Soluciones
Lenguajes y Paradigmas de Programación
Curso 2004-2005
Examen de la Convocatoria de Septiembre

Pregunta 1 (10 puntos)
A) **(5 puntos)** Define una función `(cumplen-todos pred list)` que devuelva `#t` o `#f` dependiendo de si todos los elementos de la lista `list` cumplen el predicado `pred`. La lista `list` puede ser un pseudo-arbol y contener otras listas. Por ejemplo:

```
(cumplen-todos par? '(2 4 (6 8) 10)) -> #t
(cumplen-todos impar? '(1 3 5 6)) -> #f
(cumplen-todos impar? '()) -> #t
```

B) **(2 puntos)** La función que has definido en el apartado anterior ¿es iterativa o recursiva? ¿Por qué?

C) **(3 puntos)** Supongamos que uno de los elementos de la lista no cumple el predicado ¿se sigue procesando la lista en tu solución? En el caso de que no, perfecto, **ya puedes sumar los 3 puntos de este apartado**. En el caso de que se siga procesando la lista hasta el final, modifica la función para que no lo haga y devuelva `#f` justo en ese instante.

Solución:

```
;; version recursiva
(define (cumplen-todos? pred lista)
   (cond
      ((null? lista) #t)
      ((list? (car lista)) (and (cumplen-todos? pred (car lista))
                                (cumplen-todos? pred (cdr lista))))
      (else (and (pred (car lista))
                 (cumplen-todos? pred (cdr lista))))))

;; version iterativa
(define (cumplen-todos? pred lista)
   (cond
      ((null? lista) #t)
      ((list? (car lista))
            (if (cumplen-todos? pred (car lista))
                  (cumplen-todos? pred (cdr lista))
                  #f))
      (else (if (pred (car lista))
                  (cumplen-todos? pred (cdr lista))
                  #f))))
```

Pregunta 2 (10 puntos)
Vamos a crear un TAD para las horas del día (horas y minutos). Usaremos la representación de 24 horas, en que 3:00 PM es la hora 15:00. El constructor para las horas del día es `make-time`. Toma dos argumentos: las horas y los minutos. 4:12 debería ser `(make-time 4 12)` También queremos los siguientes tres operadores:

`(hour t)` → devuelve la parte de hora de un time dado
`(minute t)` → devuelve la parte de minutos de un time dado
`(hour? t)` → devuelve #t si el time t es una hora exacta (como 6:00); devuelve #f en cualquier otro caso

Ejemplo:

```
> (define t (make-time 4 12))
> (minute t)
12
> (hour t)
4
> (hour? t)
#f
```

Implementa el constructor `make-time` y los operadores `hour`, `minute` y `hour?` usando la técnica del paso de mensajes, en la que el objeto devuelto por el constructor es una función que admite mensajes.

Solución:

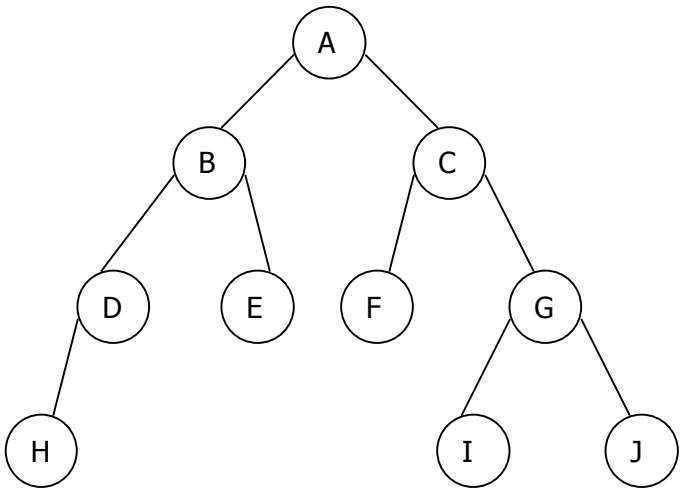
```
(define (make-time hora minuto)
   (lambda (mens)
      (cond
         ((equal? mens 'minute) minuto)
        ((equal? mens 'hour) hora)
        ((equal? mens 'hour?) (= minuto 0))
        (else "mensaje incorrecto"))))

(define (minute t)
   (t 'minute))

(define (hour t)
   (t 'hour))

(define (hour? t)
   (t 'hour?))
```

Pregunta 3 (10 puntos)
El procedimiento `(prof_hojas tree)` devuelve una lista plana con la profundidad de cada hoja de un árbol binario. Ejemplo:



`(prof_hojas tree)` devolverá (3 2 2 3 3)

Rellena los huecos para que `prof_hojas` funcione correctamente:

```
(define (prof_hojas tree)
      (define (help tree depth)
      (cond ((null? tree) _____________________________)
      ((leaf? tree) _____________________________________)
      (else ( ___________________________________________
               ___________________________________________))))
      (help tree 0))
```

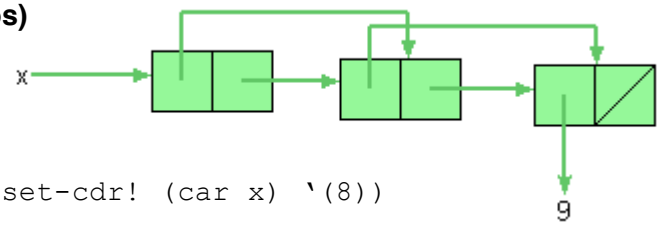
Solución:

```
(define (prof_hojas tree)
      (define (help tree depth)
            (cond ((null? tree) '())
               ((hoja? tree) (list depth))
               (else (append (help (hijo-izq tree) (1+ depth))
                             (help (hijo-der tree) (1+ depth))))))
    (help tree 0))
```

Pregunta 4 (10 puntos)
Para cada apartado:

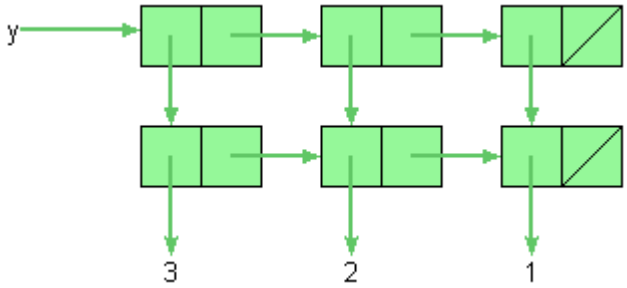
- Escribe una expresión Scheme que construya el diagrama caja y puntero de la figura.
- Dibuja el diagrama caja y puntero resultante de la mutación indicada.

A) **(5 puntos)**



Mutación: `(set-cdr! (car x) '(8))`

B) **(5 puntos)**

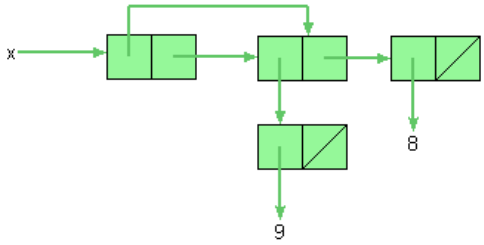


Mutación: `(set-car! (cdr (cadr y)) 4)`

Solución:

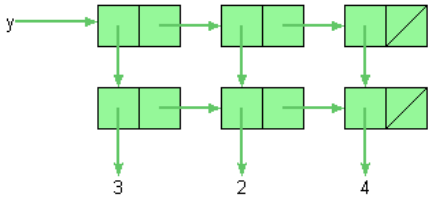
```
A)  (define x
        (let ((a '(9)))
           (let ((b (cons a a)))
              (cons b b))))
```

Después de la mutación:

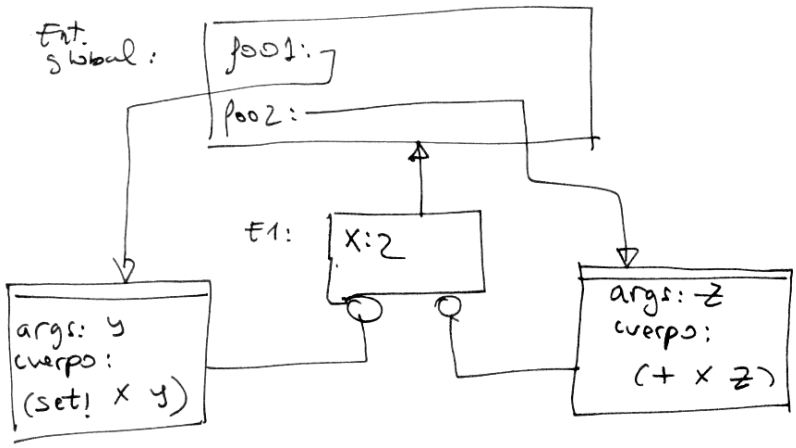



```
B) (define y
        (let ((z (list 3 2 1)))
           (list z (cdr z) (cddr z))))
```

Después de la mutación:



Pregunta 5 (10 puntos)
Supongamos el entorno definido por la siguiente figura



A) **(5 puntos)** Completa el siguiente programa para que genere el entorno anterior.

```
(define c  (_________________)
(define foo1 (______))
(define foo2 (______))
```

Nota: La variable auxiliar `c` no aparece en la figura.

B) **(4 puntos)** Explica paso a paso cómo se han evaluado las expresiones del programa que acabas de completar para generar el entorno de la figura, según el modelo de evaluación de Scheme basado en entornos.

C) **(1 punto)** Escribe un ejemplo de invocación de `foo1` y `foo2`, indicando el resultado devuelto.

Solución:

```
(define c
  (let ((x 2))
    (cons (lambda (y) (set! x y))
          (lambda (y) (+ x y)))))
(define foo1 (car c))
(define foo2 (cdr c))
```

La primera expresión se evalua en el entorno global. La forma especial “let” crea un entorno en el que la variable “x” se liga al valor “2”. Se trata del entorno E1. Este entorno es hijo del entorno global. En ese entorno se evalua la expresión “(cons (lambda …) (lambda ..))”. Cada una de las llamadas a “lambda” crean una función asociada al entorno E1. La llamada a “cons” crea una pareja que tiene como parte izquierda la primera función (el resultado de la evaluación de “(lambda (y) (set! x y))” y como parte derecha la segunda función (el resultado de evaluar “(lambda (y) (+ x y))”. Por último, la pareja resultante de la llamada a “cons” se liga con la variable “c” en el entorno global.

La segunda expresión crea una variable “foo1” en el entorno global que se liga con la parte izquierda de la pareja “c”. Esto es, “foo1” queda ligada a la función generada por la primera llamada a “lambda”.
Por último, la tercera expresión liga a “foo2” en el entorno global con la función creada por la segunda llamada a “lambda”.

```
(foo1 3) -> ok
(foo2 5) -> 8
```