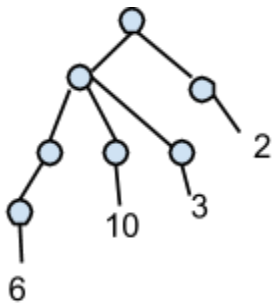
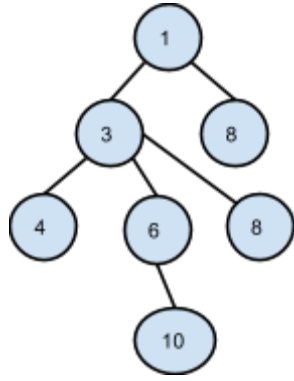


## Mañana

**b) (0,75 puntos)** Dadas las siguientes estructuras de datos recursivas, escribe su expresión correspondiente en forma de lista estructurada y las instrucciones, utilizando la barrera de abstracción adecuada, para obtener el elemento 10 de cada una.

<p>Pseudoárbol:</p> 	<p>Lista estructurada:</p> <pre>(define lista '((((6)) (10) (3)) (2)))</pre> <p>Elemento 10:</p> <pre>(car(car(cdr(car lista))))</pre>
<p>Árbol genérico:</p> 	<p>Lista estructurada:</p> <pre>(define tree '(1 (3 (4) (6 (10))(8))(8)))</pre> <p>Elemento 10:</p> <pre>(dato-tree (car (hijos-tree (cadr (hijos-tree (car (hijos-tree tree)))))))</pre>

## Ejercicio 2 (1,5 puntos)

Implementa en Scheme la función recursiva `suma-listas` que reciba dos listas estructuradas con la misma estructura y devuelva una lista plana con las sumas de todos sus elementos:

```
(suma-listas '(1 2 (3 (4) (5 (6 7))) 3)
              '(4 2 (1 (8) (7 (2 3))) 2)) => (5 4 4 12 12 8 10 5)
```

Solución (dos versiones):

```
(define (suma-listas-v1 lista1 lista2)
  (cond
    ((null? lista1) '())
```

```

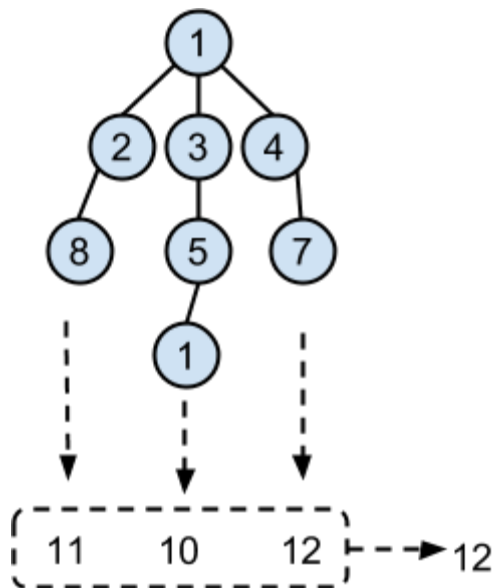
((hoja? lista1) (list (+ lista1 lista2)))
(else
 (append (suma-listas-v1 (car lista1) (car lista2))
          (suma-listas-v1 (cdr lista1) (cdr lista2)))))

(define (suma-listas-v2 lista1 lista2)
  (cond
    ((null? lista1) '())
    ((hoja? (car lista1)) (cons (+ (car lista1) (car lista2))
                                (suma-listas-v2 (cdr lista1) (cdr lista2))))
    (else (append (suma-listas-v2 (car lista1) (car lista2))
                  (suma-listas-v2 (cdr lista1) (cdr lista2)))))

```

#### Ejercicio 4 (1,5 puntos)

Utilizando recursión mutua o funciones de orden superior, define en Scheme la función `suma-max-tree` que reciba un árbol genérico y devuelva, de la suma de los nodos de cada rama, aquella que es máxima. Ejemplo:



Solución:

```

(define (suma-max-tree tree)
  (+ (dato-tree tree) (suma-max-bosque (hijos-tree tree))))

(define (suma-max-bosque bosque)
  (if (null? bosque) 0
      (max (suma-max-tree (car bosque))
            (suma-max-bosque (cdr bosque)))))

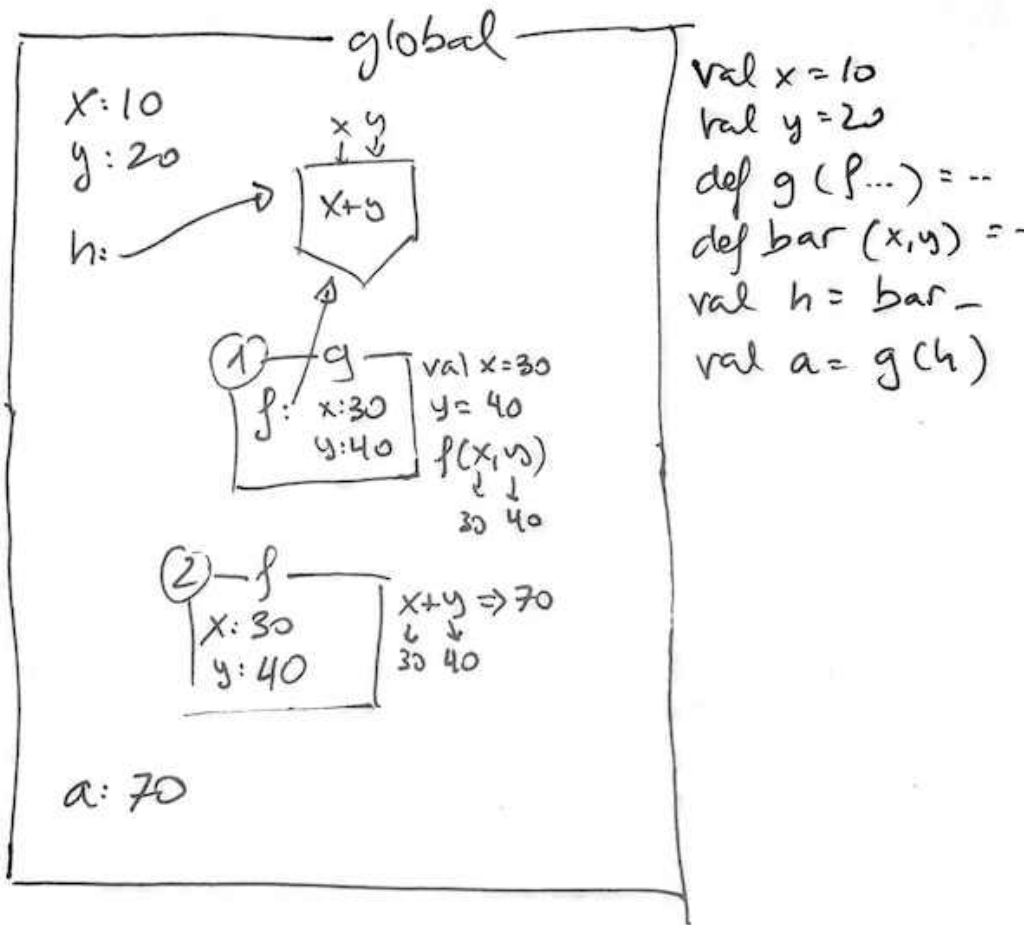
```

```
(max (suma-max-tree (car bosque))  
      (suma-max-bosque (cdr bosque))))
```

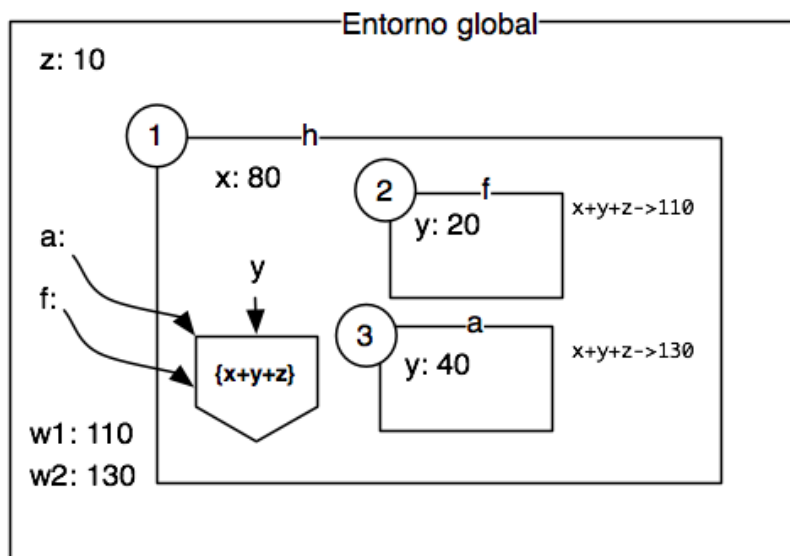
### Ejercicio 6 (2 puntos)

**a) (1 punto)** Dibuja el diagrama de ámbitos que se crea con la evaluación de las siguientes instrucciones en Scala:

```
val x = 10  
val y = 20  
def g(f:(Int,Int)=>Int) = {  
    val x = 30  
    val y = 40  
    f(x,y)  
}  
def bar(x:Int,y:Int) = x+y  
val h = bar _  
val a = g(h)
```



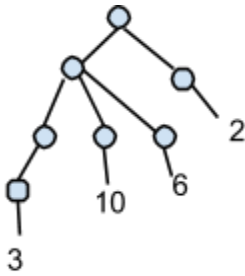
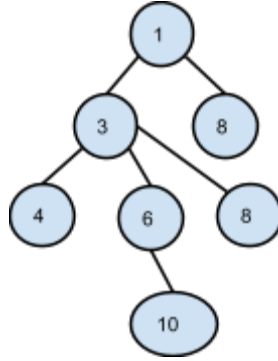
b) (1 punto) Dado el siguiente diagrama de ámbitos, escribe las instrucciones en Scala que lo crean:



```
val z = 10
def h(x:Int) = {
  (y:Int) => {x+y+z}
}
val a = h(80)
val f = a
val w1 = f(20)
val w2 = a(40)
```

## Tarde

**b) (0,75 puntos)** Dadas las siguientes estructuras de datos recursivas, escribe su expresión correspondiente en forma de lista estructurada y las instrucciones, utilizando la barrera de abstracción adecuada, para obtener el elemento 6 de cada una:

<p>Pseudoárbol:</p> 	<p>Lista estructurada:</p> <pre>(define lista '((((3))(10)(6))(2)))</pre> <p>Elemento 6:</p> <pre>(car(car(cdr(cdr(car lista)))))</pre>
<p>Árbol genérico:</p> 	<p>Lista estructurada:</p> <pre>(define tree '(1 (3 (4)(6 (10))(8))(8)))</pre> <p>Elemento 6:</p> <pre>(dato-tree (cadr (hijos-tree (car (hijos-tree tree)))))</pre>

## Ejercicio 2 (1,5 puntos)

Implementa en Scheme la función recursiva `compara-listas` que reciba dos listas estructuradas con la misma estructura y devuelva una lista plana con booleanos que indiquen si el elemento de la segunda lista es mayor o igual que el de la primera:

```
(compara-listas '(1 2 (3 (4) (5 (6 7))) 3)
                 '(4 2 (1 (8) (7 (2 3))) 2)) => (#t #t #f #t #t #f #f #f)
```

Solución (dos versiones):

```

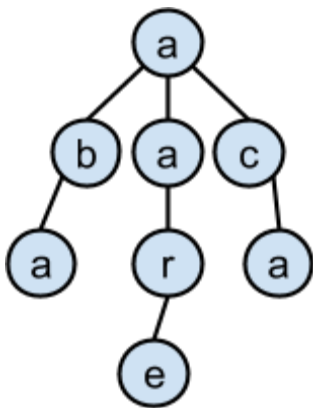
(define (compara-listas-v1 lista1 lista2)
  (cond
    ((null? lista1) '())
    ((hoja? lista1) (list (>= lista2 lista1)))
    (else
     (append (compara-listas-v1 (car lista1) (car lista2))
              (compara-listas-v1 (cdr lista1) (cdr lista2))))))

(define (compara-listas-v2 lista1 lista2)
  (cond
    ((null? lista1) '())
    ((hoja? (car lista1)) (cons (>= (car lista2) (car lista1))
                                (compara-listas-v2 (cdr lista1)
                                                      (cdr lista2)))))
    (else (append (compara-listas-v2 (car lista1) (car lista2))
                  (compara-listas-v2 (cdr lista1) (cdr lista2))))))

```

#### Ejercicio 4 (1,5 puntos)

Utilizando recursión mutua o funciones de orden superior, define en Scheme la función `cuenta-elem-tree` que reciba un árbol genérico y un elemento como argumentos, y devuelva el número de veces que aparece el elemento en los nodos del árbol. Ejemplo:



`(cuenta-elem-tree tree 'a) → 4`

**Solución:**

```

(define (cuenta-elem-tree tree x)
  (if (equal? (dato-tree tree) x)
      (+ 1 (cuenta-elem-bosque (hijos-tree tree) x))
      (cuenta-elem-bosque (hijos-tree tree) x)))

```

```

(define (cuenta-elem-bosque bosque x)
  (if (null? bosque) 0
      (+ (cuenta-elem-tree (car bosque) x)
         (cuenta-elem-bosque (cdr bosque) x))))

```

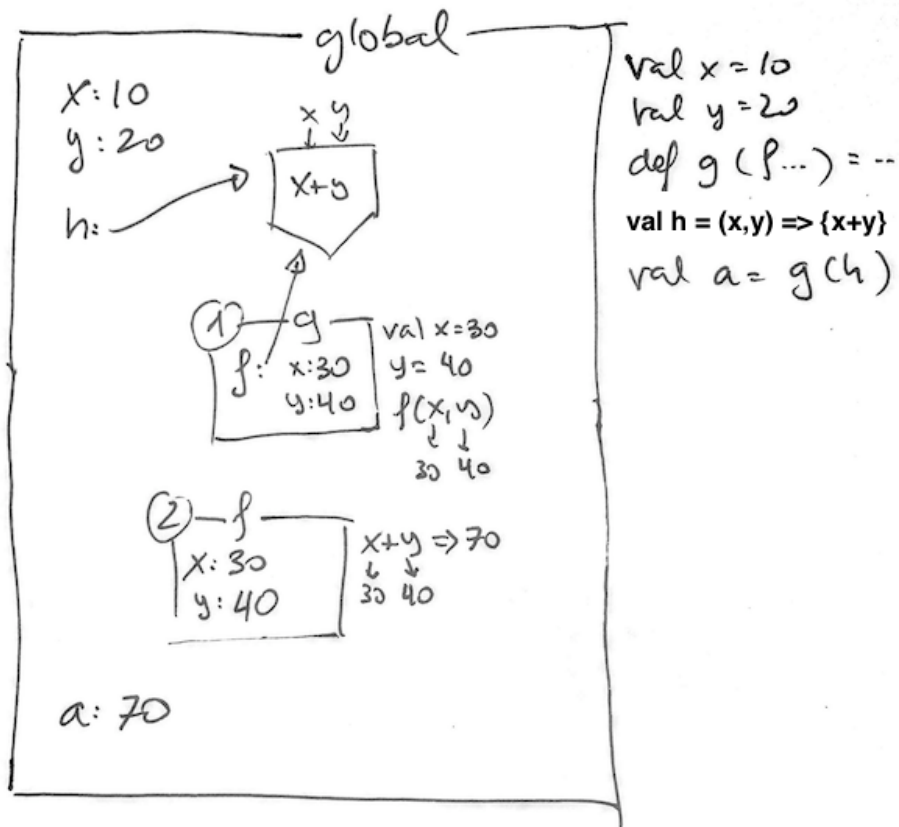
## Ejercicio 6 (2 puntos)

**a) (1 punto)** Dibuja el diagrama de ámbitos que se crea con la evaluación de las siguientes instrucciones en Scala:

```

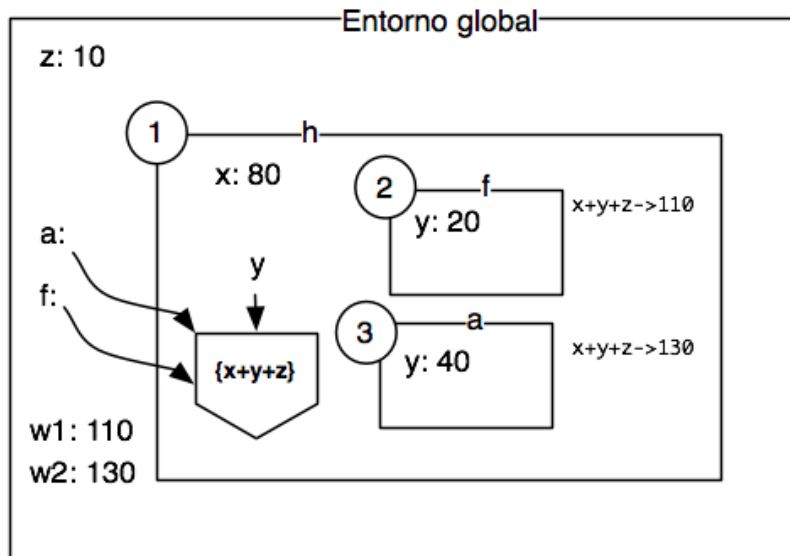
val x = 10
val y = 20
def g(f:(Int,Int)=>Int) = {
  val x = 30
  val y = 40
  f(x,y)
}
val h = (x,y) => {x+y}
val a = g(h)

```





**b) (1 punto)** Dado el siguiente diagrama de ámbitos, escribe las instrucciones en Scala que lo crean:



```
val z = 10
def h(x: Int) = {
  (y: Int) => {x+y+z}
}
val a = h(80)
val f = a
val w1 = f(20)
val w2 = a(40)
```

Nombre: \_\_\_\_\_ DNI: \_\_\_\_\_

## Lenguajes y Paradigmas de Programación

Curso 2013-2014

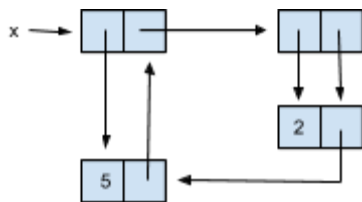
### Tercer parcial

#### 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 3 horas.

#### Ejercicio 1 (1 punto)

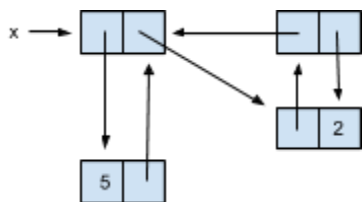
a) (0,5 punto) Escribe las instrucciones que generan el siguiente diagrama *box & pointer*:



Solución:

```
(define p1 (cons 5 '()))
(define p2 (cons 2 p1))
(define p3 (cons p2 p2))
(define x (cons p1 p3))
(set-cdr! p1 x)
```

b) (0,5 puntos) Dado el diagrama anterior, escribe las instrucciones que lo han modificado (utilizando como única referencia la x y sin añadir ningún valor ni pareja nueva) de la siguiente forma:



Solución:

```
(define p1 (cons 5 '()))
(define p2 (cons 2 p1))
(define p3 (cons p2 p2))
(define x (cons p1 p3))
```

```
(set-cdr! p1 x)
```

## Ejercicio 2 (2 puntos)

Definimos en Scheme una estructura de datos formada por una lista enlazada con cabecera que contiene parejas con el dato y la posición que el dato tiene en la lista, sin elementos repetidos. Por ejemplo, la lista que contiene los datos 'a, 'z, 'c, 'd situados en la posición 1, 2, 3 y 4 se construiría de la siguiente forma:

```
(define lis (list '*cab* (cons 'a 1) (cons 'z 2) (cons 'c 3) (cons 'd 4)))  
⇒ (*cab* (a . 1) (z . 2) (c . 3) (d . 4))
```

Escribe la función mutadora (`delete! x lista`) que busca un elemento `x` en la lista y lo elimina de ella, mutando las posiciones de todos los que hay a continuación. Si el elemento no está en la lista, debe devolver el símbolo 'no-encontrado. Puedes utilizar o no la barrera de abstracción de las listas ordenadas con cabecera. Puedes implementar las funciones auxiliares que necesites.

```
(delete! 'z lis)  
lis  
⇒ (*cab* (a . 1) (c . 2) (d . 3))  
(delete! 'z lis)  
⇒ 'no-encontrado
```

```
(define (delete! x lista)  
  (cond  
    ((null? (cdr lista)) 'no-encontrado)  
    ((equal? x (caadr lista))  
     (begin  
       (set-cdr! lista (cddr lista))  
       (restar-uno! (cdr lista))))  
    (else (delete! x (cdr lista)))))  
  
(define (restar-uno-lista! lista)  
  (if (not (null? lista))  
      (begin  
        (set-cdr! (car lista) (- (cdar lista) 1))  
        (restar-uno! (cdr lista))))  
      ))
```