Ejercicios de los exámenes de LPP del curso 2014-15 (convocatorias de junio y julio)

Explica brevemente con un pequeño párrafo (alrededor de 10 palabras) la contribución que consideres más importante de las siguientes personas a la historia de la Informática:

- Herman Hollerith
- Alan Turing:
- John von Neumann:
- John McCarthy:
- John Backus

Ordena temporalmente los siguientes lenguajes de programación

- C
- Java
- Scheme
- Go
- Scala
- Python

Dibuja el diagrama caja y puntero (*Box & Pointer*) de la siguiente expresión y explica si genera una lista o no:

```
(cons (list (cons 1 2) 3)
      (cons (cons (list 1 2)
                  '())
            3))
```

Dibuja el diagrama caja y puntero (*Box & Pointer*) de la siguiente expresión y explica si genera una lista o no:

```
 (cons (cons (list 1 2 3)
             (cons 2
                   (cons 3 4)))
       '())
```

Escribe una expresión de Scheme correcta en la que se invoque a `f` y que devuelva 12.

```
(define (f x)
     (lambda (y)
             (* x y)))
```

Implementa una función recursiva `(triangulo n)` que devuelva una lista con n listas, cada una de las cuales debe tener n asteriscos. Puedes usar funciones auxiliares, pero todas las funciones que uses deben ser recursivas.
Ejemplo:

```
(triangulo 10) ⇒
((* * * * * * * * * *)
 (* * * * * * * * *)
 (* * * * * * * *)
 (* * * * * * *)
 (* * * * * *)
 (* * * * *)
 (* * * *)
 (* * *)
 (* *)
 (*))
```

Dada una lista con al menos dos números escribe una función recursiva `(mayor-pareja lista)` que devuelva la pareja de números consecutivos cuya suma es la mayor de todas las sumas de números consecutivos de la lista.

Ejemplo:

```
(mayor-pareja '(1 5 8 2 20))
⇒(2 . 20)
(mayor-pareja '(1 15 8 2 20))
⇒(15 . 8)
```

Escribe la función `(expande lista-parejas)` utilizando funciones de orden superior. La función recibe una lista de parejas que contienen un dato y un número y devuelve una lista donde se han “expandido” las parejas, repitiendo tantos elementos como el número que indica cada pareja. Debes escribir también la implementación de cualquier función auxiliar que consideres necesaria.

Ejemplo:

```
(expande (list (cons 'a 4) (cons "hola" 2) (cons #f 3)))
⇒(a a a a "hola" "hola" #f #f #f)
```

Implementa las funciones `(suma-izq n pareja)` y `(suma-der n pareja)` definidas de la siguiente forma:

- `(suma-izq n pareja)` : recibe un número y una pareja de números, y devuelve una nueva pareja en la que se ha sumado el número n a la parte izquierda de pareja.
- `(suma-der n pareja)` : recibe un número y una pareja de números, y devuelve una nueva pareja en la que se ha sumado el número n a la parte derecha de pareja.

Ejemplo:

```
(suma-izq 4 '(12 . 20)) ⇒(16 . 20)
(suma-der 4 '(12 . 20)) ⇒(12 . 24)
```

Implementa una función recursiva `(filtra-cadenas lista-cadenas lista-num)` que recibe una lista de cadenas y una lista de números enteros, y devuelve una lista de parejas, en donde cada pareja está formada por la cadena de la i-ésima posición de lista-cadenas y el número entero situado en la i-ésima posición de `lista-num`, siempre y cuando dicho número se corresponda con la longitud de esa cadena. Puedes utilizar la función predefinida `string-length`

Ejemplo:

```
(filtra-cadenas
   '("estamos" "en" "el" "examen" "de" "LPP") '(4 2 3 6 1 3))
⇒
(("en" . 2) ("examen" . 6) ("LPP" . 3))
```

Define la función `(aplica-func f lista-parejas)` que recibe una función `f` de dos argumentos y una lista de parejas, y devuelve una lista con los resultados de aplicar la función `f` con los dos argumentos que vienen dados en cada una de las parejas.

Ejemplo:

```
(aplica-func + (list (cons 2 3) (cons 7 9) (cons 10 2)))
⇒(5 16 12)
```

Implementa 2 versiones:

- Versión recursiva
- Utilizando la función de orden superior map

Usando la función anterior, define la función recursiva `(aplica-lista-func lista-func lista-parejas)` que recibe una lista de funciones de dos argumentos y una lista de parejas, y devuelve una lista con los resultados de aplicar cada función de `lista-func` con los argumentos dados en cada una de las parejas de lista-parejas.

Ejemplo:

```
(aplica-lista-func (list + * /) (list (cons 2 3) (cons 7 9)))
⇒(5  16  6  63   2/3   7/9)
```

Define las funciones recursivas `(make-saludos lista-de-cadenas)` y `(aplica-lista-funcs lista-de-funciones dato)`. La función `make-saludos` recibe una lista de cadenas (saludos) y devuelve una lista de funciones de un argumento que concatenan un saludo con el argumento. La función `aplica-lista-funcs` recibe una lista de funciones de un argumento y un dato y devuelve una lista con los resultados de aplicar cada función al dato.

Nota: para concatenar cadenas puedes usar la función `string-append`

Ejemplo:

```
(make-saludos '("Hola " "Adios " "Cómo estás "))
⇒(#<procedure> #<procedure> #<procedure>)

(aplica-lista-funcs (make-saludos '("Hola " "Adios " "Cómo estás "))
"Pepe")
⇒("Hola Pepe" "Adios Pepe" "Cómo estás Pepe")
```

Dados el siguiente código en Scheme ¿Se crea alguna clausura? ¿Cuál es el resultado? Explica qué variables se han usado para calcular ese resultado, qué tipo de variable son (locales, globales o capturadas), qué valor tienen y dónde se ha obtenido ese valor.

```
(define z 10)
(define x 5)
(define (prueba z)
    (lambda (x) (+ x z)))
(define g (let ((x 20)
                (z (+ x 5)))
            (prueba x))
(g 4)
```

Dados el siguiente código en Scheme ¿Se crea alguna clausura? ¿Cuál es el resultado? Explica qué variables se han usado para calcular ese resultado, qué tipo de variable son (locales, globales o capturadas), qué valor tienen y dónde se ha obtenido ese valor.

```
(define z 3)
(define x 5)
(define (prueba z)
    (+ x z))
(define (g z)
     (lambda (x)
         (prueba (+ x z))))
(define f (g (+ x z)))
(f 10)
```