



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

---

FACULTAD DE CIENCIAS

LENGUAJES DE PROGRAMACIÓN

2025-1

## Semanal 7: Lenguajes de Programacion

INTEGRANTES:

García López Francisco Daniel - 320104321

Castillo Hernández Antonio - 320017438

Vázquez Reyes Jesús Elías - 320010549

FECHA DE ENTREGA:

15 de Octubre de 2024

PROFESOR:

Manuel Soto Romero

AYUDANTES:

Teo. Demian Alejandro Monterrubio Acosta

Lab. Erik Rangel Limón



1. Dada la siguiente expresión en **MiniLisp**

```
(let (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1))))))
  (sum 5))
```

- Ejecutarla y explicar el resultado.
- Modificarla usando el combinador de punto fijo Y, volver a ejecutarla y explicar el resultado.

*Ejecución de la expresión:*

```
(sum 5) -> (if0 5 0 (+ 5 (sum (- 5 1))))
-> (+ 5 (sum (- 5 1)))
-> (+ 5 (sum 4))
-> (+ 5 (if0 4 0 (+ 4 (sum (- 4 1)))))
-> (+ 5 (+ 4 (sum (- 4 1))))
-> (+ 5 (+ 4 (sum 3)))
-> (+ 5 (+ 4 (if0 3 0 (+ 3 (sum (- 3 1))))))
-> (+ 5 (+ 4 (+ 3 (sum (- 3 1)))))
-> (+ 5 (+ 4 (+ 3 (sum 2))))
-> (+ 5 (+ 4 (+ 3 (if0 2 0 (+ 2 (sum (- 2 1)))))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (sum (- 2 1))))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (sum 1)))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (if0 1 0 (+ 1 (sum (- 1 1))))))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum (- 1 1)))))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum 0))))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (if0 0 0 (+ 0 (sum (- 0 1))))))))
-> (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0)))))
-> (+ 5 (+ 4 (+ 3 (+ 2 1))))
-> (+ 5 (+ 4 (+ 3 3)))
-> (+ 5 (+ 4 6))
-> (+ 5 10)
-> 15.
```

Veamos que la expresión en MiniLisp dada, se define una función llamada *sum* la cual se manda a llamar recursivamente dentro del cuerpo de la lambda y recibe como argumento a una *n*. En dicho cuerpo también se hace uso del *if 0* que verifica si *n* es igual a 0. Si lo es, devuelve 0. De lo contrario, suma el valor de *n* al resultado de una llamada recursiva de *sum* con *n* − 1.

Entonces, al ejecutar esta función con el valor 5, la evaluación sigue una serie de llamadas recursivas, comenzando con *sum 5* el cual devuelve  $5 + \text{sum}(4)$ , y así sucesivamente hasta llegar a *sum 0*. Luego cuando se alcanza el caso base, se retorna 0 de tal forma que las llamadas recursivas regresan, cada valor de *n* se suma hasta que finalmente se obtiene el resultado. Al final obtenemos la suma de  $5 + 4 + 3 + 2 + 1 + 0$ , que da 15. Entonces podemos decir que realiza la suma de los números naturales de 5 hacia abajo, acumulando el resultado en cada paso de la recursión.

*Modificación utilizando Y combinator:*

El combinador de punto fijo Y permite obtener la recursión sin la necesidad de autoaplicar explícitamente. Este combinador se define como:

$$Y = \lambda f.(\lambda x.f (x x))(\lambda x.f (x x))$$

Donde es importante recordar que Ppra cualquier función  $g$ , se cumple que:

$$Y g = g (Y g)$$

Por lo cual la expresión en MiniLisp con el combinador de punto fijo  $Y$  es:

```
(let (Y (lambda (F) ((lambda (x) (F (lambda (v) ((x x) v))))
                    (lambda (x) (F (lambda (v) ((x x) v)))))))
  (sum (Y (lambda (sum) (lambda (n)
                        (if0 n 0 (+ n (sum (- n 1)))))))
    (sum 5)))
```

**Ejecución:**

```
(Y func) 5 -> (lambda f. lambda n. (if0 n 0 (+ n (f (- n 1))))) 5
-> (if0 5 0 (+ 5 ((Y func) 4)))
-> 5 + ((Y func) 4)
-> 5 + (if0 4 0 (+ 4 ((Y func) 3)))
-> 5 + 4 + ((Y func) 3)
-> 5 + 4 + (if0 3 0 (+ 3 ((Y func) 2)))
-> 5 + 4 + 3 + ((Y func) 2)
-> 5 + 4 + 3 + (if0 2 0 (+ 2 ((Y func) 1)))
-> 5 + 4 + 3 + 2 + ((Y func) 1)
-> 5 + 4 + 3 + 2 + (if0 1 0 (+ 1 ((Y func) 0)))
-> 5 + 4 + 3 + 2 + 1 + ((Y func) 0)
-> 5 + 4 + 3 + 2 + 1 + (if0 0 0 (+ 0 ((Y func) -1)))
-> 5 + 4 + 3 + 2 + 1 + 0
-> 15
```

Al utilizar el combinador de punto fijo  $Y$ , logramos definir la recursión dentro de una función sin hacer referencia directa a su nombre. Esto permite representar la recursión de forma anónima. El resultado obtenido es el mismo que en una función recursiva tradicional, sumando los números desde 5 hasta 0 para obtener 15.

2. Evaluar la siguiente expresión en **Racket**, explicar su resultado y dar la continuación asociada a evaluar usando la notación  $\lambda \uparrow$ .

```
> (define c #f)
> (+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
> (c 10)
```

Tenemos la continuación:  $\lambda \uparrow (v) (+ 1 (+ 2 (+ 3 (+ v 5))))$

Cuando se evalúa el código sin utilizar la continuación guardada en  $c$ , el resultado es  $1 + 2 + 3 + 4 + 5 = 15$ .

En cambio, cuando se llega a  $(c 10)$ , el programa regresa al punto en que se llamó a `let/cc`, pero con el valor 10 en lugar del valor 4. Así, tenemos la expresión:

$$1 + 2 + 3 + 10 + 5 = 21.$$

3. *Tercer Ejercicio del Semanal*

# Funciones

Listing 1: Ejercicio 3

```
-- Funcion que cuenta cuantas veces aparece un elemento en una lista
ocurrencias :: Eq a => a -> [a] -> Int
ocurrencias _ [] = 0
ocurrencias x (y:ys)
    | x == y      = 1 + ocurrencias x ys
    | otherwise   = ocurrencias x ys

-- Funcion recursiva para obtener las ocurrencias de cada elemento de la
-- segunda lista en la primera lista
ocurrenciasElementos :: Eq a => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos _ [] = []
ocurrenciasElementos xs (y:ys) = (y, ocurrencias y xs) : ocurrenciasElementos
    xs ys

-- Funcion auxiliar con acumulador para la version optimizada (recursion de
-- cola)
ocurrenciasElementosAux :: Eq a => [a] -> [a] -> [(a, Int)] -> [(a, Int)]
ocurrenciasElementosAux _ [] acc = reverse acc
ocurrenciasElementosAux xs (y:ys) acc = ocurrenciasElementosAux xs ys ((y,
    ocurrencias y xs) : acc)

-- Funcion optimizada con recursion de cola
ocurrenciasElementosCola :: Eq a => [a] -> [a] -> [(a, Int)]
ocurrenciasElementosCola xs ys = ocurrenciasElementosAux xs ys []
```

## Resultados de Ejecución

El siguiente es el resultado de la ejecución del programa para las listas [1,3,6,2,4,7,3,9,7] y [5,2,3]:

Resultado de ocurrenciasElementos:

[(5,0),(2,1),(3,2)]

Resultado de ocurrenciasElementosCola:

[(5,0),(2,1),(3,2)]