que construya una lista vacía y el procedimiento mutador (`add-dlist! dlist x`) que añada un elemento `x` en la primera posición de una lista creada por `make-dlist`.

**b) (3 puntos)** Define el procedimiento mutador (`insert-dlist! lista pos-dlist x`) que inserte un elemento `x` en la posición de la lista indicada por `pos-dlist`. Suponemos que la posición está contenida en la lista.

Por ejemplo, el siguiente código genera la figura anterior:

```
(define lista (make-dlist))
(add-dlist! lista 4)
(add-dlist! lista 3)
(add-dlist! lista 1)
(insert-dlist! lista 1 2)
```

#### Pregunta 5 (9 puntos)

Estudia el siguiente código en Scala:

```
def foo(lista: List[(Int) => Boolean]) = {
  (x: Int) => {
    var todos = true
    for (f <- lista)
      if (!lista(f)(x)) todos = false
    todos
  }
}
```

**a) (6 puntos)** Explica qué hace la función: qué recibe y qué devuelve, y escribe un ejemplo de código en Scala en el que se utilice la función

**b) (3 puntos)** Tiene un error de compilación. Encuéntralo y corrígetlo.

**Pregunta 6 (8 puntos)**

Supongamos el siguiente código en Scala:

```
def constructor() = {  
    var x = 0  
    def suma(y:Int) = {  
        x = x+y  
        x  
    }  
    def resta(y:Int) = {  
        x = x-y  
        x  
    }  
    def getFunc(s:String) = {  
        s match {  
            case "suma" => suma _  
            case "resta" => resta _  
        }  
    }  
    getFunc _  
}
```

\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

f(20) -> 20  
g(10) -> 10

**a) (4 puntos)** Rellena los huecos (una sentencia en cada uno) para que las dos últimas expresiones devuelvan los valores indicados.

**b) (4 puntos)** Dibuja y explica el diagrama de entornos creado.

# **Lenguajes y Paradigmas de Programación**

## **Curso 2002-2003**

### **Examen de la Convocatoria de Septiembre**

#### **Normas importantes**

- La puntuación total del examen son 42 puntos que sumados a los 18 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener más de 18 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- **Las notas estarán disponibles el viernes 12 y las revisiones se realizarán el martes 16 de septiembre de 10:00 a 12:00.**

#### **Pregunta 1 (6 puntos)**

Define un procedimiento llamado set-double! que reciba un árbol numérico de cualquier profundidad como parámetro y reemplace cada hoja por el doble de cada número. El método debe recorrer cada elemento del árbol.

```
>(define arbol (list 2 (list 3 4) (list 2 3 (list 3 3))))  
>arbol  
>(2 (3 4) (2 3 (3 3)))  
>(set-double! arbol)  
>arbol  
>(4 (6 8) (4 6 (6 6)))
```

#### **Pregunta 2 (6 puntos)**

Representa mediante diagramas box & pointer el resultado de evaluar las siguientes expresiones:

```
a.- (cadadr '(a (b (cd ef) d)))  
b.- (define lista (list (cons (append '(a) '(b)) '(c))))  
c.- (set-car! (cdar lista) (caar lista))
```

#### **Pregunta 3 (6 puntos)**

Escribe una función multicompose que tome como argumento una lista de cualquier longitud, cuyos elementos son funciones de un argumento. Debería devolver una función de un argumento que realiza la composición de todas las funciones dadas.

Nota: No usar mutación (set!)

```
>((multicompose (list sqrt sqrt 1+)) 80)  
3  
  
>((multicompose (list first bf)) '(la sinuosa carretera))  
sinuosa
```

## Pregunta 4 (6 puntos)

Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3 PM es la hora 15.

La implementación se hará usando dos representaciones internas diferentes.

- El constructor para las horas del día es `time`. Toma dos argumentos: las horas y los minutos. 4:12 debería ser (`time 4 12`)

- También queremos los siguientes tres operadores:

`(hour t) → devuelve la parte de hora de un time dado`

`(minute t) → devuelve la parte de minutos de un time dado`

`(hour? t) → devuelve #t si el time t es una hora exacta (como 6:00); devuelve #f en cualquier otro caso`

a/ Implementa `time`, `hour`, `minute` y `hour?` usando una representación interna en la que la hora se representa como una lista de dos números. Esto es, 4:12 debería representarse internamente como `(4 12)`

b/ Implementa `time`, `hour`, `minute` y `hour?` usando una representación interna en la que la hora se representa como un entero, a saber, el número de minutos transcurridos desde medianoche. Esto es, 4:12 debería representarse internamente como el número 252 ( $(4 \times 60) + 12$ )

## Pregunta 5 (6 puntos)

Define un procedimiento llamado (sufijo wd char) que admita como argumento una palabra wd y un carácter char y que devuelva el resto de la palabra wd desde que se encuentra el carácter char. En el caso en que la palabra no contenga a char, entonces la función devuelve la cadena vacía.

Indica si el procedimiento que has escrito es recursivo o iterativo y explica por qué.

```
> (sufijo 'hola 'o)
'la
> (sufijo 'resto 'e)
'sto
> (sufijo 'pepita 's)
""
```

### **Pregunta 6 (6 puntos)**

Explica qué valor devuelven las siguientes expresiones:

a/

```
(define foo (lambda (x)
  (if (equal? x 0)
      1
      (* x (foo (- x 1))))))

(foo 3)
```

b/

```
(define (foo3 x)
  (lambda (y) (+ y x)))

(define foo4 (foo3 1))
foo4
```

c/

```
(let ((x 3) (y 4))
  (let ((x 5) (y (+ x 3)))
    (let ((x (+ x 7)))
      (+ x y))))
```

### **Pregunta 7 (6 puntos)**

Explica la evaluación de las siguientes expresiones y dibuja el entorno resultante.

```
(define (foo y)
  (lambda (x)
    (set! y (+ x y))))

(define w (foo 5))
(w 30)
```

# **Lenguajes y Paradigmas de Programación**

## **Curso 2003-2004**

### **Examen de la Convocatoria de Septiembre**

#### **Normas importantes**

- La puntuación total del examen es de 40 puntos que sumados a los 20 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 16 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 2 horas.

#### **Pregunta 1 (8 puntos)**

Escribe una función (palabras-repetidas frase1 frase2) que tome dos frases y devuelva el número de palabras repetidas en ellas. En el caso de que una palabra esté repetida en la misma frase se debe contar en las múltiples ocasiones que aparece

Ejemplos:

```
(palabras-repetidas '(me llamo juan) '(juan es alto))  
1  
(palabras-repetidas '(hola que me me dices) '(me))  
2  
(palabras-repetidas '(hola hola hola adios) '(hola hola))  
6
```

#### **Pregunta 2 (8 puntos)**

Escribe una función denominada (perfecto? numero) que reciba como parámetro un número entero y devuelva true si y sólo si el número es perfecto. Un número es perfecto si es igual a la suma de sus divisores inferiores a él. Por ejemplo, el número 28 es perfecto porque  $28 = 1+7+4+1$  y los divisores de 28 son (exceptuando el propio 28) 14, 7, 4 y 1.

Intenta hacer el ejercicio lo más modular posible, definiendo en su caso funciones auxiliares.

### Pregunta 3 (8 puntos)

Suponemos árboles binarios definidos con la siguiente interfaz

```
(define (make-tree dato izq der)
      (list dato izq der))

(define (dato tree) (car tree))
(define (hijo-izq tree) (car (cdr tree)))
(define (hijo-der tree) (car (cdr (cdr tree))))
```

Define el procedimiento (swap-hijos! tree) que intercambie los hijos del árbol binario pasado como parámetro. El procedimiento debe modificar el árbol mutando sus hijos.

```
(define tree '(1 (2 (3) (4)) (4 (3) (4))))
(swap-hijos! tree)
tree
(1 (4 (3) (4)) (2 (3) (4)))
```

### Pregunta 4 (8 puntos)

Dibuja y explica el modelo de entorno resultante de las siguientes instrucciones Scheme. Numera los entornos en el orden en que se van creando.

```
(define (mystery)
  (let ((baz 1000))
    (define (dispatch msg)
      (cond
        ((eq? msg 'one)
         (begin
           (set! baz (/ baz 2))
           baz))
        ((eq? msg 'two)
         (let ((m (dispatch 'one)))
           (set! baz (* m 4))
           baz)))
        (else
         dispatch)))
  (x 'two))
```

**(EL EXAMEN CONTINÚA DETRÁS)**

### Pregunta 5 (8 puntos)

- a) Define un procedimiento (`scale-stream s f`) que recibe un stream de números `s` y devuelve otro stream con los números de `s` escalados por el factor `f`

```
(display-stream (scale-stream pairs 3))  
6  
12  
18  
24  
30  
...
```

- b) Necesitamos generar un stream de potencias de `x`. Define el procedimiento (`powers x`) para que genere dicho stream. **Pista:** puedes usar el procedimiento anterior `scale-stream`.

```
(display-stream (powers 2))  
1  
2  
4  
8  
16  
32  
...
```

# Lenguajes y Paradigmas de Programación

Curso 2004-2005

## Examen de la Convocatoria de Septiembre

### Normas importantes

- La puntuación total del examen es de 50 puntos que sumados a los 10 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 20 puntos en este examen**.
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 2 horas y 30 minutos.
- Las notas estarán disponibles en la web de la asignatura el próximo día 1 de Julio.

### Pregunta 1 (10 puntos)

A) **(5 puntos)** Define una función (`cumplen-todos pred list`) que devuelva `#t` o `#f` dependiendo de si todos los elementos de la lista `list` cumplen el predicado `pred`. La lista `list` puede ser un pseudo-arbol y contener otras listas. Por ejemplo:

```
(cumplen-todos par? '(2 4 (6 8) 10)) -> #t
(cumplen-todos impar? '(1 3 5 6)) -> #f
(cumplen-todos impar? '()) -> #t
```

B) **(2 puntos)** La función que has definido en el apartado anterior ¿es iterativa o recursiva? ¿Por qué?

C) **(3 puntos)** Supongamos que uno de los elementos de la lista no cumple el predicado ¿se sigue procesando la lista en tu solución? En el caso de que no, perfecto, **ya puedes sumar los 3 puntos de este apartado**. En el caso de que se siga procesando la lista hasta el final, modifica la función para que no lo haga y devuelva `#f` justo en ese instante.

### Pregunta 2 (10 puntos)

Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3:00 PM es la hora 15:00. El constructor para las horas del día es `make-time`. Toma dos argumentos: las horas y los minutos. `4:12` debería ser `(make-time 4 12)` También queremos los siguientes tres operadores:

```
(hour t) → devuelve la parte de hora de un time dado
(minute t) → devuelve la parte de minutos de un time dado
(hour? t) → devuelve #t si el time t es una hora exacta (como 6:00); devuelve #f en cualquier otro caso
```

Ejemplo:

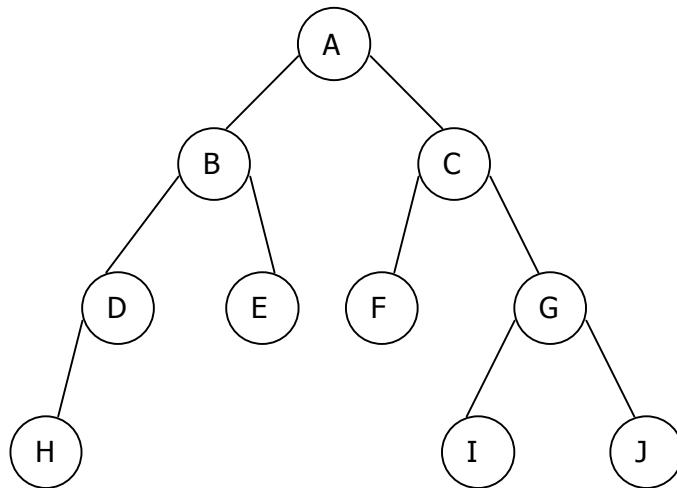
```
> (define t (make-time 4 12))
```

```
> (minute t)
12
> (hour t)
4
> (hour? t)
#f
```

Implementa el constructor `make-time` y los operadores `hour`, `minute` y `hour?` usando la técnica del paso de mensajes, en la que el objeto devuelto por el constructor es una función que admite mensajes.

### Pregunta 3 (10 puntos)

El procedimiento `(prof_hojas tree)` devuelve una lista plana con la profundidad de cada hoja de un árbol binario. Ejemplo:



`(prof_hojas tree)` devolverá (3 2 2 3 3)

Rellena los huecos para que `prof_hojas` funcione correctamente:

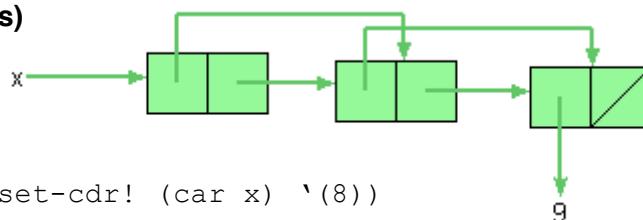
```
(define (prof_hojas tree)
  (define (help tree depth)
    (cond ((null? tree) _____)
          ((leaf? tree) _____)
          (else ( _____
                    (help tree 0))))))
```

### Pregunta 4 (10 puntos)

Para cada apartado:

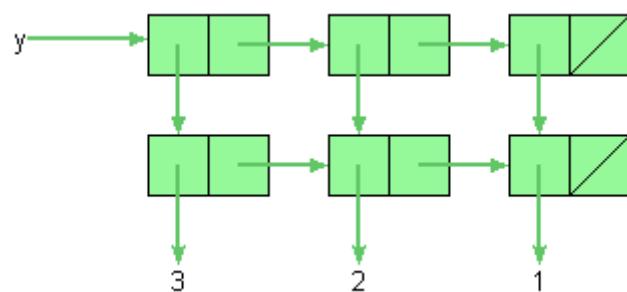
- Escribe una expresión Scheme que construya el diagrama caja y puntero de la figura.
- Dibuja el diagrama caja y puntero resultante de la mutación indicada.

#### A) (5 puntos)



Mutación: `(set-cdr! (car x) '(8))`

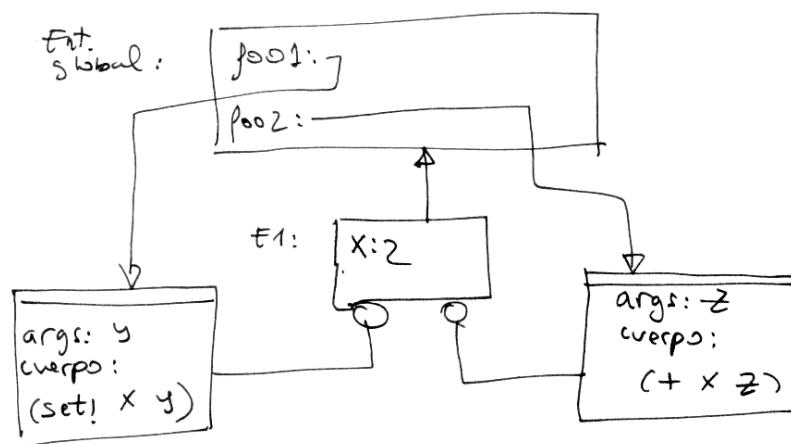
B) (5 puntos)



Mutación: (set-car! (cdr (cadr y)) 4)

Pregunta 5 (10 puntos)

Supongamos el entorno definido por la siguiente figura



A) (5 puntos) Completa el siguiente programa para que genere el entorno anterior.

```
(define c  (_____))
(define foo1 (_____) )
(define foo2 (_____) )
```

Nota: La variable auxiliar  $c$  no aparece en la figura.

B) (4 puntos) Explica paso a paso cómo se han evaluado las expresiones del programa que acabas de completar para generar el entorno de la figura, según el modelo de evaluación de Scheme basado en entornos.

C) (1 punto) Escribe un ejemplo de invocación de `foo1` y `foo2`, indicando el resultado devuelto.

Nombre: \_\_\_\_\_

**Lenguajes y Paradigmas de Programación  
Convocatoria extraordinaria C4, curso 2015-16  
Examen final**

**Normas importantes**

- Se debe contestar cada pregunta en las hojas que entregamos. No olvides poner el nombre. Puedes usar lápiz.
- El profesor que está en el aula no contestará ninguna pregunta, salvo aquellas que se refieran a posibles errores en los enunciados de los ejercicios.
- La duración del examen es de 2,5 horas

**Ejercicio 1 (1 punto)**

Escribe la **función recursiva** (`invertir-lista lista-parejas`) que recibe una lista de parejas y devuelve la lista invertida, y en la que además aparecen intercambiados los elementos de todas sus parejas.

Ejemplo:

```
(invertir-lista '((1 . 2) (3 . 4) (5 . 6))) ⇒ {{6 . 5} {4 . 3} {2 . 1}}
```

```
(define (intercambia-pareja pareja)
  (cons (cdr pareja)
        (car pareja)))
```

```
(define (invertir-lista lista-parejas)
  (if (null? lista-parejas)
      '()
      (append (invertir-lista (cdr lista-parejas))
              (list (intercambia-pareja (car lista-parejas))))))
```

## Ejercicio 2 (1 punto)

Escribe la **función recursiva** (`cuenta-a-b lista`) que recibe una lista que contiene los símbolos a y b repetidos y devuelve una pareja cuyos elementos son el número de as y bs de la lista que aparecen en la lista

Ejemplo:

```
(cuenta-a-b '(a b a b a b)) => {3 . 3}
(cuenta-a-b '(a a a a)) => {4 . 0}
```

```
(define (suma-1-izq pareja)
  (cons (+ 1 (car pareja))
        (cdr pareja)))

(define (suma-1-der pareja)
  (cons (car pareja)
        (+ 1 (cdr pareja)))))

(define (cuenta-a-b lista-a-b)
  (if (null? lista-a-b)
      (cons 0 0)
      (if (equal? 'a (car lista-a-b))
          (suma-1-izq (cuenta-a-b (cdr lista-a-b)))
          (suma-1-der (cuenta-a-b (cdr lista-a-b)))))))
```

### Ejercicio 3 (1 punto)

Suponemos la función (`aplica-operadores lista-operadores pareja`) que recibe una lista de símbolos que definen operadores aritméticos (pueden ser cualquiera de los símbolos +, -, \*, /) y una pareja y que devuelve la lista resultante de aplicar cada operador a los elementos de la pareja (no hace falta que la implementes).

Ejemplo:

```
(aplica-operadores '(- - + *) (5 . 4)) ⇒ {1 1 9 20}
```

Usando la función anterior implementa la función (`aplica-lista-op lista-operadores lista-parejas`) que recibe una lista de operadores y una lista de parejas y devuelve una lista resultante de aplicar todos los operadores de la lista a todas y cada una de las parejas. Para implementar esta función y cualquier otra función auxiliar que necesites, **debes usar funciones de orden superior**.

Ejemplo:

```
(aplica-lista-op '(+ - / *) '((5 . 4) (2 . 3))) ⇒ {9 1 5/4 20 5 -1 2/3 6}
```

```
(define (aplana-lista lista)
  (fold-right append '() lista))
```

```
(define (aplica-lista-op lista-op lista-parejas )
  (aplana-lista (map (lambda (pareja) (aplica-operadores lista-op pareja))
    lista-parejas)))
```

#### Ejercicio 4 (2 puntos)

Escribe el predicado (`valida-tree? tree`) que recibe un árbol de símbolos '`abierto`' y '`cerrado`', y recorre el árbol devolviendo `#t` en caso de que para todos sus nodos (excepto los nodos hoja) se cumpla que el dato del nodo sea el que más ocurrencias tiene en las raíces de los hijos, y `#f` en caso contrario (ver figura). Puedes usar funciones de orden superior y/o recursión mutua.

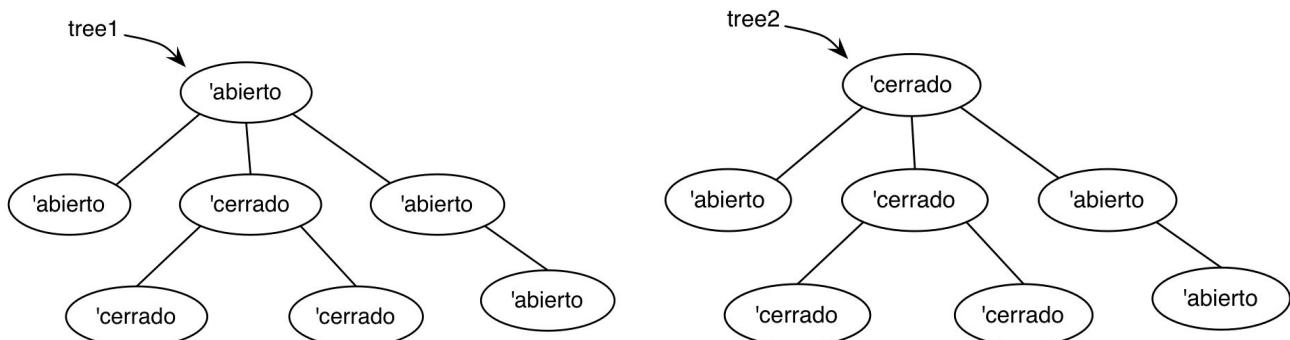
Debes usar las funciones de la barrera de abstracción de árboles (no hace falta que las implementes). Recordamos esta barrera de abstracción:

- (`dato-tree tree`): devuelve el dato de la raíz del árbol
- (`hijos-tree tree`): devuelve la lista de árboles hijos del árbol
- (`hoja-tree? tree`): `#t` o `#f` si el árbol es una hoja (no tiene hijos)

También puedes utilizar la función (`ocurrencias lista`) que recibe una lista de símbolos '`abierto`' y '`cerrado`', y devuelve una pareja cuya parte izquierda es el número de ocurrencias del símbolo '`abierto`', y su parte derecha el número de ocurrencias del símbolo '`cerrado`'. (No tienes que implementar esta función).

Ejemplos:

```
(ocurrencias '(abierto cerrado cerrado cerrado abierto)) => {2 . 3}
(ocurrencias '(abierto abierto)) => {2 . 0}
```



```
(valida-tree? tree1)  => #t
(valida-tree? tree2)  => #f
```

; Hay varias formas de hacerlo.. esta usa la FOS map para obtener la pareja de las ocurrencias de 'abierto y 'cerrado en las raíces de los hijos:

```
(define (mayor-car? pareja)
  (> (car pareja) (cdr pareja)))
```

```
(define (mayor-cdr? pareja)
  (> (cdr pareja) (car pareja)))
```

```
(define (valida-tree? tree)
  (if (hoja-tree? tree)
    #
    (and (if (equal? (dato-tree tree) 'abierto)
             (mayor-car? (ocurrencias (map dato-tree (hijos-tree tree))))))
         (mayor-cdr? (ocurrencias (map dato-tree (hijos-tree tree))))))
        (valida-bosque? (hijos-tree tree)))))
```

```
(define (valida-bosque? bosque)
  (if (null? bosque)
    #
    (and (valida-tree? (car bosque))
         (valida-bosque? (cdr bosque)))))
```

### Ejercicio 5 (1 punto)

Escribe la **función recursiva** (`veces-lista dato lista`) que recibe un dato (un símbolo) y una lista estructurada de símbolos y devuelve el número de veces que aparece el dato en la lista. Debes usar la barrera de abstracción de listas estructuradas (no hace falta que la implementes). Recordamos esta barrera de abstracción:

(`car lista`): devuelve el primer elemento de la lista  
(`cdr lista`): devuelve el resto de la lista  
(`hoja? lista`): devuelve #t o #f si el parámetro es un dato o una lista

Ejemplo:

(`veces-lista 'a '(a j f (k a (b w) a) d c))` ⇒ 3

```
(define (veces-lista dato lista)
  (cond ((null? lista)
         0)
        ((hoja? lista)
         (if (equal? dato lista)
             1
             0))
        (else
         (+ (veces-lista dato (car lista))
            (veces-lista dato (cdr lista))))))
```

## Test (4 puntos)

Cada respuesta errónea **penaliza con 0,12 puntos**. En todas las preguntas (excepto en la primera), **sólo una respuesta es la correcta. Redondea la letra** con tu respuesta.

**1. (0,4 puntos)** Conecta cada lenguaje con el tipo de ejecución de los programas escritos en él (1 fallo: 0,2 puntos, 2 fallos o más: 0 puntos y no descuenta)

- 1. C -> A
- 2. Scheme -> B
  - A. Se ejecuta el código compilado
- 3. Swift -> A
- 4. Python -> B
  - B. Se ejecuta por un intérprete
- 5. C++ -> A
- 6. Ruby -> A

**2. (0,4 puntos)** La definición de “función de orden superior” es:

- A. Una función que transforma listas, aplicando otras funciones a sus elementos, como map, filter o fold-right.
- B. Una función que contiene en su cuerpo una llamada a la forma especial lambda .
- C. Una función que recibe como parámetro otra función.
- D. Una función que generaliza otra función, añadiendo un parámetro adicional a esta última.

**3. (0,4 puntos)** ¿Qué aparece por pantalla al ejecutar el siguiente código?

```
let array = [3, 1, 4, 2, 2]
var suma = 0
for i in 1...array[0] {
    suma += array[i]
}
print("La suma es \((suma)")
```

- A. La suma es 8
- B. La suma es 5
- C. La suma es 0
- D. La suma es 7

**4. (0,4 puntos)** El siguiente código produce un error:

```
01 let array = [2, 2, 5, 1]
02 var array2: [String] = []
03 let dic = [1: "Uno", 2: "Dos", 3: "Tres", 4: "Cuatro"]
04 for i in 0..
```

¿Cuál de los siguientes cambios permitiría ejecutar el código sin errores?

- A. Sustituir la línea 4 por:  
`for i in 0..`
- B. Sustituir la línea 5 por:  
`if let elem = dic[array[i]] {  
 array2.append(elem)  
}`
- C. Sustituir la línea 5 por:  
`array2.append(dic[array[i]]!)`
- D. Sustituir la línea 2 por:  
`var array2: [String]? = []`

**5. (0,4 puntos)** Supongamos el siguiente código:

```
func foo(opcional: Int?) -> Int {  
    if let i = opcional {  
        return i  
    } else {  
        return 30  
    }  
}  
let a: Int? = 10  
print(foo(10)+foo(Int("abc"))+foo(a))
```

¿Qué valor se imprime?

- A. Da error
- B. 30
- C. 70
- D. 50

**6. (0,4 puntos)** Supongamos el siguiente código:

```
func cuadrado(x: Int) -> Int {return x*x}  
let numeros = [Int](0...5)
```

Y supongamos las siguientes expresiones con funciones de orden superior:

1. numeros.map(cuadrado).reduce(0, combine: +)
2. numeros.map {\$0 \* \$0}.reduce(0, combine: {\$0 + \$1})
3. numeros.map {\$0 \* \$0}.reduce(0, combine: +)
4. numeros.reduce(0, combine: +).map(cuadrado)

¿Cuáles de las expresiones anteriores son correctas y devuelven la suma de los números del array elevados al cuadrado, es decir el valor 55?

- A. 1 y 2
- B. 2 y 3
- C. Sólo 4
- D. 1, 2 y 3

**7. (0,4 puntos)** Supongamos la siguiente función:

```
func foo<A,B>(x: (A) -> B, y: A) -> (A,B) {  
    ...  
}
```

¿Cuál sería una expresión correcta de la sentencia return?

- A. **return (y, x(y))**
- B. **return (x, x(y))**
- C. **return (y, x(x))**
- D. Ninguna de las anteriores

**8. (0,4 puntos)** Supongamos el siguiente código:

```
struct MiStruct {  
    var x = 0  
}  
class MiClass {  
    var x = 0  
}  
var s = MiStruct()  
var c = MiClass()  
var s2 = s  
var c2 = c  
s2.x = 10  
c2.x = 10  
print("c:\\"(c.x), s:\\"(s.x))
```

¿Qué aparece por pantalla?

- A. c:0, s:10
- B. c:10, s:10
- C. c:0, s: 0
- A. c:10, s: 0**

**9. (0,4 puntos)** Supongamos la siguiente función:

```
func prueba(i: Int) -> Int? {
    if i < 100 {
        return i
    } else {
        return nil
    }
}
```

¿Cuál de las siguientes definiciones de una clase A es correcta?

A.

```
class A {
    var a: Int = prueba(0)
    var b = 0
    var c: Int
    init(valor: Int) {
        c = valor
    }
}
```

B.

```
class A {
    var a = 0
    var b = 0
    let c = 10
    init(valor: Int) {
        b = valor
    }
}
```

C.

```
class A {
    var a: Int? = prueba(200)
    var b = 0
    var c: Int
}
```

D.

```
class A {
    var a: Int? = prueba(0)
    var b = 0
    var c: Int
    init(v1: Int, v2: Int) {
        a = v1
        b = v2
    }
}
```

**10. (0,4 puntos)** Supongamos el siguiente código:

```
protocol P {  
    var x : Int { get set }  
    var y : Int { get }  
}  
  
class ClaseA {  
    var x : Int = 0  
}
```

¿Cuál de las siguientes definiciones es **errónea**?

|                                                                                                                                         |                                                                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| A.<br><pre>extension ClaseA: P {<br/>    var x : Int {<br/>        get {<br/>            return x*x<br/>        }<br/>    }<br/>}</pre> | B.<br><pre>class ClaseB: P {<br/>    var x : Int = 0<br/>    var y : Int = 0<br/>    var z : Int = 0<br/>}</pre> |
| C.<br><pre>class ClaseC: P {<br/>    var x : Int = 0<br/>    var y : Int = 0<br/>}</pre>                                                | D.<br><pre>class ClaseD: ClaseA {<br/>    var z : Int = 0<br/>}</pre>                                            |

Nombre: \_\_\_\_\_ Grupo: \_\_\_\_\_

## Lenguajes y Paradigmas de Programación

Curso 2012-2013

Convocatoria extraordinaria de Julio

### Normas importantes

- La puntuación total del examen es de 10 puntos.
- Se debe contestar cada pregunta en las hojas que entregamos. Utiliza las últimas hojas para hacer pruebas. No olvides poner el nombre.
- La duración del examen es de 2,5 horas.

### Ejercicio 1 (1,5 puntos)

**1) (0,2 puntos)** Supongamos que queremos ampliar un lenguaje orientado a objetos (como C++) para hacerlo más funcional. ¿Por dónde deberíamos empezar a trabajar? (señala la respuesta correcta):

- a) Añadiendo funciones y macros que permitan construir DSLs
- b) Definiendo primitivas de *pattern matching*
- c) Haciendo que las funciones sean objetos de primera clase del lenguaje
- d) Definiendo funciones que admitan un número variable de argumentos

**2) (0,2 puntos)** Cuál es el cambio fundamental que introdujo la arquitectura von Neumann en el diseño de los computadores (señala la respuesta correcta):

- a) Introdujo memorias de válvulas de vacío
- b) El concepto de Unidad Aritmético-Lógica capaz de realizar operaciones matemáticas sobre los registros
- c) El concepto de registros de procesador separados de los registros de memoria
- d) El concepto de programa almacenado en la memoria del computador

**3) (0,2 puntos)** Enlaza cada máquina con su fotografía:



Máquina de Pascal



Máquina de sumar



Motor de diferencias de Babagge



UNIVAC

**4) (0,4 puntos)** Explica cuáles fueron los 2 primeros lenguajes de programación de alto nivel, que dieron origen al paradigma imperativo y al paradigma funcional. Explica algunos detalles de su creación y de sus características.

**5) (0,5 puntos)** Explica los usos que tienen los traits de Scala: ampliación de comportamiento y modificaciones apilables. Pon un ejemplo de cada uso.

## Ejercicio 2 (1,5 puntos)

**a) (0,5 puntos)** Define en Scheme la función recursiva (`lista-ref pos lista`) que reciba un número y una lista, y devuelva el elemento situado en la posición indicada por dicho número. Puedes suponer que la posición siempre será correcta, es decir, será un número comprendido entre 1 y la longitud de la lista.

Ejemplos:

```
(define lista '(a b c d e f g h))
(lista-ref 1 lista) → a
(lista-ref 4 lista) → d
```

**b) (1 punto)** Define en Scheme la función recursiva (`descodifica lista-mensaje lista-pos`) que reciba dos argumentos: `lista-mensaje` es una lista cuyos elementos son listas de símbolos, por ejemplo `'((a p i) (b c l) (p m e))` y `lista-pos`, una lista cuyos elementos son parejas de números, por ejemplo `(list (cons 2 3) (cons 1 2) (cons 3 1))`. Cada pareja representa el número de sublista y una posición de esa sublista. La función tiene que devolver la lista de símbolos obtenidos según la secuencia de parejas. Suponemos que todas las posiciones son correctas. Utiliza la función definida en el apartado anterior.

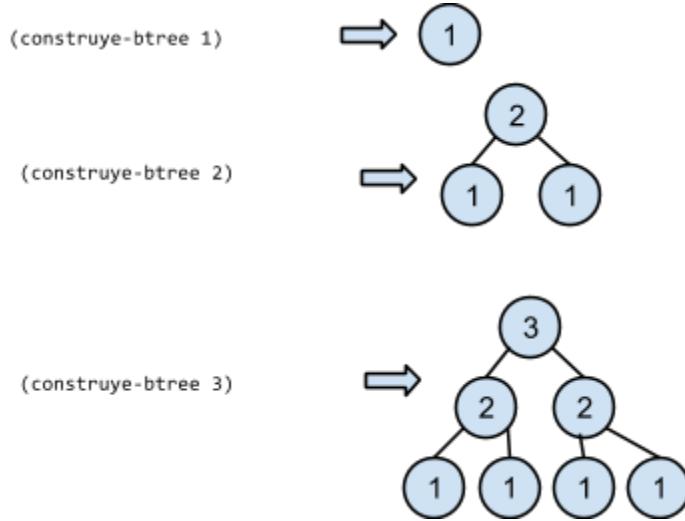
Ejemplo:

```
(define lista-mensaje '((a p i) (b c l) (p m e)))
(define lista-pos (list (cons 2 3) (cons 1 2) (cons 3 1)))
(descodifica lista-mensaje lista-pos) → (l p p)
```

### Ejercicio 3 (2 puntos)

a) (0,75 puntos) Implementa en Scheme la función (`construye-btree n`) que reciba un número `n` como argumento y devuelva un árbol binario de altura `n` donde el dato de cada nodo contiene la altura en la que se encuentra.

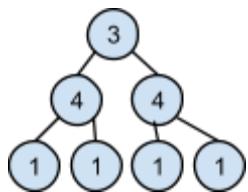
Ejemplo:



**b) (0,75 puntos)** Define la función (aplica-nivel-btree! btree nivel func) que reciba un árbol binario, un nivel y una función unaria como argumento y mute el nivel indicado del btree, con el resultado de aplicar func a cada nodo de ese nivel. Ejemplo:

```
(define btree (construye-btree 3))
(aplica-nivel-btree! btree 2 cuadrado)
```

btree →



c) (0,5 puntos) Para cada una de las siguientes estructuras, escribe su expresión-S asociada y dibuja su correspondiente box & pointer:

| Árbol genérico       | Árbol binario        | Expresión-S          |
|----------------------|----------------------|----------------------|
|                      |                      |                      |
| Expresión-S asociada | Expresión-S asociada | Expresión-S asociada |
| Box & Pointer        | Box & Pointer        | Box & Pointer        |

#### Ejercicio 4 (1,5 puntos)

a) (0,5 puntos) Escribe en Scala la función `intercambia(lista)` que reciba una lista de tuplas de dos elementos de tipo entero y devuelva una lista con las mismas tuplas pero con los elementos de cada tupla intercambiados.

Ejemplo:

```
val lista = List((1,2), (3,4), (5,6))
intercambia(lista) → List((2,1), (4,3), (6,5))
```

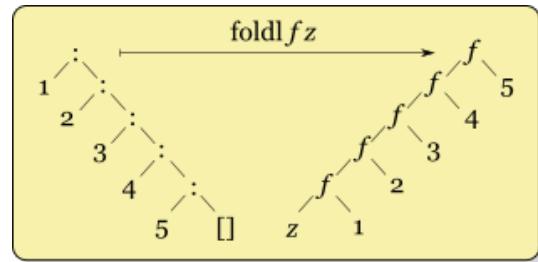
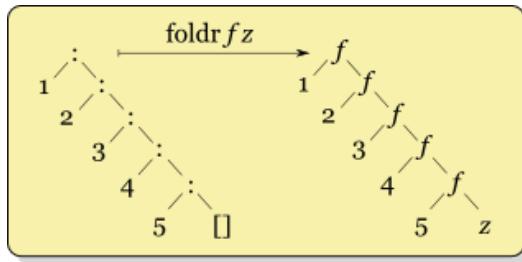
b) (1 punto) Escribe en Scala la función `asocia(func, lista)` que reciba una función y una lista de números enteros, y devuelva una lista de tuplas. La función `asocia` debe recorrer la lista de números y devolver aquellos números contiguos (en forma de tupla) que cumplan que el número de la derecha es el resultado de aplicar la función al número de su izquierda. La función `asocia` tiene que ser recursiva y debes definir por completo su prototipo.

Ejemplo:

```
def cuadrado(x:Int) = x*x
val lista = List(2, 4, 16, 5, 10, 100, 105)
asocia(cuadrado _, lista) → List((2,4),(4,16),(10,100))
```

### Ejercicio 5 (1,75 puntos)

Las funciones fold, también conocidas como "funciones plegado", reciben 3 argumentos: una función binaria, un caso base y una lista. Según sea el tipo de plegado (izquierda o derecha), llamadas fold-right y fold-left respectivamente, se aplica la función binaria de forma acumulativa a los elementos de la lista como muestran las siguientes figuras:



Por ejemplo, el funcionamiento del fold-right sería:

```
(fold-right max 0 '(3 10 14 6)) =  
(max 3 (max 10 (max 14 (max 6 0)))) y devolvería 14
```

```
(fold-right cons '() '(1 2 3 4)) =  
(cons 1 (cons 2 (cons 3 (cons 4 '())))) y devolvería (1 2 3 4)
```

En el caso del fold-left:

```
(fold-left max 0 '(3 10 14 6)) =  
(max (max (max (max 0 3) 10) 14) 6) y devolvería 14  
(fold-left cons '() '(1 2 3 4)) =  
(cons (cons (cons (cons '() 1) 2) 3) 4) y devolvería  
((((().1).2).3.4))
```

**a) (0,5 puntos)** Implementa la función fold-right. Explica si tu función genera un proceso recursivo o iterativo.

**b) (0,75 puntos)** Implementa la función fold-left. Explica si tu función genera un proceso recursivo o iterativo.

**c) (0,5 puntos)** Implementa la función (fold sentido) que reciba un símbolo '< o >' que indique el sentido de plegado y devuelva una función que realice el "plegado" correspondiente. Si lo necesitas puedes usar las funciones anteriores. Ejemplos:

```
((fold '<) max 0 '(3 7 2 4)) → 7
(define fold-derecha (fold '>))
(fold-derecha + 0 '(1 2 3)) → 6
```

### Ejercicio 6 (1,75 puntos)

a) (0,75 puntos) Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión? ¿Se crea alguna clausura?

```
var y = 1
def g(x: Int) = {
    var y = x + 2
    x + y
}
g(y + g(y))
```

**b) (1 punto)** Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión? ¿Se crea alguna clausura?

```
val x = 0
val y = 2
val z = 4

def g(x:Int) = {
    val y = 6
    (z:Int) => {x+y+z}
}

def foo(a:(Int, Int=>Int)) = {
    val x = 8
    val y = 10
    val z = 12
    a._2(a._1)
}
val f = g(16)
val a = (14,f_)
foo(a)
```

# Soluciones

## Lenguajes y Paradigmas de Programación

### Curso 2004-2005

### Examen de la Convocatoria de Septiembre

#### Pregunta 1 (10 puntos)

A) (5 puntos) Define una función (`cumplen-todos pred list`) que devuelva #t o #f dependiendo de si todos los elementos de la lista `list` cumplen el predicado `pred`. La lista `list` puede ser un pseudo-arbol y contener otras listas. Por ejemplo:

```
(cumplen-todos par? '(2 4 (6 8) 10)) -> #t
(cumplen-todos impar? '(1 3 5 6)) -> #f
(cumplen-todos impar? '()) -> #t
```

B) (2 puntos) La función que has definido en el apartado anterior ¿es iterativa o recursiva? ¿Por qué?

C) (3 puntos) Supongamos que uno de los elementos de la lista no cumple el predicado ¿se sigue procesando la lista en tu solución? En el caso de que no, perfecto, ya puedes sumar los 3 puntos de este apartado. En el caso de que se siga procesando la lista hasta el final, modifica la función para que no lo haga y devuelva #f justo en ese instante.

#### Solución:

```
;; version recursiva
(define (cumplen-todos? pred lista)
  (cond
    ((null? lista) #t)
    ((list? (car lista)) (and (cumplen-todos? pred (car lista))
                               (cumplen-todos? pred (cdr lista))))
     (else (and (pred (car lista))
                (cumplen-todos? pred (cdr lista)))))))

;; version iterativa
(define (cumplen-todos? pred lista)
  (cond
    ((null? lista) #t)
    ((list? (car lista))
     (if (cumplen-todos? pred (car lista))
         (cumplen-todos? pred (cdr lista))
         #f))
    (else (if (pred (car lista))
              (cumplen-todos? pred (cdr lista))
              #f))))
```

### Pregunta 2 (10 puntos)

Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3:00 PM es la hora 15:00. El constructor para las horas del día es `make-time`. Toma dos argumentos: las horas y los minutos. 4:12 debería ser (`make-time 4 12`) También queremos los siguientes tres operadores:

(`hour t`) → devuelve la parte de hora de un time dado  
(`minute t`) → devuelve la parte de minutos de un time dado  
(`hour? t`) → devuelve #t si el time t es una hora exacta (como 6:00); devuelve #f en cualquier otro caso

Ejemplo:

```
> (define t (make-time 4 12))  
> (minute t)  
12  
> (hour t)  
4  
> (hour? t)  
#f
```

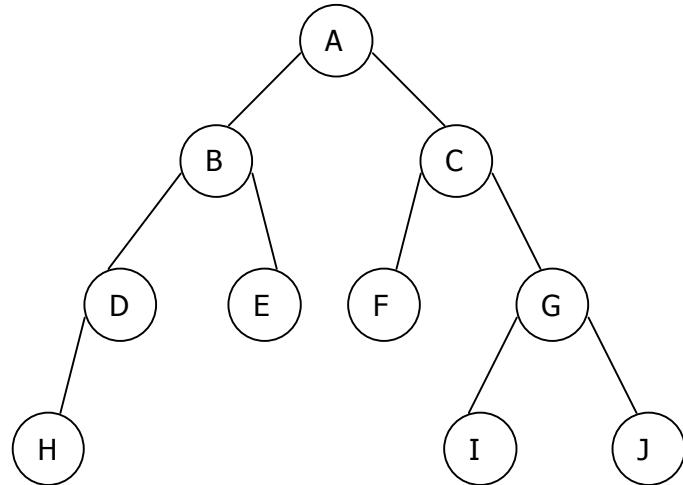
Implementa el constructor `make-time` y los operadores `hour`, `minute` y `hour?` usando la técnica del paso de mensajes, en la que el objeto devuelto por el constructor es una función que admite mensajes.

### Solución:

```
(define (make-time hora minuto)  
  (lambda (mens)  
    (cond  
      ((equal? mens 'minuto) minuto)  
      ((equal? mens 'hora) hora)  
      ((equal? mens 'hora?) (= minuto 0))  
      (else "mensaje incorrecto"))))  
  
(define (minute t)  
  (t 'minuto))  
  
(define (hour t)  
  (t 'hora))  
  
(define (hour? t)  
  (t 'hora?))
```

### **Pregunta 3 (10 puntos)**

El procedimiento (prof\_hojas tree) devuelve una lista plana con la profundidad de cada hoja de un árbol binario. Ejemplo:



(prof hojas tree) devolverá (3 2 2 3 3)

Rellena los huecos para que prof hojas funcione correctamente:

```
(define (prof_hojas tree)
  (define (help tree depth)
    (cond ((null? tree) _____)
          ((leaf? tree) _____)
          (else ( _____
                  _____
                  (help tree 0))))))
```

### **Solución:**

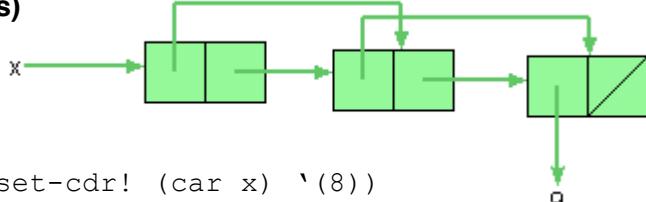
```
(define (prof_hojas tree)
  (define (help tree depth)
    (cond ((null? tree) '())
          ((hoja? tree) (list depth))
          (else (append (help (hijo-izq tree) (1+ depth))
                        (help (hijo-der tree) (1+ depth)))))))
  (help tree 0))
```

### Pregunta 4 (10 puntos)

Para cada apartado:

- Escribe una expresión Scheme que construya el diagrama caja y puntero de la figura.
- Dibuja el diagrama caja y puntero resultante de la mutación indicada.

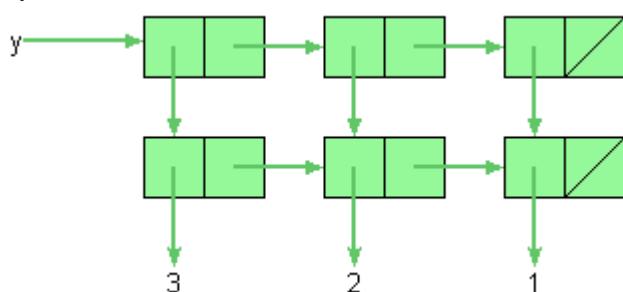
#### A) (5 puntos)



Mutación: (set-cdr! (car x) ' (8))

9

#### B) (5 puntos)

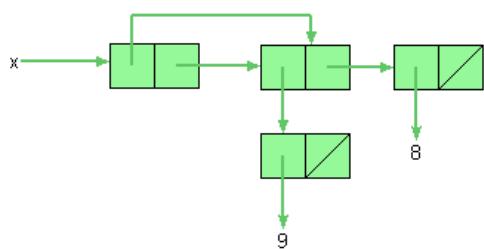


Mutación: (set-car! (cdr (cadr y)) 4)

#### Solución:

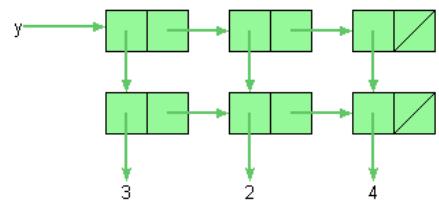
```
A) (define x
  (let ((a ' (9)))
    (let ((b (cons a a)))
      (cons b b))))
```

Después de la mutación:



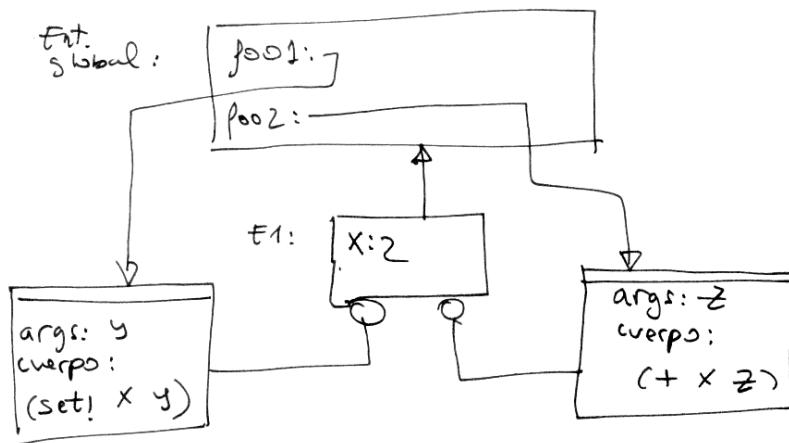
```
B) (define y
  (let ((z (list 3 2 1)))
    (list z (cdr z) (cddr z))))
```

Después de la mutación:



### Pregunta 5 (10 puntos)

Supongamos el entorno definido por la siguiente figura



A) (5 puntos) Completa el siguiente programa para que genere el entorno anterior.

```
(define c (_____))
(define foo1 (_____))
(define foo2 (_____))
```

Nota: La variable auxiliar **c** no aparece en la figura.

B) (4 puntos) Explica paso a paso cómo se han evaluado las expresiones del programa que acabas de completar para generar el entorno de la figura, según el modelo de evaluación de Scheme basado en entornos.

C) (1 punto) Escribe un ejemplo de invocación de **foo1** y **foo2**, indicando el resultado devuelto.

**Solución:**

```
(define c
  (let ((x 2))
    (cons (lambda (y) (set! x y))
          (lambda (y) (+ x y)))))
(define foo1 (car c))
(define foo2 (cdr c))
```

La primera expresión se evalúa en el entorno global. La forma especial “let” crea un entorno en el que la variable “x” se liga al valor “2”. Se trata del entorno E1. Este entorno es hijo del entorno global. En ese entorno se evalúa la expresión “(cons (lambda ...) (lambda ..))”. Cada una de las llamadas a “lambda” crean una función asociada al entorno E1. La llamada a “cons” crea una pareja que tiene como parte izquierda la primera función (el resultado de la evaluación de “(lambda (y) (set! x y))” y como parte derecha la segunda función (el resultado de evaluar “(lambda (y) (+ x y))”). Por último, la pareja resultante de la llamada a “cons” se liga con la variable “c” en el entorno global.

La segunda expresión crea una variable “foo1” en el entorno global que se liga con la parte izquierda de la pareja “c”. Esto es, “foo1” queda ligada a la función generada por la primera llamada a “lambda”.

Por último, la tercera expresión liga a “foo2” en el entorno global con la función creada por la segunda llamada a “lambda”.

```
(foo1 3) -> ok  
(foo2 5) -> 8
```

# Examen de Lenguajes y Paradigmas de Programación

## Convocatoria de Junio de 2002

Nombre y apellidos:

### Normas generales

- ?? El examen consta de 2 partes; la primera parte se deberá hacer con el ordenador apagado y en la segunda se podrá usar el ordenador para probar los procedimientos que haya que programar. En ambas partes se pueden usar tanto los apuntes como el libro de la asignatura.
- ?? La nota total de la asignatura será la media entre la nota de prácticas y la nota del examen, siempre que ésta última sea mayor o igual a 3.
- ?? **La duración del examen es de 2 horas**

### Parte 1 (sin ordenador)

**1. (1 punto)** ¿Qué responderá Scheme a las siguientes expresiones?

```
a.- (word 10 (+ 2 3))  
b.- (let ((a 5) (b 3)) ((lambda (x) (* a x)) b))  
c.- ((lambda (p x) (p 2 x)) * 3)  
d.- (append (list 1 2) (cdr (cons 1 nil)))  
e.- (car (list (list 1 2) 3))
```

**2. (1 punto)** Supongamos el siguiente procedimiento

```
(define (multi x)  
  (if (null? x)  
      (lambda (arg) arg)  
      (lambda (arg)  
        ((car x) ((multi (cdr x)) arg))))))
```

Explica qué hace multi y escribe un ejemplo de una llamada a la función junto con el resultado que devolvería.

**3. (1 punto)** Supongamos el siguiente código

```
(define (pairs x lista)  
  (if (null? lista) lista  
      (append (list (cons x (car lista))) (pairs x (cdr lista)))))
```

Explica qué hace pairs y escribe un ejemplo de una llamada a la función junto con el resultado que devolvería.

**4. (1 punto)** Dibuja diagramas box-and-pointer que representan los resultados de las siguientes expresiones

```
(define x1 (list (cons 'a 'b) (list 'c 'd)))  
(define x2 (cons 'a (cons 'b (cons 'c 'd))))  
(define x3 (list 'a 'b 'c (cons 'd '()))))  
(set-cdr! x3 (cdr x1)))
```

## Parte 2 (con ordenador)

**5. (1,5 puntos)** Escribe una función test que tome como argumentos un número y una lista de números y devuelva #t si y sólo si existe alguna pareja de números de la lista que sume lo mismo que el primer argumento.

Por ejemplo:

```
(test 5 (1 4 5 8 3))  
#t  
(test 5 (2 4 5 8 3))  
#f
```

**6. (1,5 puntos)** Imagina que tienes una versión de Scheme en la que solo están disponibles los números enteros y que, sin embargo, quieras poder representar cantidades de dinero como 3,95 €. Decides crear un tipo abstracto de datos con dos componentes, llamados euros y cents. Escribe los constructores y selectores:

```
(define (make-price e c) (+ (* e 100) c))  
(define (euros p) (quotient p 100))  
(define (cents p) (remainder p 100))
```

- Escribe el procedimiento +price que tome dos precios como argumento y devuelva su suma como un nuevo precio. *Respetá la abstracción de datos.*
- Ahora queremos cambiar la representación interna de forma que en lugar de representar el precio como un número de céntimos lo representemos como una pareja de dos números. Escribe los constructores y los selectores necesarios, respetando la abstracción de datos.

**7. (1'5 puntos)** Recuerda la forma de definir un constructor de contadores en el que se crea una variable “de clase” que mantiene la suma de todos los contadores y una variable “de instancia” que guarda el valor del contador:

```
(define make-count  
  (let ((glob 0))  
    (lambda ()  
      (let ((loc 0))  
        (lambda ()  
          (set! loc (+ loc 1))  
          (set! glob (+ glob 1))  
          (list loc glob))))))
```

Escribe una versión modificada de make-count en la que la variable del clase glob cuente el número de contadores que se han creado.

**8. (1,5 puntos)** Implementa, usando variables locales y paso de mensajes, las funciones de manejo de árboles que hemos visto en teoría:

```
(make-arbol dato hijos)  
(dato nodo)  
(hijos nodo)  
(hoja? nodo)
```

Usando esas funciones, escribe una función (contar-hojas arbol) que cuente el número de hojas de un árbol.

**Lenguajes y Paradigmas de Programación**  
**Examen de la convocatoria de Diciembre**  
**Curso 2003-2004**

Notas:

1. La duración del examen es de 3 horas
2. Los puntos máximos que se pueden obtener en el examen son 42
3. La puntuación de las prácticas se acumulará a la obtenida en el examen
4. Para aprobar la asignatura es necesario obtener en este examen más de 18 puntos y, junto con las prácticas, más de 30 puntos

**Pregunta 1 (6 puntos).** Dadas estas definiciones:

```
(define (square x) (* x x))
(define (inc) (set! count (+ count 1)) count)
(define count 5)
```

¿Cuál es el valor de la expresión `(square (inc))` bajo un orden de evaluación aplicativo? ¿Cuál es el valor de la misma expresión bajo un orden de evaluación normal?

**Pregunta 2 (6 puntos).** Escribe una función prefijo-a-infijo que tome una expresión aritmética de Scheme como argumento, en forma de árbol infijo, y devuelva una lista con la expresión en la forma de notación usual infija, con los operadores entre los operandos, como esto:

```
> (prefix-to-infix '(+ (* 2 3) (- 7 4)))
((2 * 3) + (7 - 4))
> (prefix-to-infix '(* (remainder 9 2) 5)
((9 remainder 2) * 5))
```

**Pregunta 3 (10 puntos).** Esta pregunta se refiere al tipo abstracto de datos árbol, definido por estos selectores y constructor:

```
(define datum car)
(define children cdr)
(define make-node cons)
```

Escribe el procedimiento de alto nivel `tree-accumulate`. Sus dos argumentos son una función `fn` y un árbol. La función `fn` tomará dos argumentos y será asociativa (esto es, no tienes que preocuparte del orden en el que encuentres los nodos en el árbol). El procedimiento `tree-accumulate` combinará todos los datos del árbol mediante la aplicación de `fn` a ellos de dos en dos, de forma análoga al procedimiento ordinario `accumulate`. Por ejemplo:

```
> (define my-tree
  (make-node 3 (list (make-node 4 '())
                     (make-node 7 '())
                     (make-node 2 (list (make-node 3 '())
   (make-node 8 '()))))))
> (tree-accumulate + my-tree)
27
> (tree-accumulate max my-tree)
8
```

**Pregunta 4 (10 puntos).** Escribe un procedimiento deep-subst! que tome tres argumentos, dos de los cuales son palabras y el tercero es cualquier estructura de lista (cualquier cosa hecha de parejas). El procedimiento debe mutar la estructura de la lista de forma que cualquier ocurrencia de la primer palabra se substituya por la segunda palabra. Ejemplos:

```
> (deep-subst! 'foo 'baz (list (cons 'hello 'goodbye) (cons 'moby 'foo)))
((hello . goodbye) (moby . baz))

> (deep-subst! 'a 'x (list (list 'a 'b 'c) (list 'b 'a 'd)
                           (list 'f 'a 'b)))
((x b c) (b x d) (f x b))
```

La función no debe crear nuevas parejas.

**Pregunta 5 (10 puntos).** Dibuja los diagramas de entornos resultantes de la evaluación de las siguientes expresiones:

```
(define (kons a b)
  (lambda (m)
    (if (eq? m 'kar) a b)))
(define p (kons (kons 1 2) 3))
```

# Lenguajes y Paradigmas de Programación

## Curso 2002-2003

### Examen de la Convocatoria de Junio

#### Normas importantes

- La puntuación total del examen son 42 puntos que sumados a los 18 puntos de las prácticas dan el total de 60 puntos sobre los que se valora la nota de la asignatura.
- Para sumar los puntos de las prácticas **es necesario obtener más de 18 puntos en este examen.**
- Se debe contestar cada pregunta **en un hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.

#### Pregunta 1 (3 puntos)

Escribe un procedimiento `add-numbers` que tome una frase como argumento. Algunas de las palabras en la frase pueden ser números. El procedimiento deberá devolver la suma de estos números, por ejemplo:

```
> (add-numbers '(8 dias 1 semana))  
9  
  
> (add-numbers '(76 trombones 4 flautas y 2 trompetas))  
82  
  
> (add-numbers '(all you need is love))  
0
```

#### Pregunta 2 (5 puntos)

Rellena los espacios en blanco para obtener el resultado esperado:

a) (2 puntos)

```
> ((lambda (x y z) (x (_____) z)) * + 3)  
> 15
```

b) (3 puntos)

```
> (((lambda (a b) b) + _____) 3 5)  
> 15
```

### Pregunta 3 (4 puntos)

Escribe un procedimiento (`(pref wd char)`) que tome como argumentos una palabra `wd` y un carácter `char` y que devuelva el prefijo de la palabra `wd` hasta llegar al carácter `char`.

Ejemplo:

```
> (pref "hola" "l")
> "ho"
> (pref "pepito" "i")
> "pep"
> (pref "hola" "i")
> "hola"
```

El procedimiento que has escrito es ¿recursivo o iterativo?. Explica la respuesta.

### Pregunta 4 (3 puntos)

Vamos a crear un TAD para representar una fecha. El constructor (`(make-fecha dia mes año)`) tendrá tres argumentos: día mes año.

Tendremos cuatro operadores:

`(año fec)` -> devuelve el año de la fecha pasada por parámetro.  
`(mes fec)` -> devuelve el mes de la fecha pasada por parámetro.  
`(dia fec)` -> devuelve el dia de la fecha pasada por parámetro.  
`(fin-mes fec)` -> devuelve true si es el último día del mes de la fecha `fec` (no tener en cuenta años bisiestos).

- Implementar `make-fecha`, `año`, `mes`, `dia` y `fin-mes` usando una representación interna de una lista de tres números. Es decir 02/12/2001 (02 12 2001).
- Implementar `make-fecha`, `año`, `mes`, `dia` y `fin-mes` usando una representación interna en la que la fecha se representa como un entero `((añox10000)+(mesx 100)+dia)`

### Pregunta 5 (5 puntos)

Escribe una función (`(max-levels)`) que recorra una lista de listas (un árbol sin datos en los nodos) y que devuelva el número de niveles de la rama que tiene mayor profundidad.

Por ejemplo

```
> (max-levels '(1 2 3 4))
> 1
> (max-levels '((1 (2) 3)))
> 2
> (max-levels '((((1) (2 (3))))))
> 3
```

## Pregunta 6 (4 puntos)

Vamos a hacer un conversor de monedas. Imaginariamente podemos tener una progresión lineal con las conversiones de moneda de este tipo:

... -> dólar -> rupia -> euro -> peseta -> ...  
... -> 1 -> 12 -> 62 -> 166 -> ...

Hemos rellenado una tabla con las conversiones:

```
(put 'convert 'dolar (attach-tag 'rupia 12)) ; 12 rupias son 1 dolar  
(put 'convert 'rupia (attach-tag 'euro 62)) ; 62 euros son 1 rupia  
(put 'convert 'euro (attach-tag 'peseta 166)) ; 166 pesetas son 1 euro  
....
```

Sólo hay una conversión para cada moneda. Si se quiere convertir de una moneda a otra que no sea correlativa, se deberá convertir una a una hasta llegar al tipo deseado. Por ejemplo, si queremos pasar de rupias a pesetas, debemos convertir rupias a euros y euros a pesetas.

Escribe el procedimiento `conversion` que tenga dos argumentos, `desde` y `hasta`, y deberá devolver un número que represente la conversión de la moneda `desde` a la moneda `hasta`.

```
>(conversion 'dolar 'rupia)  
>12  
>(conversion 'dolar 'euro)  
>744
```

## Pregunta 7 (7 puntos)

Queremos definir en POO una clase llamada **animal**, que tendrá como variable las vidas que tiene un animal y que por defecto valdrá 1. También definiremos las clases **gato** y **perro**, teniendo en cuenta que una gato tiene 7 vidas. Implementar un método que nos devuelva la cantidad de animales que se han creado, otro que nos devuelva la cantidad de perros y otro para la cantidad de gatos. Además implementaremos un método que se llamará **finVida** que restará una vida al animal al que se le aplique, de forma que cuando no le queden más vidas visualizará un cero. También implementaremos un método **estado** que devuelva el número de vidas de un animal. Por último podremos jugar a ser "DIOS" y redefinir el número de vidas de los animales que queramos, con el método **cambioVidas**. Este método cambia el contador de vidas del animal al número que queramos, siempre que ese número no sea mayor que la cantidad máxima de vidas que tiene el tipo de animal (1 para perro y 7 para gato).

### Pregunta 8 (6 puntos)

Explica la evaluación de las siguientes expresiones y dibuja el entorno resultante.

```
(define m
  (lambda (x)
    (lambda (y)
      (+ x y)))))

(define k (m 5))

(k 7)
```

### Pregunta 9 (5 puntos)

Queremos implementar el procedimiento `swap-cars!` que tome dos listas como parámetros e intercambie entre ellas el primer elemento de cada una. Por ejemplo:

```
>(define pares (list 2 4 6))
>(define impares (list 1 3 5))
>(swap-cars! pares impares)
>pares
>(1 4 6)
>impares
>(2 3 5)
```

(a) (3 puntos) ¿Cuál o cuáles de los siguientes procedimientos funcionan correctamente? Rodea SI o NO en cada uno. Si rodeas el NO, razona por qué en la casilla de la derecha.

|    |                                                                                                                           |  |
|----|---------------------------------------------------------------------------------------------------------------------------|--|
| SI | (define (swap-cars! x y)   (let ((temp x))     (set! x (cons (car y) (cdr x)))     (set! y (cons (car temp) (cdr y )))))) |  |
| NO | (define (swap-cars! x y)   (let ((temp (car x)))     (set! (car x) (car y))     (set! (car y) temp)))                     |  |
| SI | (define (swap-cars! x y)   (set-car! x (car y))   (set-car! y (car x)))                                                   |  |
| NO | (define (swap-cars! x y)   (let ((temp (cons (car x) (cdr x))))     (set-car! x (car y))     (set-car! y (car temp))))    |  |

(b) (2 puntos) Rellena el espacio en blanco del siguiente procedimiento para que `swap-cars!` funcione correctamente:

```
(define (swap-cars! X y)
  (set! x (cons (car y) x))
  (set-car! y (cadr x))
  ;; Rellena el espacio en blanco con una
  ;; llamada a set!, set-car! o set-cdr!
  (_____))
```

Nombre: \_\_\_\_\_ Grupo: \_\_\_\_\_

## Lenguajes y Paradigmas de Programación

Curso 2012-2013

Convocatoria extraordinaria de Julio

### Normas importantes

- La puntuación total del examen es de 10 puntos.
- Se debe contestar cada pregunta en las hojas que entregamos. Utiliza las últimas hojas para hacer pruebas. No olvides poner el nombre.
- La duración del examen es de 2,5 horas.

### Ejercicio 1 (1,5 puntos)

**1) (0,2 puntos)** Supongamos que queremos ampliar un lenguaje orientado a objetos (como C++) para hacerlo más funcional. ¿Por dónde deberíamos empezar a trabajar? (señala la respuesta correcta):

- a) Añadiendo funciones y macros que permitan construir DSLs
- b) Definiendo primitivas de *pattern matching*
- c) Haciendo que las funciones sean objetos de primera clase del lenguaje
- d) Definiendo funciones que admitan un número variable de argumentos

**2) (0,2 puntos)** Cuál es el cambio fundamental que introdujo la arquitectura von Neumann en el diseño de los computadores (señala la respuesta correcta):

- a) Introdujo memorias de válvulas de vacío
- b) El concepto de Unidad Aritmético-Lógica capaz de realizar operaciones matemáticas sobre los registros
- c) El concepto de registros de procesador separados de los registros de memoria
- d) El concepto de programa almacenado en la memoria del computador

**3) (0,2 puntos)** Enlaza cada máquina con su fotografía:



Máquina de Pascal



Máquina de sumar



Motor de diferencias de Babagge



UNIVAC

**4) (0,4 puntos)** Explica cuáles fueron los 2 primeros lenguajes de programación de alto nivel, que dieron origen al paradigma imperativo y al paradigma funcional. Explica algunos detalles de su creación y de sus características.

**5) (0,5 puntos)** Explica los usos que tienen los traits de Scala: ampliación de comportamiento y modificaciones apilables. Pon un ejemplo de cada uso.

## Ejercicio 2 (1,5 puntos)

**a) (0,5 puntos)** Define en Scheme la función recursiva (`lista-ref pos lista`) que reciba un número y una lista, y devuelva el elemento situado en la posición indicada por dicho número. Puedes suponer que la posición siempre será correcta, es decir, será un número comprendido entre 1 y la longitud de la lista.

Ejemplos:

```
(define lista '(a b c d e f g h))
(lista-ref 1 lista) → a
(lista-ref 4 lista) → d
```

**b) (1 punto)** Define en Scheme la función recursiva (`descodifica lista-mensaje lista-pos`) que reciba dos argumentos: `lista-mensaje` es una lista cuyos elementos son listas de símbolos, por ejemplo `'((a p i) (b c l) (p m e))` y `lista-pos`, una lista cuyos elementos son parejas de números, por ejemplo `(list (cons 2 3) (cons 1 2) (cons 3 1))`. Cada pareja representa el número de sublista y una posición de esa sublista. La función tiene que devolver la lista de símbolos obtenidos según la secuencia de parejas. Suponemos que todas las posiciones son correctas. Utiliza la función definida en el apartado anterior.

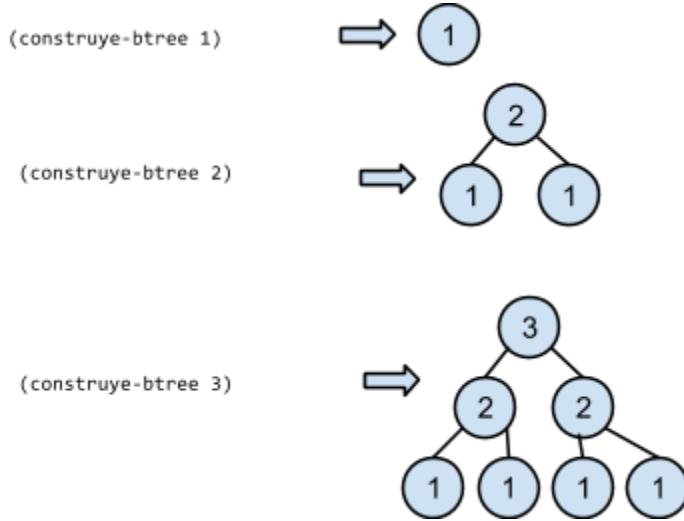
Ejemplo:

```
(define lista-mensaje '((a p i) (b c l) (p m e)))
(define lista-pos (list (cons 2 3) (cons 1 2) (cons 3 1)))
(descodifica lista-mensaje lista-pos) → (l p p)
```

### Ejercicio 3 (2 puntos)

a) (0,75 puntos) Implementa en Scheme la función (`construye-btree n`) que reciba un número `n` como argumento y devuelva un árbol binario de altura `n` donde el dato de cada nodo contiene la altura en la que se encuentra.

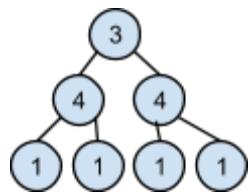
Ejemplo:



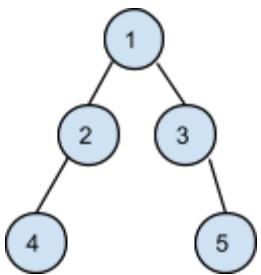
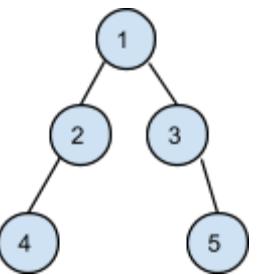
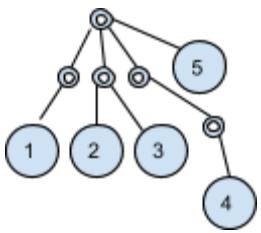
**b) (0,75 puntos)** Define la función (aplica-nivel-btree! btree nivel func) que reciba un árbol binario, un nivel y una función unaria como argumento y mute el nivel indicado del btree, con el resultado de aplicar func a cada nodo de ese nivel. Ejemplo:

```
(define btree (construye-btree 3))  
(aplica-nivel-btree! btree 2 cuadrado)
```

btree →



c) (0,5 puntos) Para cada una de las siguientes estructuras, escribe su expresión-S asociada y dibuja su correspondiente box & pointer:

| Árbol genérico                                                                    | Árbol binario                                                                     | Expresión-S                                                                         |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------|
|  |  |  |
| Expresión-S asociada                                                              | Expresión-S asociada                                                              | Expresión-S asociada                                                                |
| Box & Pointer                                                                     | Box & Pointer                                                                     | Box & Pointer                                                                       |

#### Ejercicio 4 (1,5 puntos)

a) (0,5 puntos) Escribe en Scala la función `intercambia(lista)` que reciba una lista de tuplas de dos elementos de tipo entero y devuelva una lista con las mismas tuplas pero con los elementos de cada tupla intercambiados.

Ejemplo:

```
val lista = List((1,2), (3,4), (5,6))
intercambia(lista) → List((2,1), (4,3), (6,5))
```

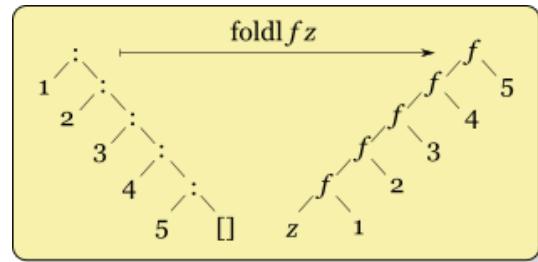
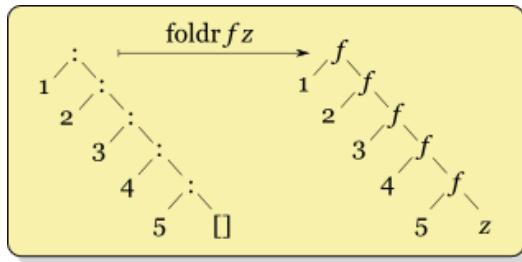
b) (1 punto) Escribe en Scala la función `asocia(func, lista)` que reciba una función y una lista de números enteros, y devuelva una lista de tuplas. La función `asocia` debe recorrer la lista de números y devolver aquellos números contiguos (en forma de tupla) que cumplan que el número de la derecha es el resultado de aplicar la función al número de su izquierda. La función `asocia` tiene que ser recursiva y debes definir por completo su prototipo.

Ejemplo:

```
def cuadrado(x:Int) = x*x
val lista = List(2, 4, 16, 5, 10, 100, 105)
asocia(cuadrado _, lista) → List((2,4),(4,16),(10,100))
```

### Ejercicio 5 (1,75 puntos)

Las funciones fold, también conocidas como "funciones plegado", reciben 3 argumentos: una función binaria, un caso base y una lista. Según sea el tipo de plegado (izquierda o derecha), llamadas fold-right y fold-left respectivamente, se aplica la función binaria de forma acumulativa a los elementos de la lista como muestran las siguientes figuras:



Por ejemplo, el funcionamiento del fold-right sería:

```
(fold-right max 0 '(3 10 14 6)) =  
(max 3 (max 10 (max 14 (max 6 0)))) y devolvería 14
```

```
(fold-right cons '() '(1 2 3 4)) =  
(cons 1 (cons 2 (cons 3 (cons 4 '())))) y devolvería (1 2 3 4)
```

En el caso del fold-left:

```
(fold-left max 0 '(3 10 14 6)) =  
(max (max (max (max 0 3) 10) 14) 6) y devolvería 14  
(fold-left cons '() '(1 2 3 4)) =  
(cons (cons (cons (cons '() 1) 2) 3) 4) y devolvería  
((((().1).2).3.4))
```

**a) (0,5 puntos)** Implementa la función fold-right. Explica si tu función genera un proceso recursivo o iterativo.

**b) (0,75 puntos)** Implementa la función fold-left. Explica si tu función genera un proceso recursivo o iterativo.

**c) (0,5 puntos)** Implementa la función (fold sentido) que reciba un símbolo '< o >' que indique el sentido de plegado y devuelva una función que realice el "plegado" correspondiente. Si lo necesitas puedes usar las funciones anteriores. Ejemplos:

```
((fold '<) max 0 '(3 7 2 4)) → 7
(define fold-derecha (fold '>))
(fold-derecha + 0 '(1 2 3)) → 6
```

### Ejercicio 6 (1,75 puntos)

a) (0,75 puntos) Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión? ¿Se crea alguna clausura?

```
var y = 1
def g(x: Int) = {
    var y = x + 2
    x + y
}
g(y + g(y))
```

**b) (1 punto)** Dibuja y explica los ámbitos que se crean al ejecutar las siguientes expresiones en Scala. ¿Cuántos ámbitos locales se crean? ¿En qué orden? ¿Qué valor devuelve la última expresión? ¿Se crea alguna clausura?

```
val x = 0
val y = 2
val z = 4

def g(x:Int) = {
    val y = 6
    (z:Int) => {x+y+z}
}

def foo(a:(Int, Int=>Int)) = {
    val x = 8
    val y = 10
    val z = 12
    a._2(a._1)
}
val f = g(16)
val a = (14,f_)
foo(a)
```

1. Dada la siguiente definición de árbol binario, rellenar el hueco:

```
indirect enum ArbolBinario{
    case nodo(Int, ArbolBinario, ArbolBinario)
    case vacio
}
func suma_pares(arbolb: ArbolBinario) -> Int{
    switch arbolb{
        case let .nodo(______):
            if(valor % 2 == 0){
                return valor + suma_pares(arbolb : hijolq) + suma_pares(arbolb : hijoDer)
            }
            else{
                return suma_pares(arbolb : hijolq) + suma_pares(arbolb: hijoDer)
            }
        case .vacio:
            return 0
    }
}
```

**Comentado [JMBP1]:** (valor, hijolq, hijoDer)

2. Rellena la siguiente instrucción para que la función devuelva la multiplicación de los dos parámetros:

```
func multiplica(n num: Double, por veces : Double) -> Double{
    return _____
```

**Comentado [JMBP2]:** num \* veces  
func(n:3, por:2)

3. Indica qué es incorrecto en las siguientes funciones:

```
indirect enum Arbol<T>{
    case nodo(T, [Arbol<T>])
}
func toArray<T>(arbol: Arbol) -> [T] {
    switch arbol{
        case let .nodo(dato, hijos):
            return [dato] + toArray(bosque : hijos)
    }
}
func toArray<T>(bosque : [Arbol]) -> [T]{
    if let primero = bosque.first{
        let resto = Array(bosque.dropFirst())
        return toArray(arbol:primero) + toArray(bosque :resto)
    }
    else{
        return []
    }
}
```

a. No pueden llamarse igual la segunda debería llamarse toArrayBosque.  
b. Hace falta un case default en el switch de la primera función.  
c. Falta el <T> en el tipo Arbol de parámetro en ambas funciones.

**Comentado [JMBP3]:** c.  
var a : Arbol<Int> = .nodo(3, [])
var b : Arbol<Int> = .nodo(3, [.nodo(7, [])])  
print(a)  
print(b)

- d. No hace falta el ligado opcional en la primera sentencia de la función bosque.

4. Dada la siguiente definición de función:

```
func f(a: [String], c: Character) -> Int{
    return a.filter( ) { $0.characters.contains(c)}.reduce(0) {$0 + $1.characters.count}
}
```

Indica que muestra por pantalla la siguiente función print:  
`print(f(a:["En", "un", "lugar", "de", "La", "Mancha"], c: "a"))`

**Comentado [JMBP4]: 13**

Se eliminan las cadenas que no contengan la a.  
 Se realiza un plegado por la izquierda, donde la primera suma sería 0 + la longitud de "lugar"

5. Indica que muestra por pantalla la función print:

```
let posiciones = [(1,2), (3,4), (0,2), (5,7), (8,4), (4,3)]
print(posiciones.filter{$_0.1 % 2 == 1}.reduce(0){(res:Int, p : (Int, Int)) -> Int in res + p.0})
```

**Comentado [JMBP5]: 9**

Primero se hace el filter que se queda con las parejas que tengan como segundo elemento valor impar, luego se pliega sumando los primeros elementos de las parejas resultantes [(5,7), (4,3)]

6. Rellena el hueco para que la función sume el producto de los elementos de las parejas, cuya suma sea par.

```
let posiciones = [(1,2), (3,4), (0,2), (5,7), (8,4), (4,3)]
print(posiciones.filter{_____}.reduce(0){(res:Int, p : (Int, Int)) -> Int in _____})
```

**Comentado [JMBP6]:  $(\$0.0 + \$0.1) \% 2 == 0$**

7. Indica qué muestra por pantalla la función print:

```
class Foo1{
    var valor : Int = 1
    func incremental(){
        valor += 5
    }
}
class Foo2 : Foo1{
    var miValor : Int = 3
    override func incremental(){
        super.incremental()
        miValor = super.valor
    }
}
var f2 = Foo2()
f2.incremental()
print(f2.miValor)
```

**Comentado [JMBP7]: se invoca a super.incremental()**

. valor = 6  
 miValor = 6  
 imprime 6

**8.** Indica que se imprime por pantalla al ejecutar el siguiente código:

```
class Clase{
    var almacen : [Int] = []
    func guarda(dato: Int){
        almacen = almacen + [dato]
    }
    var p : Int{
        get{
            return almacen.count
        }
        set{
            if newValue > 0 && newValue < p{
                var nuevo : [Int] = []
                for i in 0..<newValue{
                    nuevo.append(almacen[i])
                }
                almacen = nuevo
            }
        }
    }
}
var a = Clase()
a.guarda(dato: 2)
a.guarda(dato: 7)
a.guarda(dato: 3)
a.guarda(dato: 1)
a.p = 2
print(a.almacen)
```

**9.** Indica que muestra por pantalla la función print:

```
struct Foo{
    var valor1 : Int = 1
    var valor2 : Int = 2{
        didSet{
            if valor2 > 0{
                valor1 += 10
            }
        }
    }
}
var v2 = Foo()
v2.valor2 = -3
print(v2.valor2 + v2.valor1)
```

**Comentario [JMBP8]:** los guarda dejan el vector como:  
[2, 7, 3, 1]  
La propiedad calculada p en su set:  
Como el newValue = 2 está dentro de la longitud del vector,  
copia en un nuevo valor las posiciones desde 0 hasta la  
posición anterior al nuevo valor, quedando:  
[2, 7]  
Nota: cuando usamos p dentro del set, estamos llamando al  
get que devuelve el número de elementos del almacen.

**Comentario [JMBP9]:** Nota: los observadores NO se  
ejecutan en el constructor.  
En el observador didSet, valor2 ya vale -2, porque el valor  
anterior está en la variable implícita oldValue.

**10.** Indica que aparece en el siguiente programa.

```
struct Foo{
    var valor1 : Int = 1
    var valor2 : Int = 2{
        didSet{
            if valor2 > oldValue{
                valor1 += oldValue
            }
            else{
                valor2 += oldValue
            }
        }
    }
    init(x:Int){
        valor1 = valor1 + x
        valor2 = valor2 + x
    }
}
var v2 = Foo(x:2)
v2.valor2 = -3
print(v2.valor2 + v2.valor1)
```

**11.** ¿Qué aparece por pantalla al ejecutar el siguiente código?

```
struct A {
    static var a = 1
    var a = 2 {
        willSet {
            A.a += newValue
        }
        didSet {
            a = oldValue
        }
    }
}
var a = A()
a.a = 3
a.a = 4
print(A.a + a.a)
```

**Comentado [JMBP10]:** Nota: los observadores NO se ejecutan en el constructor.  
 Después de init  
 valor1 = 3  
 valor2 = 4  
 En el didSet  
 oldValue = 4, valor2 = -3  
 valor2 = -3 + 4 = 1  
 print(1 + 3) => 4

**Comentado [JMBP11]:** A.a = 1  
 El didSet deja el valor como estaba a = 2.  
 El willSet acumula el valor nuevo en la variable estatica que es todo el rato la misma:  
 A.a = 1 + 3 + 4 = 8  
 a.a = 2  
 print(10)

**12.** Indica qué aparece por pantalla al ejecutar el siguiente código

```
class A {
    var a = 10
    var b: String {
        return String(a)
    }
    func foo(x: Int) {
        a += x
    }
}
class B: A {
    override var b: String {
        return super.b + "%"
    }
    override func foo(x: Int) {
        a = x
    }
}
let b : A = B()
b.foo(x: 20)
print(b.b)
```

**Comentado [JMBP12]:** b.a = 10

```
b.foo(x = 20)
. .b.a = 20
print(b.b)
. ."20%"
```

Modificado el tipo de la variable b para que sea una referencia a tipo Base. Se invoca al método sobreescrito porque el enlace es dinámico.

**13.** Indica qué aparece por pantalla al ejecutar el siguiente código

```
let posiciones = [(1,2), (3,4), (0,2)]
print(posiciones.map{$0.1}.reduce(1,*))
```

**Comentado [JMBP13]:** El map aplica la función a cada elemento:

- . La función es return \$0.1
- . [2, 4, 2]

El reduce multiplica los elementos del vector:

$$1 * 2 * 4 * 2 \Rightarrow 16$$

**14.** Indica qué muestra por pantalla la función print:

```
let palabras = ["Hola", "me", "llamo", "Yolanda"]
print(palabras.map{$0, $0.count}.reduce(0){(res:Int, x:(String,Int)) in res+x.1})
```

**Comentado [JMBP14]:** Resultado del map:

[("Hola", 4), ("me", 2), ("llamo", 5), ("Yolanda", 7)]

Resultado del reduce:

$0 + 4 + 2 + 5 + 7 \Rightarrow 18$

**Comentado [JMBP15]:** {\$0.contains("e")}

**15.** Rellena los huecos para que el resultado sea el siguiente.

"mejesus"

```
let palabras = ["Hola", "me", "llamo", "Yolanda", "jesus", "muylarga"]
print(palabras.filter(______).reduce(""){(res:String, x:String) in _____ })
```

**Comentado [JMBP15]:** {\$0.contains("e")}

**16.** Considerando el siguiente código Swift:

```
let cadenas2 = ["En", "un", "lugar", "de", "La", "Mancha"]
print(cadenas2.reduce([]) {(res: [(String, Int)], c: String) -> [(String, Int)] in
    _____ }).sorted(by: {$0.1 < $1.1}))
```

rellena el hueco para que imprima por pantalla:

[("En", 2), ("un", 2), ("de", 2), ("La", 2), ("lugar", 5), ("Mancha", 6)]

Nombre: \_\_\_\_\_

**Examen convocatoria extraordinaria (Julio)**  
**Lenguajes y Paradigmas de Programación (Curso 2016-17)**

**Normas importantes del examen**

- El examen consta de una parte de ejercicios (**6 puntos**) y otra preguntas cortas (**2 puntos**) y otra de tipo test (**2 puntos**)
- Los profesores no contestarán ninguna cuestión durante la realización del examen, exceptuando aquellas que estén relacionadas con algún posible error en el enunciado de alguna pregunta.
- La duración del examen es de **2'5 horas**

## Ejercicio 1 (1,25 puntos)

a) (0,25 puntos) Escribe el **predicado** (`menor-punto? p1 p2 coord`) que recibe dos parejas que representan dos puntos con sus coordenadas 2D, y un símbolo que denota la coordenada x o y, y devuelve true si la coordenada especificada del primer punto es menor que su correspondiente coordenada del segundo punto.

```
(menor-punto? (cons 2 4) (cons 1 5) 'x) => #f  
(menor-punto? (cons 2 4) (cons 1 5) 'y) => #t
```

```
(define (menor-punto? p1 p2 coord)  
  (if (equal? coord 'x)  
      (< (car p1) (car p2))  
      (< (cdr p1) (cdr p2))))
```

b) (0,5 puntos) Escribe la **función recursiva** (`inserta-punto punto lista-puntos coord`) que recibe una pareja que representa un punto 2D, una lista de puntos 2D y un símbolo que denota la coordenada x o y, y devuelve una lista de puntos en la que el punto especificado se ha insertado en la lista original en la primera posición que cumpla que su coordenada especificada es menor que la del siguiente punto.

Debes utilizar la función `menor-punto?` definida en el apartado anterior.

```
(define lista-puntos  
  (list (cons 10 20) (cons 3 4) (cons 80 9) (cons 60 50)))  
 (inserta-punto (cons 40 30) lista-puntos 'x)  
 => {{10 . 20} {3 . 4} {40 . 30} {80 . 9} {60 . 50}}  
 (inserta-punto (cons 4 70) lista-puntos 'y)  
 => {{10 . 20} {3 . 4} {80 . 9} {60 . 50} {4 . 70}}
```

```
(define (inserta-punto punto lista-puntos coord)  
  (if (null? lista-puntos)  
      (list punto)  
      (if (menor-punto? punto (car lista-puntos) coord)  
          (cons punto lista-puntos)  
          (cons (car lista-puntos) (inserta-punto punto (cdr lista-puntos) coord))))))
```

c) (0,5 puntos) Escribe la **función recursiva** (`ordena-puntos lista-puntos coord`) que recibe una lista de puntos 2D y un símbolo que denota la coordenada x o y, y devuelve una lista de puntos en la que se han ordenado los puntos de la lista original con respecto a la coordenada especificada.

**Debes utilizar la función `inserta-punto` definida en el apartado anterior.**

```
(define lista-puntos
  (list (cons 10 20) (cons 3 4) (cons 80 9) (cons 60 50)))
(ordena-puntos lista-puntos 'x)
⇒ {{3 . 4} {10 . 20} {60 . 50} {80 . 9}}
(ordena-puntos lista-puntos 'y)
⇒ {{3 . 4} {80 . 9} {10 . 20} {60 . 50}}
```

```
(define (ordena-puntos lista-puntos coord)
  (if (null? lista-puntos)
      '()
      (inserta-punto (car lista-puntos)
                    (ordena-puntos (cdr lista-puntos) coord)
                    coord)))
```

## Ejercicio 2 (1,5 puntos)

a) (0,75 puntos) Escribe la función (add n pareja-listas pivot) que recibe una pareja con dos listas de números y añade el número n a la lista izquierda o a la derecha dependiendo de si es menor o mayor o igual que el número pivot.

```
;; Supongamos que la variable pareja-listas contiene la pareja
;; {{10 20 1} . {-10 -2 -30}}
(add 8 pareja-listas 0) => {{8 10 20 1} . {-10 -2 -30}}
```

```
(define (add n pareja-listas pivot)
  (if (< n pivot)
      (cons (cons n (car pareja-listas))
            (cdr pareja-listas))
      (cons (car pareja-listas)
            (cons n (cdr pareja-listas))))))
```

b) (0,75 puntos) Define la **función recursiva** (divide lista pivot) que recibe una lista de un números y un pivot (otro número) y devuelve una pareja con dos listas: la izquierda con todos los números menores que el pivot y la derecha con todos los números mayores o iguales.

Debes usar la función add definida en el apartado anterior.

```
(divide '(8 -10 10 20 -2 -30 1) 0) => {{-10 -2 -30} . {8 10 20 1}}
```

```
(define (divide lista pivot)
  (if (null? lista)
      (cons '() '())
      (add (car lista) (divide (cdr lista) pivot) pivot)))
```

### Ejercicio 3 (1,25 puntos)

Escribe la función (`mayor-rama tree`) que devuelve la lista de los datos de la rama más larga del árbol. Puedes usar recursión o funciones de orden superior. Debes usar las funciones de la barrera de abstracción de árboles: `dato-tree` e `hijos-tree`.

```
(define tree '(35 (10 (4) (6)) (25 (22 (12 (3)) (8)))))  
(mayor-rama tree) => {35 25 22 12 3})
```

; Versión con recursión mutua

; función auxiliar

```
(define (mayor-lista l1 l2)  
  (if (> (length l1) (length l2))  
      l1  
      l2))
```

```
(define (mayor-rama tree)  
  (if (hoja-tree? tree)  
      (list (dato-tree tree))  
      (cons (dato-tree tree)  
            (mayor-rama-bosque (hijos-tree tree)))))
```

```
(define (mayor-rama-bosque bosque)  
  (if (null? bosque)  
      ()  
      (mayor-lista (mayor-rama (car bosque))  
                  (mayor-rama-bosque (cdr bosque)))))
```

; versión FOS

```
(define (mayor-rama-fos tree)  
  (if (hoja-tree? tree)  
      (list (dato-tree tree))  
      (cons (dato-tree tree) (fold-right mayor-lista ()  
   (map mayor-rama-fos (hijos-tree tree))))))
```

## Ejercicio 4 (1,25 puntos)

Define la función (`comprueba-quiniela resultados quiniela`) que recibe una lista `resultados` con parejas que indican resultados de partidos de fútbol y la lista `quiniela` con valores 1, X o 2. Ambas listas tienen el mismo número de elementos. La función debe devolver cuántos aciertos tiene la quiniela. Debes utilizar **alguna función de orden superior y definir funciones auxiliares** (no lo hagas todo en una única función).

```
(define resultados
  (list (cons 1 1) (cons 2 1) (cons 2 0) (cons 0 0) (cons 1 2)))
(comprueba-quiniela resultados '(X 1 1 X 2)) => 5
(comprueba-quiniela resultados '(1 X X 2 1)) => 0
```

```
(define (convierte-a-quiniela resultado)
  (if (> (car resultado) (cdr resultado))
    1
    (if (< (car resultado) (cdr resultado))
      2
      'X)))

(define (comprueba-aciertos quiniela1 quiniela2)
  (if (null? quiniela1)
    0
    (if (equal? (car quiniela1) (car quiniela2))
      (+ 1 (comprueba-aciertos (cdr quiniela1) (cdr quiniela2)))
      (comprueba-aciertos (cdr quiniela1) (cdr quiniela2)))))

(define (comprueba-quiniela resultados quiniela)
  (comprueba-aciertos (map convierte-a-quiniela resultados) quiniela))
```

### Ejercicio 5 (0,75 puntos)

Vamos a representar un cartón de bingo como una lista de sublistas, donde cada sublista representa una línea del cartón. Rellena los huecos para completar la función mutadora `tacha-num!` que reciba un número y un cartón y, si se encuentra en el cartón, lo sustituya por un asterisco. Consideraremos que los cartones no tienen números repetidos, pero pueden tener distintos tamaños.

```
(define (tacha-num! num carton)
  (cond ((null? carton) #t)
        ((list? (car carton)) _____)
        ((equal? num (car carton)) _____) ;
        (else _____)))
```

```
(define carton '((3 40 2)
                  (25 12 33)
                  (20 10 22)))
(tacha-num! 40 carton)
carton => {(3 * 2)
            (25 12 33)
            (20 10 22)}
```

```
(define (tacha-num! num carton)
  (cond ((null? carton) #t)
        ((list? (car carton)) (tacha-num! num (car carton)))
        ((equal? num (car carton)) (set-car! carton '*))
        (else (tacha-num! num (cdr carton))))))
```

### Preguntas cortas (2 puntos)

- Cada pregunta tiene una puntuación de **0,4 puntos**.

1. Indica brevemente 4 características funcionales del lenguaje Swift (**0,1 punto por característica correcta**)

- Posibilidad de definir funciones
- Funciones como objetos de primera clase: expresiones lambda y clausuras
- Funciones de orden superior sobre colecciones: map, filter, reduce, etc.
- Inmutabilidad: let
- Semántica de tipos valor con struct

2. Ordena las siguientes parejas de lenguajes de programación, indicando en cada caso cuál es anterior y cuál posterior (**0,1 puntos por ordenación correcta, -0,1 por ordenación incorrecta**):

| Lenguajes         | Anterior | Posterior |
|-------------------|----------|-----------|
| Java, Python      | Python   | Java      |
| Smalltalk, Prolog | Prolog   | Smalltalk |
| C++, Java         | C++      | Java      |
| C#, Java          | Java     | C#        |

3. Rellena el hueco siguiente con un **bucle for** que guarde en la variable suma la suma de todos los valores devueltos por la función obtenerValores():

```
let valores: [Int] = obtenerValores()  
var suma = 0
```

```
for i in valores {  
    suma += i  
}
```

```
print("La suma de todos los valores es \(suma)")
```

4. Escribe qué aparece por pantalla al ejecutar el siguiente código?

```
let parejas = [(2,7), (3,5), (10,4), (5,5)]  
let resultado =  
    parejas.filter({$0.0 % 2 == 0}).map({($0.1, $0.1)})  
print(resultado)
```

```
[(7, 7), (4, 4)]
```

5. Escribe el código que debería ir en el reduce para que se devuelva una tupla cuya parte izquierda tenga el número de elementos del array y su parte derecha la suma de todos sus elementos:

```
let nums = [1,2,3,4,5,6,7]  
nums.reduce(  
) ⇒ (7, 28)
```

```
(0,0), {($0.0+1, $0.1+$1)}
```

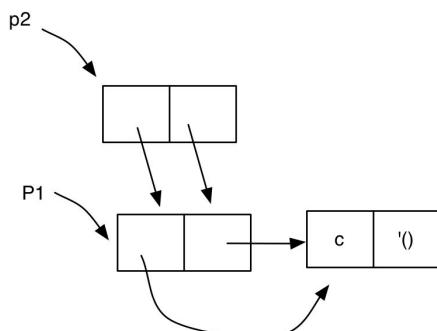
### Preguntas de tipo test (2 puntos)

- Cada pregunta tiene una puntuación de **0,4 puntos**.
- Todas las preguntas de tipo test tienen **una única respuesta correcta**.
- Redondea claramente la respuesta que consideres correcta.
- Las respuestas incorrectas tienen una **penalización de 0,1 puntos**.

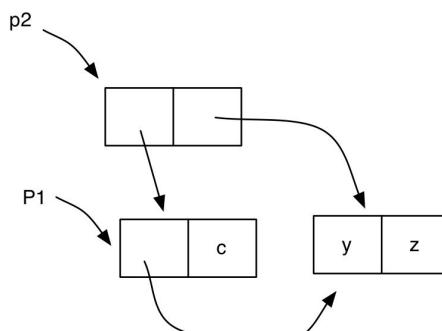
1. ¿Cuál es el diagrama box & pointer resultante de ejecutar las siguientes instrucciones? (Utiliza la última página del examen para hacerlo en sucio)

1. (define p1 (cons (list 'a) 'c))
2. (define p2 (list p1 (cons 'y 'z)))
3. (set-car! (car p1) (cadr p2))
4. (set-cdr! p2 (car p1))

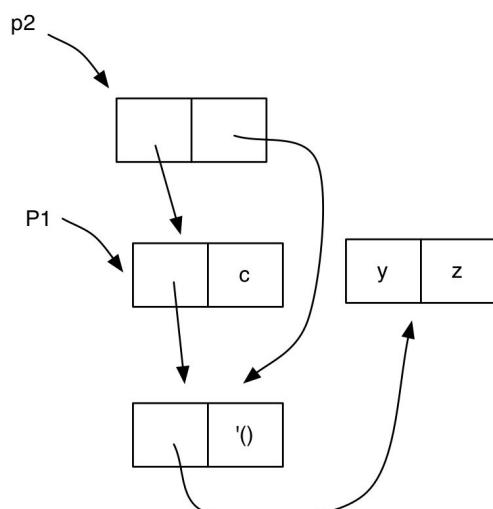
1.



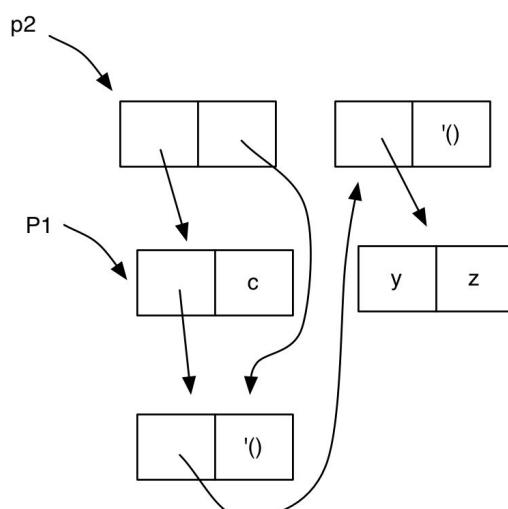
2.



3.



4.



2. En las siguientes instrucciones se asigna a foo un valor de una clave que no existe en el diccionario dic. ¿Qué sucede?

```
let dic = [1: "Uno", 2: "Dos", 3: "Tres", 4: "Cuatro"]
var foo = dic[5]
```

- A. La sentencia provoca un error de compilación
- B. La sentencia provoca un error de ejecución
- C. Se guarda en foo el valor "" (cadena vacía)
- D. **Se guarda en foo el valor nil**

3. Supongamos un protocolo P:

```
protocol B {
    var a: Int {get}
    var b: Int? {get set}
}
```

A continuación se muestran posibles definiciones de tipos que adoptan P. ¿Cuál de ellas **provocaría un error de compilación**?

A.

```
struct A: B {
    let a = 10
    var b: Int? = 10
}
```

B.

```
class A: B {
    var a: Int {
        return b! + 10
    }
    var b: Int? = nil
}
```

C.

```
class A: B {
    var a = 0
    var b: Int {
        return a + 10
    }
}
```

D.

```
struct A: B {
    let a = 10
    var b: Int?
}
```

4. Dado el siguiente código:

```
let x = 5
func foo(_ x: Int) -> ((()->Int, ()->Int) {
    func f() -> Int {
        return x + x
    }
    func g() -> Int {
        return f()
    }
    return (f,g)
}
```

Indica la llamada correcta:

|                                           |                                         |
|-------------------------------------------|-----------------------------------------|
| A.                                        | B.                                      |
| <pre>let y = foo(x+2) y[0]() y[1]()</pre> | <pre>let y = foo(x+2) y.0 y.1</pre>     |
| C.                                        | D.                                      |
| <pre>let y = foo(x+2) y.0() y.1()</pre>   | <pre>let y = foo(x+2) y.f() y.g()</pre> |

5. Supongamos el siguiente código que define un árbol binario

```
indirect enum ArbolBinario<T>{
    case nodo(T, ArbolBinario<T>, ArbolBinario<T>)
    case vacio
}
```

¿Cuál de las siguientes expresiones es la correcta para crear un árbol binario con 10 en la raíz, 8 como hijo izquierdo y 12 como hijo derecho?

A.

```
let a: ArbolBinario = .nodo(12,
    .nodo(5, nil, nil),
    .nodo(18, nil, nil))
```

B.

```
let a: ArbolBinario = .nodo(12,
    .nodo(5, .vacio, .vacio),
    .nodo(18, .vacio, .vacio))
```

C.

```
let a = ArbolBinario(12,
    ArbolBinario(5, .vacio, .vacio),
    ArbolBinario(18, .vacio, .vacio))
```

D.

```
let a: ArbolBinario = .nodo(12<Int>,
    .nodo(5<Int>, .vacio, .vacio),
    .nodo(18<Int>, .vacio, .vacio))
```

Nombre: \_\_\_\_\_

Titulación: \_\_\_\_\_

**Lenguajes y Paradigmas de Programación**

**Curso 2011-2012**

**Examen convocatoria extraordinaria C4**

**Grado en Ingeniería Informática / plan antiguo**

**Normas importantes**

- La puntuación total del examen es de 9 puntos (la nota de prácticas completa el punto adicional) para el Grado en Informática. Para el plan antiguo se escalará hasta 10 (multiplicando por 1,11)
- Se debe contestar cada pregunta en las hojas que entregamos. Utiliza las últimas hojas para hacer pruebas. No olvides poner el nombre.
- La duración del examen es de 3 horas.

**Ejercicio 1 (1 punto)**

Explica los siguientes conceptos:

- **(0,25 puntos)** Definición e implementación de las listas en Scheme. Ejemplos.
- **(0,25 puntos)** Explica las diferentes formas de definir actores en Scala. Ejemplos.
- **(0,25 puntos)** Diferencia entre valor y referencia. Ejemplos.
- **(0,25 puntos)** Tipos de variables en Scala. Ejemplos

## Ejercicio 2 (1,25 puntos)

a) **(0,5 puntos)** Define dos implementaciones en Scheme, una recursiva pura y otra iterativa, de la función (concat-mayores-de lista n) que tome una lista de cadenas y un número n y devuelva la concatenación de las cadenas con tamaño mayor o igual que n.

Ejemplo:

```
(concat-mayores-de '("hola", "pep", "automovil", "alicante"), 4) -> "holaaautomovilalicante"  
(concat-mayores-de '("hola", "adios", "barcelona", 10) -> ""
```

b) **(0,75 puntos)** Define una función de orden superior que generalice<sup>1</sup> la función anterior concat-mayores-de. Cuanto más general sea, mejor. Piensa bien el nombre que le darías, explica lo que haría y define su implementación en Scheme. Por último, escribe cómo implementarías concat-mayores-de usando la función general.

<sup>1</sup>Ejemplo de generalización: la función (filter lista predicado) permite generalizar todas las funciones concretas que filtran los elementos de una lista. Así, podemos definir una función concreta como (filtra-numeros-pares lista) usando la función general filter:

```
(define (filtra-numeros-pares lista)  
  (filter lista even?))
```

### Ejercicio 3 (1,5 puntos)

Supongamos que estamos diseñando una aplicación gráfica en la que codificamos puntos de coordenadas de pantalla (x,y) en forma de tuplas de dos enteros.

a) **(0,35 puntos)** Define una implementación recursiva en Scala de la función contarPuntos que toma como parámetros:

- una lista de puntos
- un predicado que se aplica a dos enteros

y que devuelve el número de puntos de la lista que cumplen el predicado.

Por ejemplo, supongamos que la función enCentroPantalla(x,y) determina si la coordenada 'x' está en el rango [250,350] y la coordenada 'y' en [150,250]. Un ejemplo de llamada a contarPuntos con esta función:

```
contarPuntos(List((330,170), (340,400), (10,100), (290,190)), enCentroPantalla_) => 2
```

b) **(0,4 puntos)** Define en Scala la función construyeComparadorRango que tome como parámetros:

- límite inferior y superior de 'x'
- límite inferior y superior de 'y'

y que devuelva un predicado que tome dos coordenadas 'x' e 'y' y compruebe si están dentro de los rangos.

Ejemplo:

```
val p = construyeComparadorRango(0,100,0,100)
contarCoords(List((20,30), (200,3), (3,90)), p) => 2
```

c) **(0,75 puntos)** Utilizando las funciones anteriores, define la función rangoConMasPuntos que tome:

- lista de puntos
- lista de 4-tuplas de 4 enteros que representan los rangos

Y que devuelva la 4-tupla que define el rango de la lista de rangos en la que se encuentran más número de puntos:

```
val listaPuntos = List((330,170), (340,400), (10,10), (290,190))
val listaRangos = List((0,100,0,100), (100,200,100,200))
rangoConMasPuntos(listaPuntos, listaRangos) => (0,100,0,100)
```

## Ejercicio 4 (1 punto)

Dadas las siguientes definiciones:

```
abstract class Clase1 {  
    def f(x:Int) : Int  
    def g(x:Int,y:Int) = x*y  
}  
  
class Clase2 extends Clase1 {  
    def f(x:Int) = x * 2  
}  
  
class Clase3 extends Clase2 {  
    override def g(x:Int,y:Int) = super.g(x,y) + 100  
    override def f(x:Int) = super.f(x+3)  
}  
  
trait Trait1 extends Clase2 {  
    abstract override def f(x:Int) = super.f(x+2)  
}  
  
trait Trait2 extends Clase1 {  
    abstract override def f(x:Int) = super.f(x*2)  
    abstract override def g(x:Int,y:Int) = super.g(x+2, y)  
}
```

a) (0,5 puntos) Indica y explica el resultado de las siguientes instrucciones. Si hay algún error, corrígelos y da una versión correcta de la expresión:

| Expresión                                                       | Resultado / Corrección |
|-----------------------------------------------------------------|------------------------|
| val a = new Clase1 with Trait2<br>a.g(2,3)                      |                        |
| val b = new Clase3 with Trait1<br>(b with Trait2).g(1,2)        |                        |
| val c = new Clase3 with Trait2<br>(c with Trait1).f(3)          |                        |
| val d = new Clase2 with Trait2 with Trait1<br>d.f(20)           |                        |
| val e : Clase1 = new Clase2 with Trait2 with Trait1<br>e.g(2,3) |                        |

b) (0,5 puntos) Para cada una de las siguientes expresiones, rellena los huecos para que se produzca el resultado esperado:

| Expresión                           | Resultado |
|-------------------------------------|-----------|
| val a : Clase 1 = _____<br>a.g(4,2) | 8         |
| val b = _____<br>b.g(3,2)           | 110       |

|                                   |    |
|-----------------------------------|----|
| val c : Clase1 = _____<br>c.f(10) | 30 |
| val d = _____<br>d.f(10)          | 54 |
| val e : Clase2 = _____<br>e.f(4)  | 26 |

## Ejercicio 5 (1,5 puntos)

a) **(0,75 puntos)** Define la función (`intercambia-si-suma! exp-s1 exp-s2 n`) que reciba dos expresiones-S que tienen la misma estructura pero datos diferentes y un número n. Deberá intercambiar los elementos entre las expresiones-S si la suma de los elementos en la misma posición es mayor o igual que n. Explica tu solución utilizando diagramas caja y puntero.

```
(define exps1 '(1 (2 3 (4) (3 (4 (5 (6 7)))))))
(define exps2 '(5 (1 2 (1) (2 (1 (0 (2 0)))))))
(intercambia-si-suma! exps1 exps2 6)
exps1 → (5 (2 3 (4) (3 (4 (5 (2 0))))))
exps2 → (1 (1 2 (1) (2 (1 (0 (6 7))))))
```

b) **(0,5 puntos)** Dada la siguiente expresión-S, dibuja su estructura jerárquica (arbórea) y su diagrama caja y puntero:

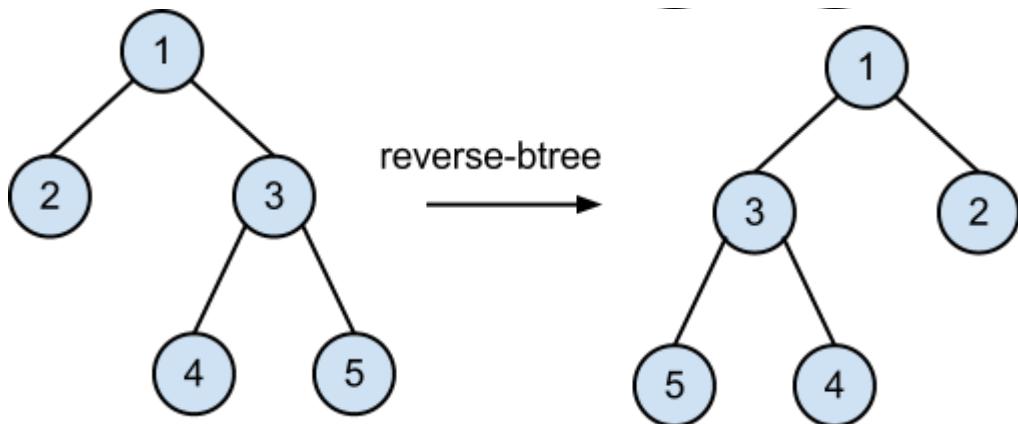
```
(define x '(1 (2) (3 (4 (5 6) 7) (8))))
```

c) **(0,25 puntos)** Modifica el diagrama caja y puntero para reflejar la siguiente mutación:

```
(set-cdr! (cdaddr x) (cdddr x))
```

### Ejercicio 6 (1,5 puntos)

a) **(0,75 puntos)** Define el procedimiento funcional (`reverse-btree btree`) que reciba un árbol binario y lo invierta.



b) **(0,75 puntos)** Implementa ahora su versión imperativa.

### Ejercicio 7 (1,25 puntos)

a) **(1 punto)** Dibuja y explica los ámbitos generados por la ejecución de las siguientes expresiones en Scala. ¿Qué resultado devuelve la última expresión?

```
var a=10
def f(x: Int) = {
    a = x*a
    a
}
def g(x:Int, h:(Int)=>Int) = {
    var cont = x
    ()=>{
        cont = h(cont)
        cont
    }
}
var c = g(10,f)
c()
c()
```

b) **(0,25 puntos)** ¿Se crea alguna clausura? Explica por qué.