Nombre:_____

Lenguajes y Paradigmas de Programación
Curso 2015-16
Segundo parcial

Normas importantes

- **Se debe contestar cada pregunta en las hojas que entregamos. Utiliza la última hoja para hacer pruebas. No olvides poner el nombre. Puedes usar lápiz.**
- **El profesor que está en el aula no contestará ninguna pregunta, salvo aquellas que se refieran a posibles errores en los enunciados de los ejercicios.**
- **La duración del examen es de 2 horas.**

Ejercicio 1 (1,75 puntos)

Escribe la **función recursiva** (pasar-a-impares lista) que recibe una lista estructurada de números enteros y devuelve otra lista con la misma estructura, pero en la que se han incrementado en uno todos los números pares. Debes usar la barrera de abstracción de listas estructuradas (no hace falta que la implementes).

Ejemplo:

(pasar-a-impares '(4 7 3 (9 5 (1 2) 8) 10 6)) ⇒ {5 7 3 {9 5 {1 3} 9} 11 7}

```
(define (pasar-a-impares lista)
  (cond ((null? lista)
         '())
        ((hoja? lista)
         (if (even? lista)
             (+ 1 lista)
             lista))
        (else
         (cons (pasar-a-impares (car lista))
               (pasar-a-impares (cdr lista))))))
```

Ejercicio 2 (1,75 puntos)

Escribe la función (veces-tree arbol dato) que recibe un árbol de símbolos y un dato (otro símbolo) y recorre el árbol devolviendo el número de veces que aparece el dato en el árbol. Debes usar las funciones de la barrera de abstracción de árboles (no hace falta que las implementes). Puedes usar funciones de orden superior y/o recursión mutua.

Ejemplo:

(define arbol '(a (b (c) (a)) (c) (d (c) (b))))
(veces-tree arbol 'b) ⇒ 2
(veces-tree arbol 'f) ⇒ 0

Versión 1 (recursión mutua):

```
(define (veces-tree arbol dato)
  (if (equal? dato (dato-tree arbol))
      (+ 1 (veces-bosque (hijos-tree arbol) dato))
      (veces-bosque (hijos-tree arbol) dato)))

(define (veces-bosque bosque dato)
  (if (null? bosque)
      0
      (+ (veces-tree (car bosque) dato)
         (veces-bosque (cdr bosque) dato))))
```

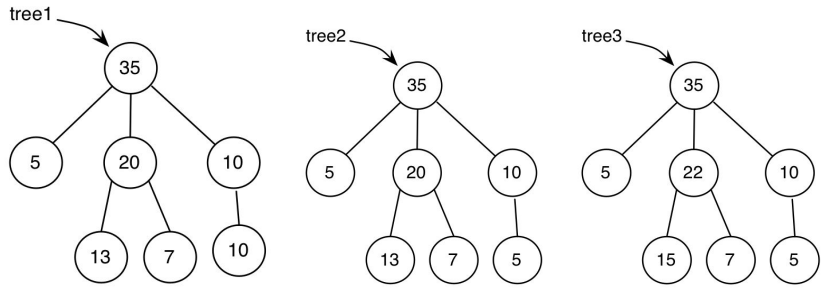
Versión 2 (veces-tree-FOS):

```
(define (comprueba-raiz tree dato)
 (if (equal? (dato-tree tree) dato)
     1
     0))

(define (veces-tree-FOS tree dato)
 (+ (comprueba-raiz tree dato)
    (fold-right + 0 (map (lambda(t)
          (veces-tree-FOS t dato)) (hijos-tree tree)))))
```

Ejercicio 3 (2 puntos)

Escribe el predicado (suma-hijos-tree? tree) que recibe un árbol de números enteros y recorre el árbol devolviendo #t en caso de que para todos sus nodos (excepto los nodos hoja) se cumpla que la suma de las raíces de los hijos coincida con el dato del nodo. Debes usar las funciones de la barrera de abstracción de árboles (no hace falta que las implementes). Puedes usar funciones de orden superior y/o recursión mutua.

Ejemplo:



(suma-hijos-tree? tree1) ⇒ #t
(suma-hijos-tree? tree2) ⇒ #f
(suma-hijos-tree? tree3) ⇒ #f

```
(define (suma-lista lista)
  (fold-right + 0 lista))

(define (suma-hijos-tree? tree)
  (if (hoja-tree? tree)
      #t
      (and (= (dato-tree tree)
              (suma-lista (map dato-tree (hijos-tree tree))))
           (suma-hijos-bosque? (hijos-tree tree)))))

(define (suma-hijos-bosque? bosque)
  (if (null? bosque)
      #t
      (and (suma-hijos-tree? (car bosque))
           (suma-hijos-bosque? (cdr bosque)))))
```

Versión FOS:

```
(define (and-lista lista)
  (fold-right (lambda(x y) (and x y)) #t lista))

(define (suma-hijos-tree-FOS? tree)
  (if (hoja-tree? tree)
      #t
      (and (= (dato-tree tree)
              (suma-lista (map dato-tree (hijos-tree tree))))
           (and-lista (map suma-hijos-tree-FOS? (hijos-tree tree))))))
```

Ejercicio 4 (2 puntos)

Escribe el **procedimiento mutador recursivo** (eliminar-claves! lista l-assoc), que recibe una lista mutable con cabecera y una lista de asociación, y elimina de la lista mutable las claves que no existen en la lista de asociación.

Puedes usar cualquier función estándar de Scheme, incluyendo las que trabajan con listas de asociación. No puedes usar las funciones de la barrera de abstracción de lista ordenada mutable, sino que debes realizar la implementación usando las sentencias mutadoras de Scheme.

Ejemplo:

```
(define lista '(*clista* a f g b h c))
(define l-assoc (list (cons 'a 1) (cons 'b 2) (cons 'c 3) (cons 'd 4)))
(eliminar-claves! lista l-assoc)
lista ⇒ {*clista* a b c}
```

```
(define (eliminar-claves! lista l-assoc)
  (if (not (null? (cdr lista)))
      (if (assq (cadr lista) l-assoc)
          (eliminar-claves! (cdr lista) l-assoc)
          (begin
            (set-cdr! lista (cddr lista))
            (eliminar-claves! lista l-assoc)
            ))))
```

Ejercicio 5 - tipo test (2,5 puntos)

Cada respuesta errónea **penaliza con 0,15 puntos**. En todas las preguntas, **sólo una respuesta es la correcta**. **Redondea la letra** con tu respuesta.

5.1) (0,4 puntos) Supongamos que la variable arbol contiene un árbol definido en Scheme, y que llamamos a la función hijos-tree con ese parámetro:

```
(define foo (hijos-tree arbol))
```

Supongamos las siguientes expresiones:

1. (map (lambda (x) (hijos-tree x)) foo)
2. (hoja-tree? (cadr foo))
3. (dato-tree (cdr foo))
4. (cdr (hijos-tree foo))

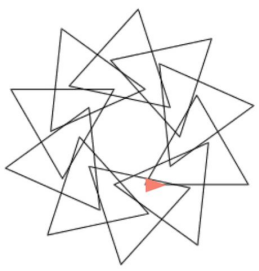
¿Qué expresiones de las anteriores usan correctamente la barrera de abstracción de árboles?

A. 1, 2 y 3: Correctas, 4: Incorrecta
B. 1 y 2: Correctas, 3 y 4: Incorrectas
C. 2: Correcta, 1, 3 y 4: Incorrectas
D. 2 y 4: Correctas, 1 y 3: Incorrectas

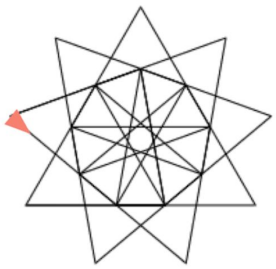
5.2) (0,4 puntos) Supongamos el siguiente código Scheme que usa la librería de gráficos de tortuga:

```
(define (figura a b)
  (if (> a 0)
      (begin
        (draw 100)
        (turn 120)
        (draw 100)
        (turn 120)
        (draw 100)
        (turn 120)
        (turn b)
        (figura (- a 1) b))))

(figura 10 36)
```

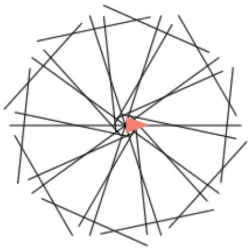
¿Cuál de las siguientes figuras se genera? Redondea la letra bajo la figura. La correcta es: d)



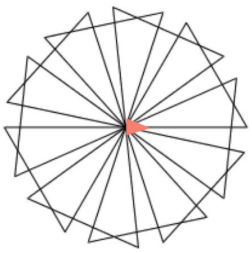
a)



b)



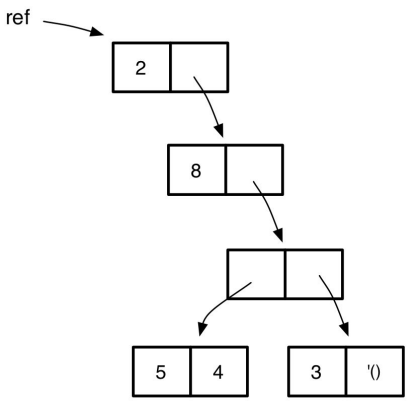
c)



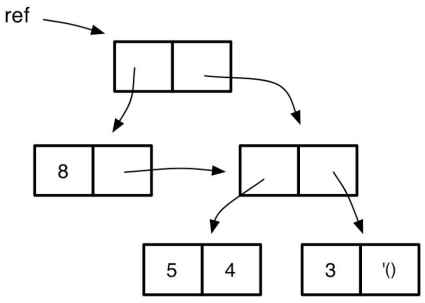
d)

5.3) (0,4 puntos) Supongamos el siguiente código que realiza mutación en la siguiente estructura definida por la variable ref:

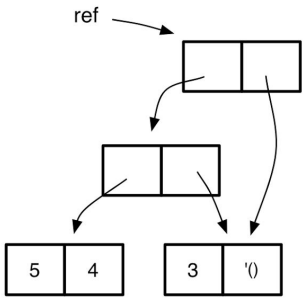
```
(define (foo ref)
  (if (null? (cddr ref))
      ref
      (foo (cdr ref))))

(define (prueba! lista)
  (let ((aux (foo lista)))
    (set-cdr! lista (cdr aux))
    (set-car! lista aux)))

(prueba! ref)
```



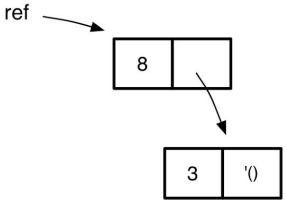
¿Cual de los siguientes diagramas representan el estado correcto de la estructura después de llamar a la anterior sentencia (prueba! ref)? Redondea la letra bajo la figura. La correcta es: b)



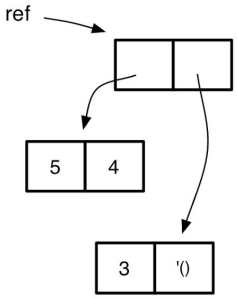
a)



b)



c)



d)

5.4) (0,4 puntos) Supongamos el siguiente código Scheme

```
(define num 0)

(define (inc)
  (set! num (+ num 1))
  num)

(define (make-contador i)
  (let ((num i))
    inc))

(define f (make-contador 10))
(define g (make-contador 20))
```

(f) ⇒ ?
(g) ⇒ ?

Indica qué devuelven las últimas dos sentencias:

A. Devuelven 11 y 21
B. Devuelven 1 y 2
C. Devuelven 11 y 12
D. Se produce un error

5.5) (0,4 puntos) Supongamos la siguiente función

```
(define (lista->str lista str)
   (if (null? lista)
      str
      (string-append (car lista) (lista->str (cdr lista) ""))))
```

Y supongamos la siguiente llamada a la función:

```
(lista->str '("hola" "como" "estas") "")
```

¿Cuál de las siguientes afirmaciones es correcta?:

A. La función usa recursión por la cola y la llamada devuelve "holacomoestas"
B. La función no usa recursión por la cola y la llamada devuelve la cadena vacía ("")
C. La función no usa recursión por la cola y la llamada devuelve "holacomoestas"
D. La función usa recursión por la cola y la llamada devuelve la cadena vacía ("")

5.6) (0,5 puntos) Supongamos el siguiente código

```
(define (make-contador i)
  (define num i)

  (define (inc-x x)
    (if (equal? x 'reset)
        (begin
          (set! num 0)
          num)
        (begin
          (set! num (+ num x))
          num)))

  (define (dec)
     (set! num (- num 1))
     dec)

  (define (dispatch mens)
    (cond
      ((equal? mens 'inc-x) inc-x)
      ((equal? mens 'dec) dec)
      (else 'error)))

  dispatch)

(define f (make-contador 1))
```

¿Cuál de las siguientes expresiones devolvería 0? :

A. ((f) 'dec)
B. ((f -1) 'inc-x)
C. ((f 'reset))
D. ((f 'inc-x) 'reset)