Lenguajes y Paradigmas de Programación
Curso 2006-2007
Examen de la Convocatoria de Septiembre

Normas importantes

- La puntuación total del examen es de 60 puntos. Para obtener la nota final (en escala 0-10) se suman los puntos de prácticas y se divide por 6.
- Para sumar los puntos de las prácticas **es necesario obtener un mínimo de 24 puntos en este examen**.
- Se debe contestar cada pregunta **en una hoja distinta**. No olvides poner el nombre en todas las hojas.
- La duración del examen es de 3 horas.
- Las notas (y la fecha de revisión) estarán disponibles en la web de la asignatura el próximo día 18 de Septiembre.

Pregunta 1 (10 puntos)

a) (5 puntos) Vamos a crear un sistema para manipular polinomios:

$p1(x) = x^3 + 3x + 2$
$p2(x) = 3x^2 + 2x + 5$

Representaremos estos polinomios por sus coeficientes, almacenándolos en orden incremental de su potencia. Las expresiones anteriores se representarían como:

```
(2 3 0 1)
(5 2 3)
```

Para sumar dos polinomios, necesitamos sumar sus coeficientes:

$p1(x)+p2(x)=(x^3 + 3x + 2)+(3x^2 + 2x + 5)=(x^3 + 3x^2 + 5x + 7)$

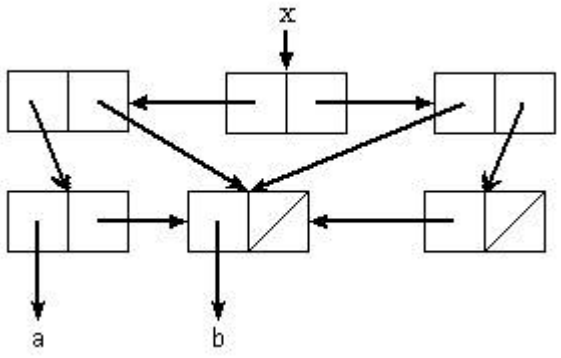
En nuestra representación obtendríamos `(7 5 3 1)`.

Implementa el procedimiento suma-polinomios que tome dos polinomio (listas de coeficientes) como argumento y devuelva un nuevo polinomio (lista de coeficientes) resultado de la suma de ambos.

b) (5 puntos) Dado el siguiente diagrama *box & pointer*:

b.1) Escribe las instrucciones de Scheme que lo generan.

b.2) Dibuja el diagrama *box & pointer* resultante después de evaluar la mutación: `(set-car! (cddr x) (caar x))`



Pregunta 2 (10 puntos)

a) (5 puntos) Implementa la función `(split-k lista n)` que divide una lista original en *k* sublistas. Las primeras *k-1* sublistas deben tener exactamente *n* elementos escogidos secuencialmente de la lista original. La última sublista tendrá los elementos sobrantes (pueden ser también *n* si el número de elementos de la lista original es módulo *n*). Debes hacer la implementación **sin utilizar mutadores**. Ejemplo:

```
(define lista '(1 2 3 4 5 6 7 8 9 10))
(split-k lista 4)-> '((1 2 3 4)(5 6 7 8) (9 10))
(split-k lista 3)-> '((1 2 3)(4 5 6)(7 8 9) (10))
```

b) (5 puntos) Implementa la función `(split-k! lista n)` que haga lo mismo que `split-k` pero **utilizando mutadores**. En esta versión no se debe llamar a `cons` más de *k* veces.

Pregunta 3 (10 puntos)

Queremos construir un tipo de datos `matrix` que nos permita manipular matrices representadas como una lista de listas, por ejemplo:

```
((10 11 12 13 14)
 (15 16 17 18 19)
 (20 21 22 23 24))
```

representaría la matriz:

```
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

a) (2 puntos) Define la barrera de abstracción del tipo de dato matriz e implementa 2 funciones de esta barrera de abstracción.

b) (4 puntos) Escribe un procedimiento`(make-matrix rows cols start)` que construya una matriz de `rows` filas por `cols` columnas, empezando en `start`, como indica el ejemplo:

`(make-matrix 3 5 10)` devolverá:
```
  ((10 11 12 13 14)(15 16 17 18 19)(20 21 22 23 24))
```

c) (4 puntos) Escribe un procedimiento `(transpuesta m)` que tome una matriz como argumento y devuelva su matriz transpuesta. Ejemplo:

```
(define m (make-matrix 3 5 10))
(transpuesta m) => ((10 15 20)(11 16 21)(12 17 22)(13 18 23)(14
19 24))
```

Pregunta 4 (10 puntos)

Supongamos que estamos implementando una calculadora con la que podemos lanzar operaciones (factorial, cuadrado, etc.) sobre un único número. A la calculadora le pasamos un símbolo que representa el nombre de una función a aplicar y un número al que aplicar esa función. Una posible implementación de este problema sería la siguiente:

```
(define (factorial n)
   (if (= n 0) 1
       (* n (factorial (- n 1)))))

(define (cuadrado n)
    (* n n))

(define (calculadora funcion n)
  (cond
    ((equal? funcion 'factorial) (factorial n))
    ((equal? funcion 'cuadrado) (cuadrado n))
    (else (error "funcion desconocida"))))

(calculadora ‘factorial 3) -> 6
(calculadora ‘doble 3) -> Error funcion desconocida
```

a) El problema fundamental de esta implementación es que no es posible añadir nuevas funciones a la calculadora sin modificar el procedimiento `calculadora`. Si queremos añadir una función que, por ejemplo, devuelva el doble de un número habría que definir esa función y modificar el condicional que implementa `calculadora`.

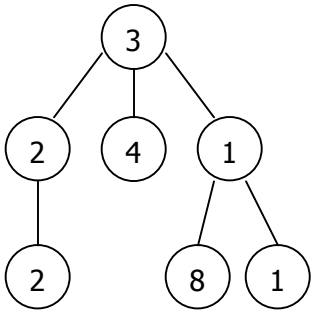
> (4 puntos) Cambia la implementación del procedimiento calculadora para que pueda trabajar con un número indeterminado de funciones. Puedes utilizar estructuras de datos adicionales. Explica qué habría que hacer para añadir nuevas funciones a la calculadora.
>
> (2 puntos) Define e implementa un procedimiento `añade-funcion` para añadir nuevas funciones a la calculadora. ¿Qué parámetros tendría? ¿Cómo se implementaría?

b) Otro problema de la implementación de calculadora es que únicamente se puede calcular con un número.

> (4 puntos) Modifica el procedimiento calculadora para que admita más de un número. Por ejemplo,
>
> ```
> (calculadora 'suma 2 3 4) -> 9
> (calculadora 'media 2 3 4) -> 3
> ```

Pregunta 5 (10 puntos)

Escribe un procedimiento `sumaniveltree` que tome un árbol genérico y un número que indique un nivel como argumentos, y devuelva la suma de los datos de ese nivel del árbol. Consideramos que la raíz tiene nivel 0.




```
(sumaniveltree 0 tree)
3
(sumaniveltree 1 tree)
7
(sumaniveltree 2 tree)
20
```

Pregunta 6 (10 puntos)

Supongamos las siguientes expresiones en Scheme:

```
(define x 3)
(define z 10)
(define g
   (let ((z 8)
         (x (+ z 3)))
      (lambda (z)
         (set! x (+ x z))
         x)))
(g 5)
```

a) (6 puntos) Dibuja y explica el diagrama de entornos creado al ejecutar las expresiones.
b) (2 puntos) ¿Qué valor devolverá la última expresión?
c) (2 puntos) ¿Cómo modificarías una única línea del programa para que la misma llamada `(g 5)` devolviera 14?.