Tecnologías y Paradigmas de la Programación - Ex1

1) (equal? '((b a) . (a b)) (append '(b) '(a . ()) '(a b)))

(equal? '((b a) a b) (append '(b) '(a . ()) '(a b)))
(equal? '((b a) a b) (append '(b) '(a) '(a b)))
(equal? '((b a) a b) '(b a a b))
#f

2) (equal? '((b a) . (a b)) (append '((b a) a b) '()))
(equal? '((b a) a b) (append '((b a) a b) '()))
(equal? '((b a) a b) '((b a) a b))
#t

3) (map (lambda (x y z) (if (> y x) z y)) '(40 30 20) '(28 25 21)
'((a) a (NO (a))))
((lambda ...) 40 28 '(a)) ((lambda ...) 30 25 'a) ((lambda ..) 20 21 '(NO (a)))
((if (> 28 40) '(a) 28) (if (> 25 30) 'a 25) (if (> 21 20) '(NO (a)) 21))
(28 25 '(NO (a)))

4) (map (lambda (x y z) (z x y)) (list list cons) (list list cons)
(list cons list))
(map (lambda (x y z) (z x y)) (list cons) (list cons) (cons list))
((lambda (..) ..) (~~list~~ cons ~~cons~~ list) ((lambda (..) ..) (~~cons~~ list ~~list~~ cons))
((list . list) (cons cons))

5) ((lambda (x) ((cdr x) (car x))) (cons 5 null?))
((lambda (x) ((cdr x) (car x))) '(5 . null?))
((car '(5 . null?)) (car '(5 . null?)))
(null? 5)
#f

6) (let* ((x 1) (y 2))
(let* ((y 8) (x (+ 3 y)) (z (+ x y))) (list x y z)))
; x = 1, y = 2
(let (y 8) (x (+ 3 2)) (z (+ 1 2))) (list x y z)))
; y = 8, x = 5, z = 3
(list 5 8 3)
(5 8 3)

7) (take-while positive? '(-1 -2 3 4))
→ coge mientras
()
↑ 1er elemento es negativo, luego finaliza la función

8) (let ((x list) (y 2)) (let* ((y 8) (z (+ y 5))) (x y z)))

;x = list , y = 2

(let ((x list) (y 2)) (let* ((y 8) (z (+ y 5))) (x y z)))

; x = list, y = 8 ,

(let ((x list) (y = 2)) (let* ((y 8) (z (+ 8 5))) (x y z)))

; x = list , y = 8, z = 13

(list 8 13)

(8 13)

9) ((lambda x (cons x '(a b))) 2 4)

(cons '(2 4) '(a b))

((2 4) a b)

10) (curry apply -) '(-1 3))

; lambda (x) (apply - x)

(lambda (x) (apply - x)) '(-1 3))

(apply - '(-1 3))

(- -1 3)

-4

11) (map (car (list cons 5)) '(1 2) '(3 4))

(map (car '(cons 5)) '(1 2) '(3 4))

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

((1 . 2) (3 . 4))

12) (map apply (list max min) '((3 4) (5 6)))

(map apply (max min) '((3 4) (5 6)))

(apply max '(3 4)) (apply min '(5 6))

(max '(3 4)) (min '(5 6))

(3 4)

13) ((lambda (x . y) (list x y)) 2 3 4)

(list 2 '(3 4))

(2 (3 4))

14) (drop-until positive? '(1 2 -3 -4))

(1 2 -3 -4)

(drop-until negative '(1 2 -3 -4)) => (-3 -4)

15) (equal? (((() . c) . b) . a) (reverse (append '(a) '(b) '(c))))

(equal? (((() . c) . b) . a) (reverse '(a b c)))

(equal? (((() . c) . b) . a) (c b a)

#f

16) (((curry filter) list?) '(a b c))

(((lambda (x) (filter x) list?) '(a b c)))

(filter list '(a b c))

()

17) (equal? '((b . a) (a b)) (append (list (cons b a)) '((a b) . ())))
(equal? '((b . a) (a b)) (append (list (b . a)) '((a b) ())))
(equal? '((b . a) (a b)) (append ((b . a)) '((a b) ())))
(equal? '((b . a) (a b)) '((b . a) (a b)))
#t

¿Cuándo una función se considera de orden superior?
Se considera una FOS si devuelve una función, devuelve una estructura que contiene una función o alguno de sus argumentos es una función.

Analiza y justifica la siguiente afirmación: "La FOS filter de Scheme puede considerarse un patrón de programación".
Correcto, puesto que si abstraemos la sintaxis tenemos: filter <función> lista, donde función puede ser cualquier función definida en Scheme o incluso funciones lambda.

¿Qué es la evaluación perezosa y qué ventaja tiene frente a la evaluación ansiosa? Pon un ejemplo.
En la evaluación ansiosa, siempre se evalúan primero los argumentos y luego la función tal que:

f (g(..) , h(..))
3° 1° 2°

Mientras que en la evaluación perezosa, los argumentos se evalúan solo si es necesario:

f (g(..) , h(..))
1° ? ?

f(x, y) ::== si x > y entonces x
sino y

f(2, 1/0) → en evaluación ansiosa, esto daría error
↳ en evaluación perezosa ⇒ 2

¿Es cierto que las S-expresiones o bien son listas o bien son átomos?

No es cierto, ya que () es un átomo, pero por ejemplo (S1 . L) el cdr es una lista, mientras que (S1 . S2), S2 no es una lista.

¿En qué consiste la técnica de currying? Pon un ejemplo.
Consiste en definir una función de n-argumentos como n-funciones de 1 argumento. Devuelve una expresión lambda con x argumentos ya introducidos que espera por el resto:

(define (op x y)
(curry + 5))