Resolución TP5 – Funciones en LISP

Introducción

En este práctico se profundiza el trabajo con funciones en LISP. La consigna propone utilizar principalmente **COND**, **DEFUN** y **MAPCAR**. Por lo tanto, las resoluciones se orientan a practicar esas herramientas:

DEFUN permite definir funciones propias.
COND permite evaluar varias condiciones, funcionando como una estructura de selección múltiple.
MAPCAR permite aplicar una función a cada elemento de una lista y construir una nueva lista con los resultados.

En algunos ejercicios también se utilizan funciones auxiliares necesarias para resolver operaciones más complejas, como sumar una lista, calcular factoriales, verificar primos, recorrer listas o construir nuevas listas. La idea central es resolver cada problema de manera sistemática: identificar el caso base, definir la condición, resolver el caso recursivo y observar qué valor se va generando en cada paso.

Ejercicio 1

Consigna: Asigne a tres variables una lista distinta a cada una con tres variables numéricas.

```
(setf lista1 '(1 2 3))
(setf lista2 '(2 2 2))
(setf lista3 '(3 2 1))
```

Explicación y lógica:

Se crean tres variables y en cada una se almacena una lista de tres números. La apóstrofe se usa para indicar que cada lista debe tomarse literalmente como dato y no como una instrucción a ejecutar.

Ejercicio 2

Consigna: Use COND para que: a) sume los elementos de las tres listas si sus sumas son iguales; b) reste la suma de la primera lista menos la suma de la tercera si solamente la primera suma es igual a la segunda; c) multiplique el primer elemento de las tres listas en caso contrario.

```
(defun suma-lista (l)
  (+ (first l) (first (rest l)) (first (rest (rest l)))))

(cond
  ((and (= (suma-lista lista1) (suma-lista lista2))
        (= (suma-lista lista2) (suma-lista lista3)))
   (+ (suma-lista lista1) (suma-lista lista2) (suma-lista lista3)))

  ((and (= (suma-lista lista1) (suma-lista lista2))
        (/= (suma-lista lista1) (suma-lista lista3)))
   (- (suma-lista lista1) (suma-lista lista3)))

  (t
   (* (first lista1) (first lista2) (first lista3))))
```

Explicación y lógica:

Primero se define una función auxiliar llamada SUMA-LISTA para sumar los tres elementos de una lista. Luego COND evalúa los casos en orden. Si las tres sumas son iguales, se suman las tres sumas. Si solamente la primera y la segunda son iguales, se resta la suma de la primera menos la tercera. Si no se cumple ningún caso anterior, se ejecuta el caso T, que funciona como “caso contrario”.

Lista	Valor	Suma
lista1	(1 2 3)	6
lista2	(2 2 2)	6
lista3	(3 2 1)	6
Resultado	se cumple el primer caso	18

Ejercicio 3

Consigna: Implemente la función FACTORIAL en forma recursiva.

```
(defun factorial (n)
  (cond
    ((= n 0) 1)
    (t (* n (factorial (- n 1))))))
```

Explicación y lógica:

El factorial de un número se calcula multiplicando ese número por todos los enteros positivos anteriores. La función es recursiva porque se llama a sí misma. El caso base es n = 0, donde el resultado es 1. El caso recursivo multiplica n por el factorial de n - 1.

Llamada	Desarrollo
(factorial 4)	4 * factorial(3)
(factorial 3)	3 * factorial(2)
(factorial 2)	2 * factorial(1)
(factorial 1)	1 * factorial(0)
(factorial 0)	1
Resultado	24

Ejercicio 4

Consigna: Implemente la función SUMATORIA de una lista en forma recursiva.

```
(defun sumatoria (lista)
  (cond
    ((null lista) 0)
    (t (+ (first lista) (sumatoria (rest lista))))))
```

Explicación y lógica:

La función suma todos los elementos de una lista. El caso base ocurre cuando la lista está vacía; en ese caso devuelve 0. En el caso recursivo se suma el primer elemento con la sumatoria del resto de la lista. Así la lista se va reduciendo hasta quedar vacía.

Llamada	Qué hace
(sumatoria '(4 5 6))	4 + sumatoria(5 6)
(sumatoria '(5 6))	5 + sumatoria(6)
(sumatoria '(6))	6 + sumatoria()
(sumatoria '())	0
Resultado	15

Ejercicio 5

Consigna: Implemente SUMATORIA recibiendo la lista a analizar y los límites en los cuales se analizará la sumatoria.

```
(defun elemento-en (lista pos)
  (cond
    ((= pos 1) (first lista))
    (t (elemento-en (rest lista) (- pos 1)))))

(defun sumatoria-limites (lista desde hasta)
  (cond
    ((> desde hasta) 0)
    (t (+ (elemento-en lista desde)
          (sumatoria-limites lista (+ desde 1) hasta)))) )
```

Explicación y lógica:

Se define primero una función auxiliar ELEMENTO-EN, que obtiene el elemento ubicado en una posición determinada. Luego SUMATORIA-LIMITES suma solamente los elementos cuyas posiciones están entre DESDE y HASTA. El caso base ocurre cuando DESDE supera a HASTA. Mientras eso no ocurra, se toma el elemento correspondiente y se avanza a la siguiente posición.

Ejemplo	Resultado
(sumatoria-limites '(10 20 30 40 50) 2 4)	20 + 30 + 40 = 90

Ejercicio 6

Consigna: Realice una función, sin usar MAPCAR, que reciba una lista y devuelva otra lista indicando si los elementos son números o no.

```
(defun numeros-sin-mapcar (lista)
  (cond
    ((null lista) nil)
    ((numberp (first lista))
     (cons 'numero (numeros-sin-mapcar (rest lista))))
    (t
     (cons 'no-numero (numeros-sin-mapcar (rest lista))))))
```

Explicación y lógica:

La función recorre la lista elemento por elemento usando recursividad. Si la lista está vacía, devuelve NIL. Si el primer elemento es numérico, agrega el símbolo NUMERO a la nueva lista. Si no es numérico, agrega NO-NUMERO. Luego continúa analizando el resto de la lista.

Entrada	Salida

(numeros-sin-mapcar '(1 a 3 b))	(NUMERO NO-NUMERO NUMERO NO-NUMERO)

Ejercicio 7

Consigna: Realice lo anterior usando MAPCAR.

```
(defun numeros-con-mapcar (lista)
  (mapcar
    (lambda (x)
      (cond
        ((numberp x) 'numero)
        (t 'no-numero)))
    lista))
```

Explicación y lógica:

MAPCAR aplica una función a cada elemento de una lista. En este caso, la función anónima LAMBDA recibe cada elemento como X. Si X es número, devuelve NUMERO; si no, devuelve NO-NUMERO. El resultado final es una nueva lista con una clasificación por cada elemento original.

Elemento analizado	Resultado
1	NUMERO
A	NO-NUMERO
3	NUMERO
B	NO-NUMERO

Ejercicio 8

Consigna: Realice una función que multiplique por 2 y reste 1 a todos los elementos de una lista usando MAPCAR.

```
(defun doble-menos-uno (lista)
  (mapcar
    (lambda (x)
      (- (* x 2) 1))
    lista))
```

Explicación y lógica:

La función aplica a cada elemento la operación: x * 2 - 1. MAPCAR toma cada número de la lista, lo pasa a la función LAMBDA y arma una nueva lista con los resultados.

Entrada	Proceso	Salida
(1 2 3 4)	2x - 1	(1 3 5 7)

Ejercicio 9

Consigna: Realizar una función que reciba un número y devuelva una lista con los números que lo forman en binario.

```
(defun binario (n)
  (cond
    ((< n 2) (list n))
    (t (append (binario (floor n 2))
               (list (mod n 2))))))
```

Explicación y lógica:

Para obtener el número binario se divide sucesivamente por 2 y se toman los restos. La función es recursiva: primero calcula el binario del cociente y luego agrega el resto. Con APPEND se unen las partes para formar la lista final.

Número	Cociente	Resto
10 / 2	5	0
5 / 2	2	1
2 / 2	1	0
1	caso base	1
Resultado		(1 0 1 0)

Ejercicio 10

Consigna: Realice una función que calcule si un número es primo o no.

```
(defun divisorp (n d)
  (= (mod n d) 0))

(defun buscar-divisor (n d)
  (cond
    ((= d n) nil)
    ((divisorp n d) t)
    (t (buscar-divisor n (+ d 1)))))

(defun primo (n)
  (cond
    ((= n 1) t)
    ((< n 1) nil)
    (t (not (buscar-divisor n 2)))))
```

Explicación y lógica:

Un número primo se considera aquel que no tiene divisores entre 2 y el número anterior. La función BUSCAR-DIVISOR intenta encontrar algún divisor. Si encuentra uno, devuelve verdadero. La función PRIMO devuelve verdadero cuando no se encuentra ningún divisor. Se toma 1 como primo porque el enunciado posterior usa como ejemplo la lista (1 2 3 5 7).

Ejercicio 11

Consigna: Evalúe una lista indicando si sus elementos son números primos.

```
(defun evaluar-primos (lista)
  (mapcar
    (lambda (x)
      (cond
        ((primo x) 'primo)
        (t 'no-primo)))
    lista))
```

Explicación y lógica:

Se utiliza MAPCAR para aplicar la función PRIMO a cada elemento de la lista. Por cada número se devuelve el símbolo PRIMO o NO-PRIMO. El resultado es una lista de la misma longitud que la original, pero con la evaluación de cada elemento.

Entrada	Salida
(evaluar-primos '(1 2 4 5 6))	(PRIMO PRIMO NO-PRIMO PRIMO NO-PRIMO)

Ejercicio 12

Consigna: Realice una función que reciba un número y devuelva una lista con esa cantidad de números primos.

```
(defun dime-primos-aux (cantidad actual encontrados)
  (cond
    ((= cantidad 0) nil)
    ((primo actual)
     (cons actual
           (dime-primos-aux (- cantidad 1) (+ actual 1) (+ encontrados 1))))
    (t
     (dime-primos-aux cantidad (+ actual 1) encontrados))))

(defun dime-primos (cantidad)
  (dime-primos-aux cantidad 1 0))
```

Explicación y lógica:

La función comienza a buscar números primos desde 1. Si el número actual es primo, lo agrega a la lista y disminuye la cantidad pendiente. Si no es primo, simplemente pasa al siguiente número. El proceso termina cuando ya se obtuvo la cantidad solicitada.

Llamada	Resultado
(dime-primos 5)	(1 2 3 5 7)

Ejercicio 13

Consigna: Sin usar LENGTH, realice una función que reciba una lista y devuelva la cantidad de elementos.

```
(defun cantidad-elementos (lista)
  (cond
    ((null lista) 0)
    (t (+ 1 (cantidad-elementos (rest lista))))))
```

Explicación y lógica:

La función cuenta los elementos recursivamente. Si la lista está vacía, devuelve 0. Si no está vacía, cuenta 1 por el primer elemento y sigue contando el resto de la lista.

Entrada	Proceso	Salida
(a b c d)	1 + 1 + 1 + 1 + 0	4

Ejercicio 14

Consigna: Implemente una función que reciba un número y devuelva una lista con esa cantidad de números de la serie Fibonacci.

```
(defun fibonacci-aux (cantidad a b)
  (cond
    ((= cantidad 0) nil)
    (t (cons a
             (fibonacci-aux (- cantidad 1) b (+ a b))))))

(defun fibonacci (cantidad)
  (fibonacci-aux cantidad 0 1))
```

Explicación y lógica:

La serie Fibonacci comienza con 0 y 1. Cada nuevo valor se obtiene sumando los dos anteriores. La función auxiliar mantiene dos valores: A y B. En cada paso agrega A a la lista y avanza con B y A+B.

Llamada	Salida
(fibonacci 10)	(0 1 1 2 3 5 8 13 21 34)

Ejercicio 15

Consigna: Implemente una función que reciba una lista formada por sublistas y los parámetros de búsqueda. Debe devolver el elemento solicitado y validar los datos.

```
(defun elemento-en (lista pos)
  (cond
    ((null lista) nil)
    ((= pos 1) (first lista))
    (t (elemento-en (rest lista) (- pos 1)))))

(defun todas-sublistas (lista)
  (cond
    ((null lista) t)
    ((listp (first lista)) (todas-sublistas (rest lista)))
    (t nil)))

(defun dame (lista pos-elemento pos-sublista)
  (cond
    ((not (todas-sublistas lista)) 'error-no-son-sublistas)
    ((or (< pos-elemento 1) (< pos-sublista 1)) 'error-posiciones-invalidas)
    ((null (elemento-en lista pos-sublista)) 'error-sublista-inexistente)
    ((null (elemento-en (elemento-en lista pos-sublista) pos-elemento))
     'error-elemento-inexistente)
    (t (elemento-en (elemento-en lista pos-sublista) pos-elemento))))
```

Explicación y lógica:

La función DAME primero valida que la lista esté formada por sublistas. Después verifica que las posiciones sean válidas. Luego busca la sublista indicada y dentro de ella el elemento solicitado. Para el ejemplo (dame '((1 2 3) (4 5 6) (7 8 9)) 3 2), primero toma la segunda sublista, es decir (4 5 6), y luego obtiene su tercer elemento: 6.

Paso	Resultado
Lista original	((1 2 3) (4 5 6) (7 8 9))
Segunda sublista	(4 5 6)
Tercer elemento	6

Ejercicio 16

Consigna: Realice una función que reciba como parámetro dos números y devuelva el máximo común divisor.

```
(defun mcd (a b)
  (cond
    ((= b 0) a)
    (t (mcd b (mod a b)))))
```

Explicación y lógica:

Se aplica el algoritmo de Euclides. El máximo común divisor de A y B se obtiene reemplazando A por B y B por el resto de dividir A entre B. Cuando B llega a 0, el valor de A es el máximo común divisor.

Llamada	Transformación
(mcd 48 18)	mcd 18 12
(mcd 18 12)	mcd 12 6
(mcd 12 6)	mcd 6 0
Resultado	6

Ejercicio 17

Consigna: Realice una función que reciba una lista y devuelva otra lista con todos los números no primos menores que 10.

```
(defun no-primos-menores-10 (lista)
  (cond
    ((null lista) nil)
    ((and (numberp (first lista))
          (< (first lista) 10)
          (not (primo (first lista))))
     (cons (first lista)
           (no-primos-menores-10 (rest lista))))
    (t
     (no-primos-menores-10 (rest lista)))))
```

Explicación y lógica:

La función recorre la lista elemento por elemento. Solo conserva aquellos elementos que cumplen tres condiciones: son números, son menores que 10 y no son primos. Los elementos que no cumplen esas condiciones se descartan.

Ejercicio 18

Consigna: Realice una función que reciba un número y devuelva una lista con sus factores primos.

```
(defun factores-primos-aux (n divisor)
  (cond
    ((= n 1) nil)
    ((= (mod n divisor) 0)
     (cons divisor
           (factores-primos-aux (/ n divisor) divisor)))
    (t
     (factores-primos-aux n (+ divisor 1)))))

(defun factores-primos (n)
  (factores-primos-aux n 2))
```

Explicación y lógica:

Se intenta dividir el número comenzando desde 2. Si el divisor divide exactamente al número, se agrega ese divisor a la lista y se continúa dividiendo el cociente. Si no divide, se prueba con el divisor siguiente. El proceso termina cuando el número llega a 1.

Ejemplo	Salida
(factores-primos 48)	(2 2 2 2 3)
(factores-primos 60)	(2 2 3 5)
(factores-primos 23)	(23)

Ejercicio 19

Consigna: Implemente una función que reciba una lista de números y devuelva cada número con la cantidad de veces que se repite.

```
(defun contar-iguales (x lista)
  (cond
    ((null lista) 0)
    ((= x (first lista)) (+ 1 (contar-iguales x (rest lista))))
    (t (contar-iguales x (rest lista)))))

(defun quitar-iguales (x lista)
  (cond
    ((null lista) nil)
    ((= x (first lista)) (quitar-iguales x (rest lista)))
    (t (cons (first lista) (quitar-iguales x (rest lista))))))

(defun contar-repetidos (lista)
  (cond
    ((null lista) nil)
    (t (cons (first lista)
             (cons (contar-iguales (first lista) lista)
                   (contar-repetidos
                     (quitar-iguales (first lista) lista)))))))
```

Explicación y lógica:

La función toma el primer número de la lista, cuenta cuántas veces aparece y lo agrega junto con su cantidad. Luego elimina todas las apariciones de ese número para no volver a contarlo. El proceso se repite con el resto de los números hasta que la lista queda vacía.

Entrada	Salida
(contar-repetidos '(2 2 2 2 3))	(2 4 3 1)
(contar-repetidos '(2 2 3 5))	(2 2 3 1 5 1)
