Mañana

Ejercicio 1 (1,5 puntos)

a) (0,75 puntos) Realiza la evaluación de las siguientes expresiones, utilizando el orden normal e indicando al final el resultado:

```
(define (g x y)
   (+ (* x y) y))
(define (f a b)
   (g (+ a b) a))
(g (f 3 2) (- 3 2))
```

Solución:

```
(g (f 3 2) (- 3 2))
(+ (* (f 3 2) (- 3 2)) (- 3 2))
(+ (* (g (+ 3 2) 3) (- 3 2)) (- 3 2))
(+ (* (+ (* (+ 3 2) 3) 3) (- 3 2)) (- 3 2))
(+ (* (+ (* 5 3) 3) (- 3 2)) (- 3 2))
(+ (* (+ 15 3) (- 3 2)) (- 3 2))
(+ (* 18 (- 3 2)) (- 3 2))
(+ (* 18 1) (- 3 2))
(+ 18 (- 3 2))
(+ 18 1)
19
```

b) (0,75 puntos) Indica qué devuelven las siguientes expresiones:

```
(map (lambda (x y) (list x y)) '(a b c) '(j k l)) →
((a j) (b k) (c l))
((lambda (a f b) (f a b)) 2 - 4) → -2
((lambda (arg) arg) (lambda () 5)) → #<procedure>
```

Ejercicio 2 (1,5 puntos)

Escribe la función `(divide-preds pred1 pred2 lista)` que reciba una lista y dos predicados y devuelva una pareja donde su parte izquierda contiene una lista con los elementos de la lista que cumplen el `pred1` y la parte derecha contiene los que cumplen el `pred2`.

```
(divide-preds (lambda (x) (> x 5)) (lambda (x) (odd? x)) '(1 2 3 4 5 6 7 8 9)) → ((6 7 8 9).(1 3 5 7 9))
```

Solución (versión 1):

```
(define (divide-preds p1 p2 lista)
  (if (null? lista) (cons '() '())
      (let ((pareja (divide-preds p1 p2 (cdr lista))))
```

```
            (cond ((and (p1 (car lista)) (p2 (car lista))))
                        (cons (cons (car lista) (car pareja))
                              (cons (car lista) (cdr pareja))))
                  ((p1 (car lista))
                        (cons (cons (car lista) (car pareja))
                              (cdr pareja)))
                  ((p2 (car lista))
                        (cons (car pareja)
                              (cons (car lista) (cdr pareja)))
                  (else pareja)))))
```

Solución (versión 2, con una función auxiliar):

```
(define (divide-preds p1 p2 lista)
  (if (null? lista)
      (cons '() '())
      (let ((pareja (divide-preds p1 p2 (cdr lista))))
        (cons (añade-si p1 (car lista) (car pareja))
              (añade-si p2 (car lista) (cdr pareja))))))

(define (añade-si pred x lista)
  (if (pred x)
      (cons x lista)
      lista))
```

Ejercicio 3 (1,5 puntos)

Define una función recursiva llamada `ocurrencias-elementos` que tome como argumentos dos listas y devuelva una lista de parejas, en donde cada pareja contiene en su parte izquierda un elemento de la segunda lista y en su parte derecha el número de veces que aparece dicho elemento en la primera lista. Puedes definir las funciones auxiliares que necesites.

Ejemplo :
`(ocurrencias-elementos '(1 3 6 2 4 7 3 9 7) '(5 2 3)) →`
`((5 . 0) (2 . 1) (3 . 2))`

Solución:

```
(define (veces dato lista)
  (cond
    ((null? lista) 0)
    ((equal? dato (car lista))
     (+ 1 (veces dato (cdr lista))))
    (else (veces dato (cdr lista)))))

(define (ocurrencias-elementos lista elementos)
  (if (null? elementos)
      '()
      (cons (cons (car elementos)
```

```
            (veces (car elementos) lista))
        (ocurrencias-elementos lista (cdr elementos)))))
```

Ejercicio 5 (2 puntos)

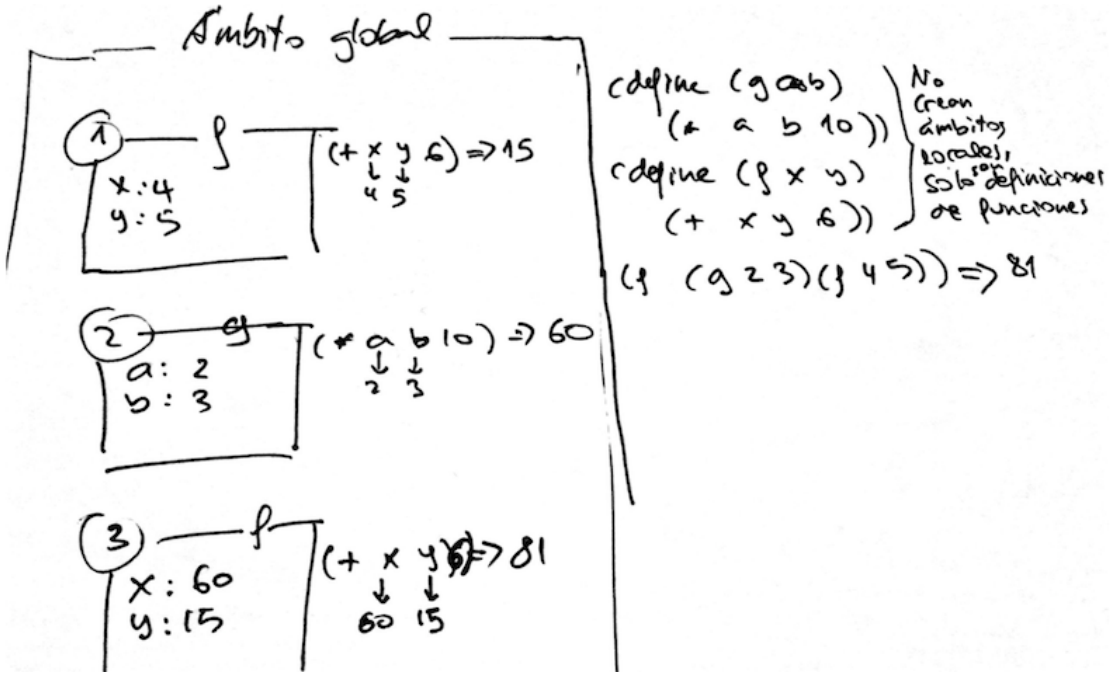
a) (1 punto) Supongamos el siguiente código en Scheme

```
(define (g a b)
   (* a b 10))
(define (f x y)
   (+ x y 6))
(f (g 2 3) (f 4 5))
```

Dibuja en un diagrama los entornos que se generan. Junto a cada entorno escribe un número indicando en qué orden se ha creado. Explica la evaluación de las expresiones haciendo referencia a esos números. ¿Cuál es el resultado? ¿Cuántos ámbitos se crean? ¿Se crea alguna clausura?

Solución:

Se crean 3 ámbitos, el resultado es 81 y no se crea ninguna clausura.



b) (1 punto) Supongamos el siguiente código en Scheme

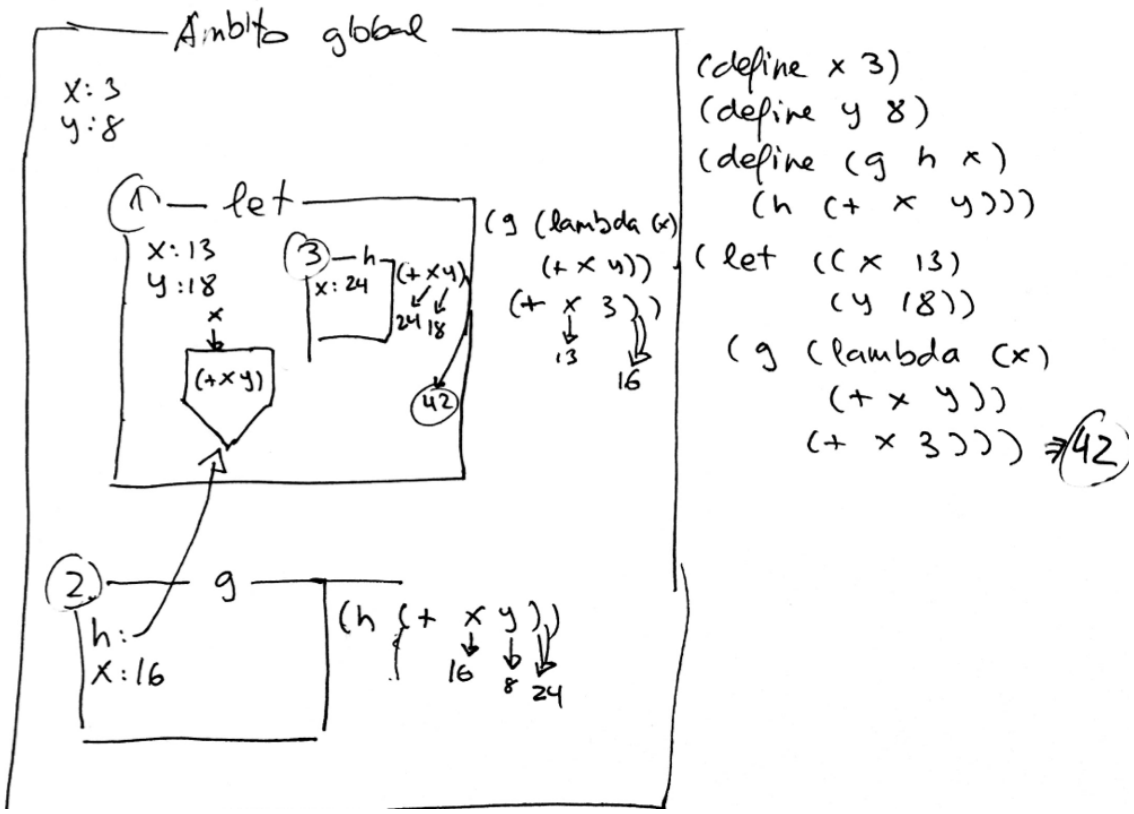
```
(define x 3)
(define y 8)
(define (g h x)
   (h (+ x y)))
(let ((x 13)
      (y 18))
```

```
(g  (lambda (x) (+ x y)) (+ x 3)))
```

Junto a cada entorno escribe un número indicando en qué orden se ha creado. Explica la evaluación de las expresiones haciendo referencia a esos números. ¿Cuál es el resultado? ¿Cuántos ámbitos se crean? ¿Se crea alguna clausura?

Solución:

Se crean 3 ámbitos, el resultado es 42 y se crea una clausura, creada en el ámbito del let y pasada como parámetro h en la invocación de g.



Ejercicio 6 (2 puntos)

b) (0,2 puntos) El uso de la abstracción en los programas permite (rodea las opciones que sean correctas)

1) Facilitar la visualización del funcionamiento del programa y del sistema software
2) Implementar software que corre de una forma más eficiente y rápida
3) Encontrar más fácilmente los errores en un programa
4) Obtener software más modular en el que las partes pueden ser construidas, reemplazadas y probadas por separado

c) (0,2 puntos) El nombre de Lisp viene de (rodea la correcta):

1) Lambda Processing
2) Lots of Insipid and Stupid Parenthesis

3) List Instance Sentence Programs
4) List Processing

d) (0,2 puntos) ¿Cuál de las siguientes afirmaciones son ciertas (puede haber más de una)?

a) En Pascal es posible pasar funciones como argumentos de otras funciones, pero hay que especificar el tipo de la función, qué tipos recibe y devuelve.
b) En Pascal es posible pasar funciones como argumentos de otras funciones, y el tipo de la función se infiere en tiempo de compilación.
c) En C y C++ se pueden pasar como parámetros o asignar a variables punteros a funciones.
d) Las funciones son objetos de primera clase en C y C++.
e) En Objective-C las clausuras se denominan *bloques*

e) (0,2 puntos) Un *l-value* es (rodea la correcta):

a) Un puntero o referencia que se guarda en una variable
b) Un valor constante e inmutable que podemos usar en programación funcional
c) Una expresión en la que puede guardarse un valor o una referencia
d) Una expresión que produce el valor utilizado en una asignación

Tarde

Ejercicio 1 (1,5 puntos)

a) (0,75 puntos) Realiza la evaluación de las siguientes expresiones, utilizando el *orden normal* e indicando al final el resultado:

```
(define (f a b)
  (g (+ a b) a))
(define (g x y)
  (+ (* x y) y))
(f (- 3 2) (g 3 2))
```

Solución:

```
(f (- 3 2) (g 3 2))
(g (+ (- 3 2) (g 3 2)) (- 3 2))
(+ (* (+ (- 3 2) (g 3 2)) (- 3 2)) (- 3 2))
(+ (* (+ (- 3 2) (+ (* 3 2) 2)) (- 3 2)) (- 3 2))
(+ (* (+ (- 3 2) (+ 6 2)) (- 3 2)) (- 3 2))
(+ (* (+ 1 (+ 6 2)) (- 3 2)) (- 3 2))
(+ (* (+ 1 8) (- 3 2)) (- 3 2))
(+ (* 9 (- 3 2)) (- 3 2))
(+ (* 9 1) (- 3 2))
(+ 9 (- 3 2))
(+ 9 1)
10
```

b) (0,75 puntos) Indica qué devuelven las siguientes expresiones:

```
(define (misterio lista n)
  (map (lambda (x) (+ n x)) lista))
(misterio '(3 6 8) 4) → (7 10 12)

((lambda (a f b) (f a b)) 5 + 7) → 12

((lambda (arg) (arg)) (lambda () 5)) → 5
```

Ejercicio 2 (1,5 puntos)

Escribe la función `(cuenta-preds pred1 pred2 lista)` que reciba una `lista` y dos predicados y devuelva una pareja donde su parte izquierda contiene el número de elementos de la lista que cumplen el `pred1` y la parte derecha contiene los que cumplen el `pred2`.

```
(cuenta-preds (lambda (x) (> x 5)) (lambda (x) (odd? x)) '(1 2 3 4 5 6 7 8 9)) → (4.5)
```

Solución:

```
 (define (cuenta-preds p1 p2 lista)
       (if (null? lista) (cons 0 0)
              (incr-pareja (cuenta-preds p1 p2 (cdr lista))
                           (p1 (car lista))
                           (p2 (cdr lista)))))

(define (incr-pareja pareja left right)
       (cons (+ (car pareja) (if left 1 0))
              (+ (cdr pareja (if right 1 0)))))
```

Ejercicio 3 (1,5 puntos)

Define la función (`mas-frecuentes lista1 lista2 n`) que reciba dos listas y un número. Devolverá una nueva lista que contenga aquellos elementos de la segunda lista que aparecen más de n veces en la primera. Puedes definir las funciones auxiliares que necesites.

Ejemplo:
`(mas-frecuentes '(a b c d d 1 3 3 a b c c d d) '(a d b c) 3)` → `'(d c)`
`(mas-frecuentes '(a b c d d 1 3 3 a b c c d d) '(a d b c) 5)` → `'()`

Solución:

```
(define (veces dato lista)
  (cond
    ((null? lista) 0)
    ((equal? dato (car lista))
     (+ 1 (veces dato (cdr lista))))
    (else (veces dato (cdr lista)))))

(define (mas-frecuentes lista elementos n)
  (if (null? elementos)
      '()
      (if (> (veces (car elementos) lista) n)
          (cons (car elementos)
                (mas-frecuentes lista (cdr elementos) n))
          (mas-frecuentes lista (cdr elementos) n))))
```

Ejercicio 5 (2 puntos)

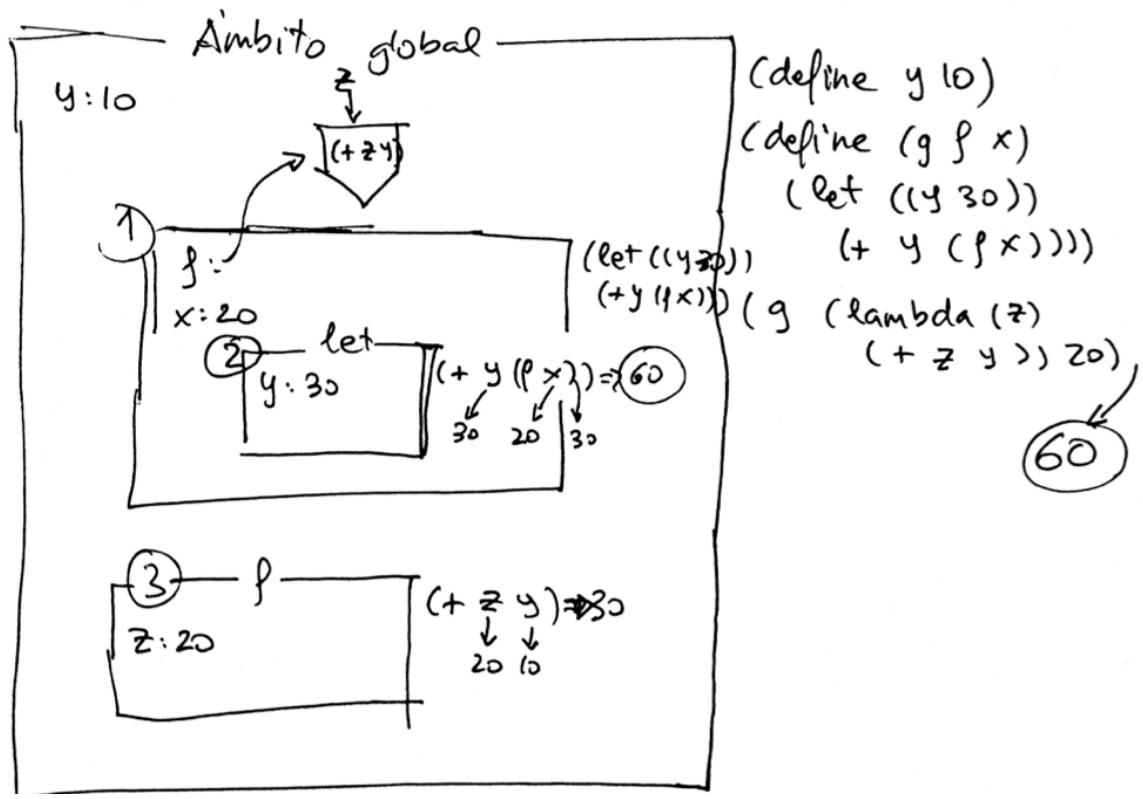
b) (1 punto) Supongamos el siguiente código en Scheme

```
(define y 10)
(define (g f x)
  (let ((y 30))
     (+ y (f x))))
(g (lambda (z) (+ z y)) 20)
```

Junto a cada entorno escribe un número indicando en qué orden se ha creado. Explica la evaluación de las expresiones haciendo referencia a esos números. ¿Cuál es el resultado? ¿Cuántos ámbitos se crean? ¿En qué orden? ¿Se crea alguna clausura?

Solución:

Se crean 3 ámbitos, el resultado es 60 y no se crea ninguna clausura porque la función f se define en el ámbito global.



Ejercicio 6 (2 puntos)

b) (0,2 puntos) En Scheme, un ejemplo de abstracción es (rodea las opciones que sean correctas):

a) La posibilidad de definir funciones con la forma especial define
b) El uso de expresiones bien delimitadas por paréntesis
c) La posibilidad de usar guiones en los nombres de los identificadores
d) La definición del tipo de datos lista, y sus funciones list, list?, length, null?, list-ref, etc.
e) El uso de la forma especial lambda para crear funciones anónimas

c) (0,2 puntos) El nombre de Lisp viene de (rodea la correcta):

a) Lambda Processing
b) Lots of Insipid and Stupid Parenthesis
c) List Instance Sentence Programs
d) List Processing

d) (0,2 puntos) Un *side effect* se produce en (rodea las opciones que sean correctas):

a) Los lenguajes de programación mal diseñados como el BASIC
b) Los lenguajes von Neumann basados en sentencias y asignación en memoria
c) Los lenguajes de programación funcionales
d) Cualquier lenguaje completo equivalente a una Máquina de Turing
e) Cualquier lenguaje imperativo

e) (0,2 puntos) Un *l-value* es (rodea la correcta):

a) Un puntero o referencia que se guarda en una variable
b) Un valor constante e inmutable que podemos usar en programación funcional
c) Una expresión en la que puede guardarse un valor o una referencia
d) Una expresión que produce el valor utilizado en una asignación