

Nombre: _____

Lenguajes y Paradigmas de Programación

Curso 2014-2015

Segundo 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 2 horas.

Ejercicio 1 (2,5 puntos)

a) (0,5 puntos) Dada la siguiente función en Scheme que define la serie de Fibonacci utilizando la técnica de *memoization*:

```
(define (fib-memo n lista)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        ((not (null? (get n lista)))
         (put n lista))
        (else (let ((result
                     (+ (fib-memo (- n 1) lista)
                        (fib-memo (- n 2) lista))))
                 (begin
                  (put n lista)
                  result))))))
```

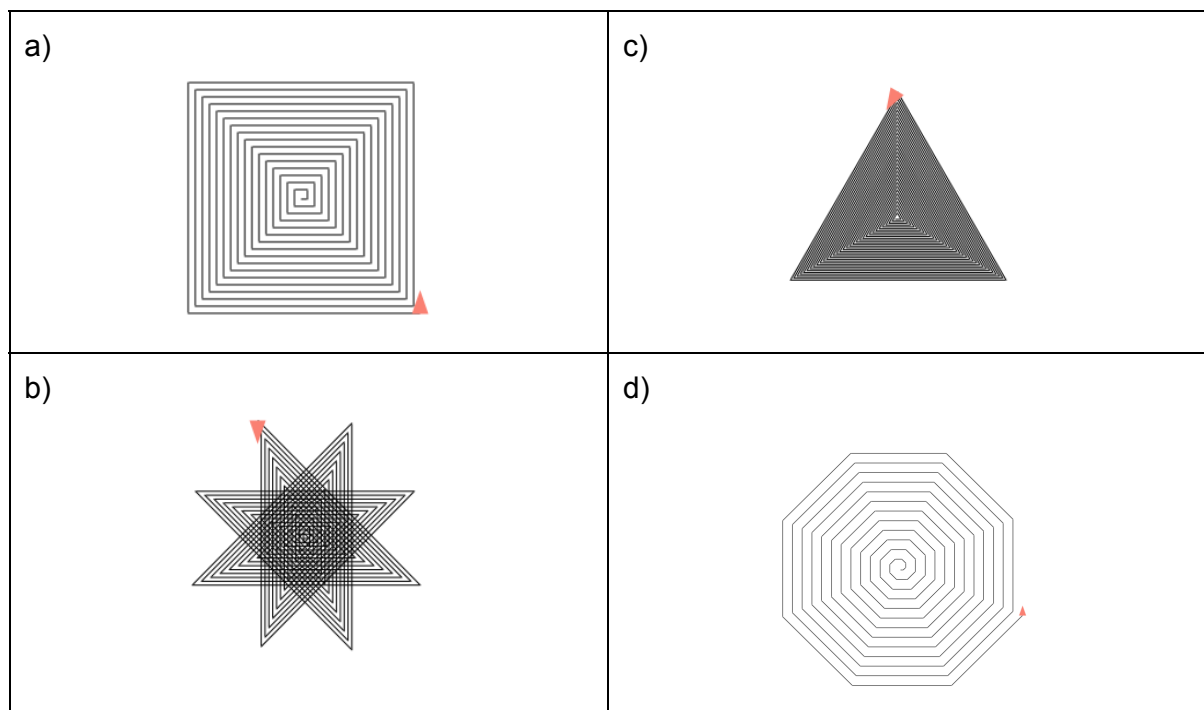
La función `fib-memo` contiene errores. Descríbelos y corrígelos.

- El primer "(put n lista)" debe ser un "(get n lista)" porque hemos comprobado que el resultado de la recursión está guardado en la tabla hash y tenemos que devolverlo.
- El segundo "(put n lista)" debe ser un "(put n result lista)" para guardar en la tabla hash el resultado obtenido de la recursión y poder utilizarlo posteriormente.

b) (0,5 puntos) Dada la siguiente función en Scheme utilizando gráficos de tortuga:

```
(define (figura lado angulo)
  (if (< lado 200)
      (begin
        (draw lado)
        (turn angulo)
        (figura (+ lado 2) angulo))))
```

Indica qué se dibuja con la llamada (figura 5 45)



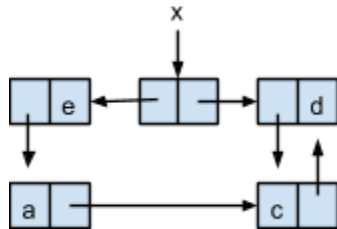
Solución: Figura d)

c) (0,5 puntos) Explica brevemente qué soluciones existen para mejorar el coste de la recursión. Ventaja e inconveniente de cada solución.

- **Memoization:** se guardan en alguna estructura de datos los resultados de las llamadas recursivas. Ventaja: no hay que cambiar el algoritmo recursivo puro, lo único que hay que hacer es añadir el código para guardar y recuperar los resultados. Inconveniente: coste espacial (memoria necesaria para guardar todos los resultados de la recursión).
- **Recursión por la cola (tail recursion):** se modifica el algoritmo recursivo para evitar las llamadas en espera, añadiendo un parámetro adicional en el que se va calculando el resultado. Ventaja: no se incurre en ningún coste espacial. Inconveniente: el algoritmo resultante es menos claro que el recursivo puro.

d) (0,5 puntos)

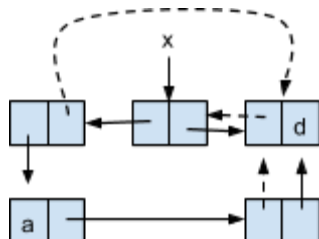
d.1) Escribe las instrucciones necesarias que generan el siguiente box & pointer:



Una posible solución (hay muchas):

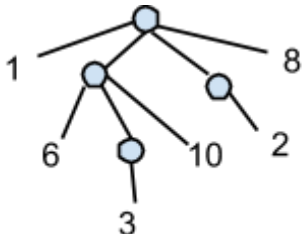
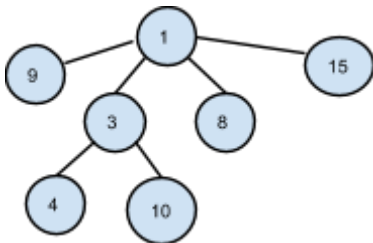
```
(define p1 (cons (cons 'a
                        (cons 'c
                              (cons 'borrar 'd)))
                  'e))
(define x (cons p1 (cdr (cdr (car p1)))))
(set-car! (cdr x) (cdr (car (car x))))
```

d.2) Escribe las instrucciones necesarias que realizan los siguientes cambios (utilizando como única referencia la x)



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

e) (0,5 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 '(1 (6 (3) 10) (2) 8))</pre> <p>Elemento 10:</p> <pre>(car (cdr (cdr (car (cdr lista)))))</pre> |
| <p>Árbol:</p>  | <p>Lista estructurada:</p> <pre>(define arbol '(1 (9) (3 (4) (10)) (8) (15)))</pre> <p>Elemento 10:</p> <pre>(dato-tree (cadr (hijos-tree (cadr (hijos-tree arbol)))))</pre> |

Ejercicio 2 (1,5 puntos)

Escribe **utilizando recursión por la cola** la función (cuadrado-lista lista) que toma como argumento una lista de números y devuelve una lista con sus cuadrados.

Ejemplo:

```
(cuadrado-lista '(2 3 4 5)) ⇒ (4 9 16 25)
```

[illegible]

Ejercicio 3 (3 puntos)

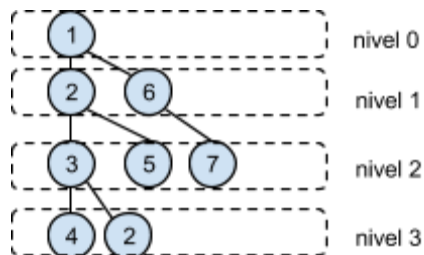
a) (1,5 puntos) Escribe la función `(cuenta-diff-listas l1 l2)` que toma como argumentos dos listas estructuradas con la misma estructura pero con diferentes elementos y devuelve el número de elementos diferentes.

Ejemplos:

```
(cuenta-diff-listas '(a (b ((c)) d e) f) '(1 (b ((2)) 3 4) f)) ⇒ 4  
(cuenta-diff-listas '((a b) c) '((a b) c)) ⇒ 0
```

```
(define (cuenta-diff-listas l1 l2)  
  (cond ((null? l1) 0)  
        ((hoja? l1) (if (equal? l1 l2)  
                        0  
                        1))  
        (else (+ (cuenta-diff-listas (car l1) (car l2))  
                  (cuenta-diff-listas (cdr l1) (cdr l2))))))
```

b) (1,5 puntos) Escribe la función `(lista-nivel-tree nivel tree)` que reciba un nivel y un árbol y devuelva una lista con todos los nodos que se encuentran en ese nivel.



```
(lista-nivel-tree 0 tree) ⇒ (1)  
(lista-nivel-tree 1 tree) ⇒ (2 6)  
(lista-nivel-tree 2 tree) ⇒ (3 5 7)  
(lista-nivel-tree 3 tree) ⇒ (4 2)
```

```
(define (lista-nivel-tree n tree)  
  (if (= n 0) (list (dato tree))  
      (lista-nivel-bosque (- n 1) (hijos-tree tree))))
```

```
(define (lista-nivel-bosque n bosque)  
  (if (null? bosque) '()  
      (append (lista-nivel-tree n (car bosque))  
                (lista-nivel-bosque n (cdr bosque)))))
```

Ejercicio 4 (1,5 puntos)

Implementa el procedimiento mutador (`intercambia-elementos! lista`) que reciba una lista con cabecera con un número par de elementos e intercambie sus elementos de dos en dos. Debes proponer una solución que no utilice `set-car!`

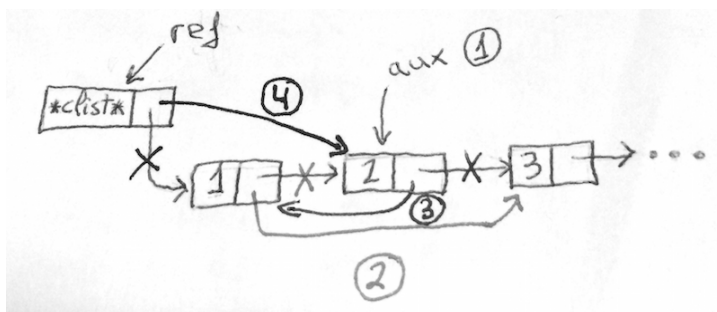
Además **explica utilizando diagramas de caja y puntero** el funcionamiento de tu solución.

Ejemplos:

```
(define lista '(*clist* 1 2 3 4))  
(intercambia-elementos! lista)  
lista ⇒ (*clist* 2 1 4 3)
```

```
(define (intercambia-siguiente! ref)  
  (let ((aux (cddr ref)))      ;; 1  
    (begin  
      (set-cdr! (cdr ref) (cdr aux)) ;; 2  
      (set-cdr! aux (cdr ref))      ;; 3  
      (set-cdr! ref aux))))        ;; 4
```

```
(define (intercambia-elementos! lista)  
  (if (or (null? (cdr lista))  
        (null? (cddr lista)))  
      lista  
      (begin  
        (intercambia-siguiente! lista)  
        (intercambia-elementos! (cddr lista))))))
```



Ejercicio 5 (1,5 puntos)

a) (0,75 puntos) Escribe la definición de la función (foo x y) para que el siguiente código funcione correctamente y devuelva los resultados indicados.

```
(define (foo x y)
  ...
)
```

```
(define f (foo 10 20))
(define g (foo 1 2))
```

(f 10) ⇒ 40

(f 5) ⇒ 45

(g 10) ⇒ 13

(g 5) ⇒ 18

```
(define (foo x y)
  (let ((suma (+ x y)))
    (lambda (x)
      (set! suma (+ suma x))
      suma)))
```

b) (0,75 puntos) Dibuja los ámbitos creados en la ejecución del código anterior, considerando sólo la primera invocación a f y a g.

