Turno de mañana

Ejercicio 2 (2 puntos)

a) (0,5 puntos) Para cada una de las siguientes expresiones, da una definición de `f` que sea correcta:

```
((f))
```

Solución:

```
(define (f)
      (lambda ()
            "hola"))
```

```
(f (f 4))
```

Solución:

```
(define (f x)
      (+ x 3))
```

b) (0,5 puntos) Rellena los huecos:

`(map (lambda (x) (> x 5)) '(1 2 3 4 5 6 7))` → _____

Solución:

```
(#f #f #f #f #f #t #t)
```

`(apply append (list (list 1 2 3) (list 2 3 4)))` → _____

Solución:

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

c) (0,5 puntos) Rellena los huecos para obtener el resultado esperado (puedes utilizar `string-length`):

```
(fold ________  _____ '("x" "abc" "xyzzz" "jk")) ; palabra más larga
→ "xyzzz"
```

Solución:

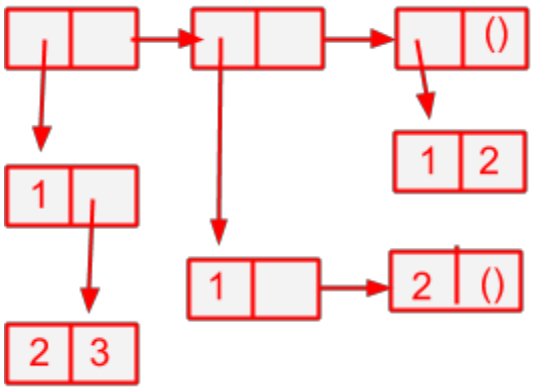
Hueco 1:
```
(lambda (str prev)
          (if (> (string-length str) (string-length prev))
                   str
                   prev))
```

Hueco 2: `""`

d) (0,5 puntos) Dibuja el *Box&Pointer* de la siguiente expresión y explica si genera una lista o no:

```
(cons (cons 1 (cons 2 3)) (list (list 1 2) (cons 1 2)))
```

Solución:



Es una lista

Ejercicio 3 (2 puntos)

a) (1 punto) Define la función recursiva `(siguientes lista x)` que reciba una lista y un elemento y devuelva una lista con los siguientes elementos de x en la lista. El elemento puede aparecer más de una vez.

Ejemplos:

```
(siguientes '(a b c d a c b c a) 'b) → (c c)
(siguientes '(a b c d a a c b a c a) 'a) → (b a c c)
```

Solución:

```
(define (siguientes lista x)
  (cond
    ((null? (cdr lista)) '())
    ((equal? (car lista) x)
     (cons (cadr lista)
           (siguientes (cdr lista) x)))
    (else (siguientes (cdr lista) x))))
```

b) (1 punto) Define la función recursiva `(min-max lista)` que recibe una lista de números y devuelve una pareja con el mínimo y el máximo de la lista:

Ejemplos:

```
(min-max '(1 2 3 4 5 6)) => (1 . 6)
(min-max '(1 1 1 1 1) => (1 . 1)
```

Solución:

(versión 1)

```
(define (min-max lista)
  (if (null? (cdr lista))
      (cons (car lista) (car lista))
      (let ((pareja (min-max (cdr lista))))
        (cond
          ((< (car lista) (car pareja))
           (cons (car lista) (cdr pareja)))
          ((> (car lista) (cdr pareja))
           (cons (car pareja) (car lista)))
          (else pareja)))))
```

(versión 2)

```
(define (min-max lista)
  (if (null? (cdr lista))
      (cons (car lista) (car lista))
      (let ((pareja (min-max (cdr lista))))
        (cons (min (car lista) (car pareja))
              (max (car lista) (cdr pareja))))))
```

Ejercicio 4 (2 puntos)

a) (0,75 puntos) Utilizando la función de orden superior que consideres más apropiada, define la función `(suman-par lista-parejas)` que reciba una lista de parejas de números y devuelva una lista con aquellas parejas cuya parte izquierda y derecha sumen un número par.

Ejemplo:

```
(suman-par '((1.2) (2.2) (3.6) (6.2) (4.5))) → ((2.2)(6.2))
```

Solución:

```
(define (suman-par lista)
  (filter (lambda (x)
            (even? (+ (car x) (cdr x)))) lista))
```

b) (1,25 punto) Define la función `(aplica-n lista-funcs lista-orden n)` que reciba dos listas, una de ellas contiene funciones unarias y la otra índices que referencian a posiciones de la primera lista (de 0 al número de elementos-1 de lista-funcs) y un número. Esta función deberá aplicar las funciones de `lista-funcs` al número n, en el orden indicado por lista-orden. Puedes utilizar `list-ref`, funciones auxiliares y/o de orden superior.

Ejemplo:

```
(aplica-n (list suma-1 mult-3 doble cuadrado) '(2 3 0 1 2 1) 3)
→ 6050  ; (doble (cuadrado (suma-1 (mult-3 (doble (mult-3 3))))))
```

Solución:

```
(define (aplica-n lista-funcs lista-orden n)
  (if (null? lista-orden)
      n
      ((list-ref lista-funcs (car lista-orden))
       (aplica-n lista-funcs (cdr lista-orden) n))))
```

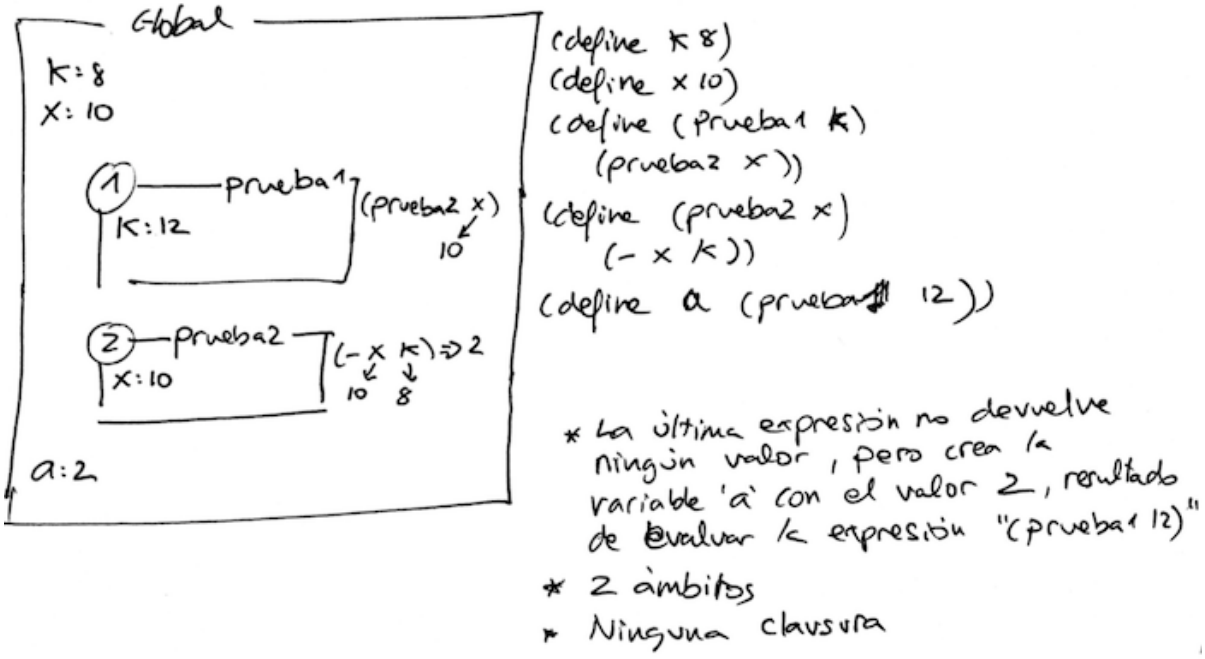
Ejercicio 5 (2 puntos)

Dados los siguientes fragmentos de código en Scheme, dibuja en un diagrama los ámbitos que se generan. Junto a cada ámbito escribe un número indicando en qué orden se ha creado. ¿Cuál es el resultado? ¿Cuántos ámbitos se crean? ¿Se crea alguna clausura?

a) (1 punto)

```
(define k 8)
(define x 10)
(define (prueba1 k)
    (prueba2 x))
(define (prueba2 x)
    (- x k))
(define a (prueba1 12))
```

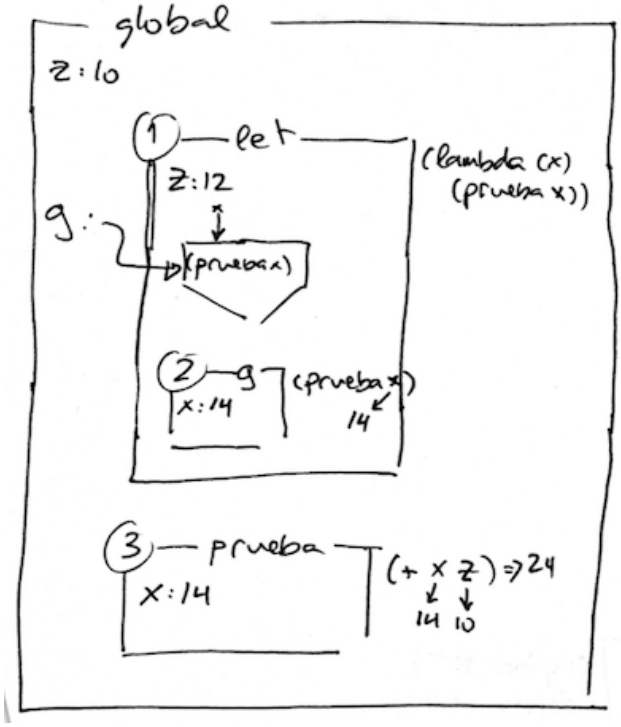
Solución:



a) (1 punto)

```
(define z 10)
(define (prueba x)
   (+ x z))
(define g (let ((z 12))
            (lambda (x)
               (prueba x))))
(g 14)
```

Solución:




```
(define z 10)
(define (prueba x)
  (+ x z))
(define g (let ((z 12))
            (lambda (x)
              (prueba x))))
```

(g 14) ⇒ 24

* Resultado: 24
* 3 ámbitos
* Sí, la función ligada a "g" es una clausura

Turno de tarde

Ejercicio 2 (2 puntos)

a) (0,5 puntos) Para cada una de las siguientes expresiones, da una definición de `f` que sea correcta:

```
((f 2))
```

Solución:

```
(define (f x)
      (lambda ()
            (+ x 3)))
```

```
(f (f 4))
```

Solución:

```
(define (f x)
      (+ x 3))
```

b) (0,5 puntos) Rellena los huecos:

```
(map car (list (cons 1 2) (cons 3 4) (cons 5 6))) → _______________
```

Solución:

```
(1 3 5)
```

```
(apply list (list (cons 1 2) (cons 4 5))) → _______________
```

Solución:

```
((1 . 2) (4 . 5))
```

c) (0,5 puntos) Rellena los huecos para obtener el resultado esperado (puedes utilizar `string-length`):

```
(fold ________  _____ '("x" "abc" "xyzzz" "jk"))  ; num caracteres de la palabra
                                                  ; más larga
→ 5
```

Solución:

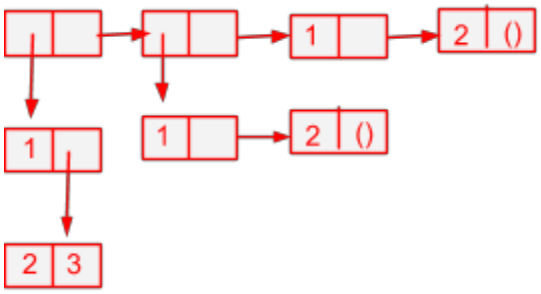
Hueco 1:
```
(lambda (str prev)
          (if (> (string-length str) prev)
              (string-length str)
              prev))
```

Hueco 2: `0`

d) (0,5 puntos) Dibuja el *Box&Pointer* de la siguiente expresión y explica si genera una lista o no:

```
(cons (cons 1 (cons 2 3)) (cons (list 1 2) (list 1 2)))
```

Solución:



Es una lista

Ejercicio 3 (2 puntos)

a) (1 punto) Define la función recursiva `(anteriores lista x)` que reciba una lista y un elemento y devuelva una lista con los elementos anteriores de `x` en la lista. El elemento puede aparecer más de una vez.

Ejemplos:

```
(anteriores '(a b c d a c b c a) 'b) → (a c)
(anteriores '(b w b b c d a c b c a t b) 'b) → (w b c t)
```

Solución:

```
(define (anteriores lista x)
  (cond
    ((null? (cdr lista)) '())
    ((equal? (cadr lista) x)
     (cons (car lista) (anteriores (cdr lista) x)))
    (else (anteriores (cdr lista) x))))
```

b) (1 punto) Define la función recursiva `(total x lista-parejas)` que recibe un símbolo x y una lista de parejas con símbolos en su parte izquierda y números en su parte derecha. Debe devolver una pareja que contiene el símbolo x y la suma de todas las partes derechas en las que aparece x.

Ejemplo:

```
(total 'a '((a . 2) (b . 3) (c. 8) (a . 1) (a . 10))) → (a . 13)
```

Solución:

(versión 1)

```
(define (total x lista)
  (if (null? lista)
      (cons x 0)
      (let ((pareja (total x (cdr lista))))
        (if (equal? x (caar lista))
            (cons x
                  (+ (cdar lista)
                     (cdr pareja)))
            pareja))))
```

(versión 2, con función auxiliar que suma dos parejas si sus cars son iguales)

```
(define (total x lista)
  (if (null? lista)
      (cons x 0)
      (suma-parejas (car lista) (total x (cdr lista)))))

(define (suma-parejas p1 p2)
  (if (equal? (car p1) (car p2))
      (cons (car p1) (+ (cdr p1) (cdr p2)))
      p2))
```

Ejercicio 4 (2 puntos)

a) (0,75 puntos) Utilizando la función de orden superior que consideres más apropiada, define la función `(mayores-izq lista-parejas)` que reciba una lista de parejas de números y devuelva una lista con aquellas parejas cuya parte izquierda sea mayor que la parte derecha.

Ejemplo:

```
(mayores-izq '((2.2)(6.2)(4.5)(5.5)(7.1))) → ((6.2)(7.1))
```

Solución:

```
(define (mayores-izq lista)
  (filter (lambda (x)
            (> (car x) (cdr x))) lista))
```

b) (1,25 puntos) Define la función `(aplica-if lista-funcs lista-bools n)` que reciba dos listas con el mismo número de elementos, una de ellas contiene funciones unarias y la otra valores booleanos que indican si las funciones situadas en esas posiciones de la primera lista se deben usar o no, y un número. Esta función deberá aplicar las funciones de `lista-funcs` seleccionadas al número `n`. Puedes utilizar funciones auxiliares y/o de orden superior.

Ejemplo:

```
(aplica-if (list suma-1 mult-3 doble cuadrado) '(#f #t #f #t) 3)
→ 27  ; (mult-3 (cuadrado 3))
```

Solución:

```
(define (aplica-if lista-funcs lista-bools n)
  (cond ((null? lista-funcs) n)
        ((car lista-bools)
           ((car lista-funcs)
                (aplica-if (cdr lista-funcs) (cdr lista-bools) n)))
        (else (aplica-if (cdr lista-funcs) (cdr lista-bools) n))))
```

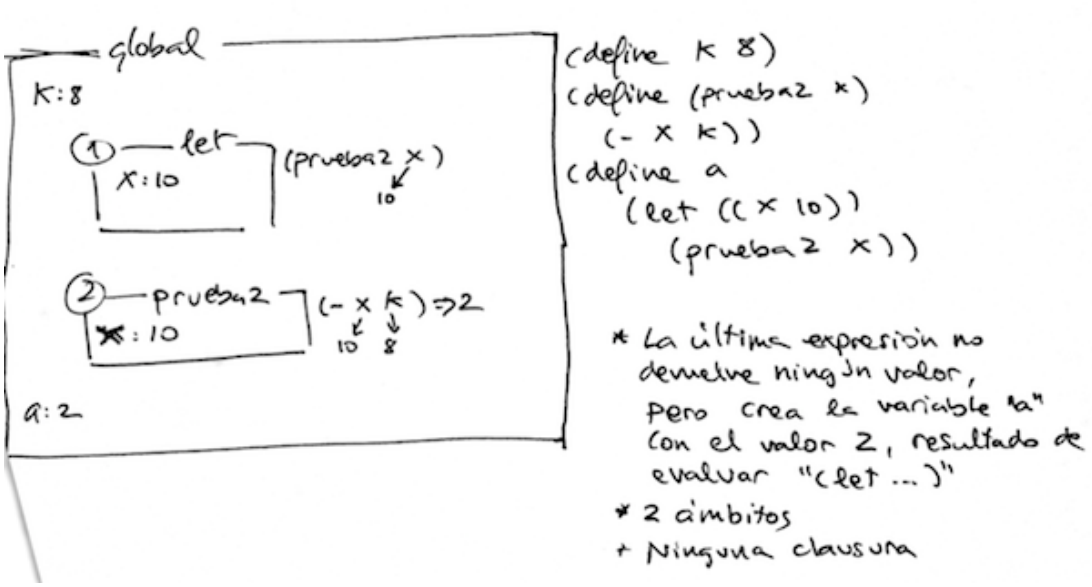
Ejercicio 5 (2 puntos)

Dados los siguientes fragmentos de código en Scheme, dibuja en un diagrama los ámbitos que se generan. Junto a cada ámbito escribe un número indicando en qué orden se ha creado. ¿Cuál es el resultado? ¿Cuántos ámbitos se crean? ¿Se crea alguna clausura?

a) (1 punto)

```
(define k 8)
(define (prueba2 x)
    (- x k))
(define a
    (let ((x 10))
        (prueba2 x))
```

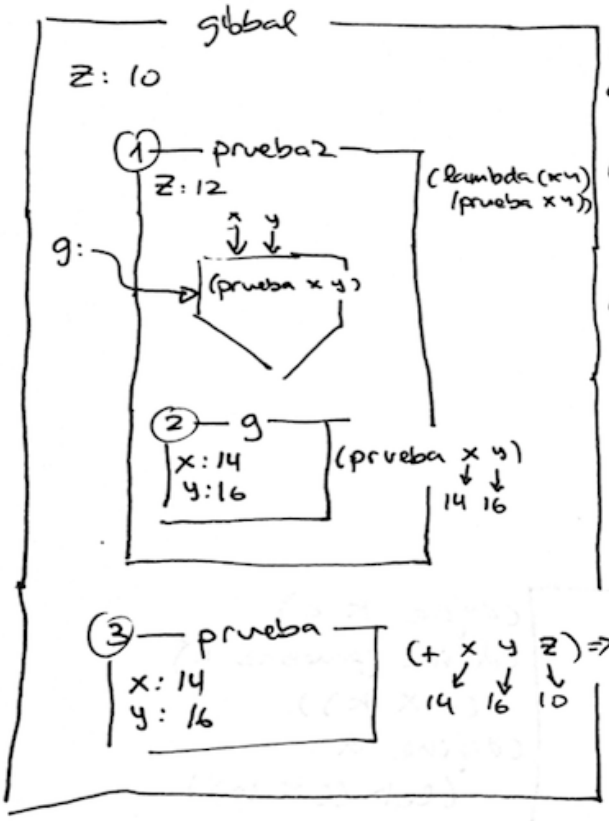
Solución:



b) (1 punto)

```
(define z 10)
(define (prueba x y)
   (+ x y z))
(define (prueba2 z)
   (lambda (x y)
      (prueba x y)))
(define g (prueba2 12))
(g 14 16)
```

Solución:




```
(define z 10)
(define (prueba x y)
  (+ x y z))
(define (prueba2 z)
  (lambda (x y)
    (prueba x y)))
(define g (prueba2 12))
(g 14 16) => 40
```

* Resultado: 40
* 3 ámbitos
* Sí, la función ligada a "g" es una clausura