UNIVERSIDAD NACIONAL AUTÓNOMA DE
MÉXICO
FACULTAD DE CIENCIAS
LENGUAJES DE PROGRAMACIÓN

Semanal 07

Victor de Jesús Villalobos Ramírez - 313098675

Miguel Angel Vargas Campos - 423114223

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.

```
Sustitución [sum:=(lambda (n) (if0 n 0 (+ n (sum (- n 1)))))]
((lambda (n) (if0 n 0 (+ n (sum (- n 1)))))5)
Sustitución [n := 5]
(if0 5 0 (+ 5 (sum (- 5 1))))
(+ 5 (sum (- 5 1)))
(+ 5 (sum 4))
error
```

     La ejecución genera un error de variable libre ya que no tiene manera de seguir, por que no sabe la definición de sum y así poder auto-referenciarse.

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

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

```
Sustitución [sum:=(Y (lambda (sum) (lambda (n) (if0 n 0 (+ n (sum (- n
(Y (lambda (sum) (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))5))
Sustitución [n := 5]
(Y (lambda (sum) (if0 5 0 (+ 5 (sum (- 5 1))))))
(Y (lambda (sum) (+ 5(sum (- 5 1)))))
(y (lambda (sum) (+ 5(sum 4))))
```

```
Sustitución [sum:=(Y (lambda (sum) (lambda (n) (if0 n 0 (+ n (sum (- n
(Y (lambda (sum) (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))4))
Sustitución [n := 4]
```

```
(Y (lambda (sum) (+ 5(if0 4 0 (+ 4 (sum (- 4 1)))))))
(Y (lambda (sum) (+ 5(+ 4 (sum (- 4 1))))))
(lambda (sum) (+ 5(+ 4 (sum 3)))Y)
...
...
...

(+5 (+4 (+3 (+2 (+1 0)))))
15
```

El resultado final de la ejecución es 15 ya que gracias al usar el combinador de punto fijo Y podemos hacer auto-referencia a la función completa de sum y así evitar los problemas de variables libres que se podrían presentar, como en la ejecución anterior.

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)
```

Evaluación:

Figure 1: Ejecución del código en *Racket*.

Explicación:

- `(define c #f)`

  Definimos la variable c como $\#f$.

- `(+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))`

  Se ejecuta:

  `(+ 1 (+ 2 (+ 3 (+ 4 5))))`

Para obtener 15 y guardamos en c la continuación asociada al punto donde está el 4.

- `(c 10)`

  Ejecutamos la continuación que habíamos almacenado en c pasando 10 como parámetro real y obtenemos 21 como resultado.

Continuación asociada:

$\lambda\uparrow v.(+\ 1\ (+\ 2\ (+\ 3\ (+\ v\ 5))))$

3. Realizar los siguientes ejercicios en **Haskell**:

   Definir la función recursiva *ocurrenciasElementos* que toma como argumentos dos listas y devuelve una lista de parejas, en donde cada pareja contiene en su parte izquierda un elemento de la segunda lista y en su parte derecha el número de veces que aparece dicho elemento en la primera lista. Por ejemplo:

```
> ocurrenciasElementos [1,3,6,2,4,7,3,9,7] [5,2,3]
[(5,0),(2,1),(3,2)]
```

   Funcion en **Haskell**:

```
ocurrenciasElementos :: Eq a => [a] -> [a] -> [(a,Integer)]
ocurrenciasElementos _ [] = []
ocurrenciasElementos list (y:ys)
    = (y, count list y) : ocurrenciasElementos list ys

count :: Eq a => [a] -> a -> Integer
count [] _ = 0
count (x:xs) elem
    | x == elem = 1 + count xs elem
    | otherwise = count xs elem
```

   - Mostrar los registros de activación generados por la función definida en el ejercicio anterior con la llamada *ocurrenciasElementos [1,2,3] [1,2]*.

```
> ocurrenciasElementos [1,2,3] [1,2]
> (1, count [1,2,3] 1) : ocurrenciasElementos [1,2,3] [2]
> (1, 1 + count [2,3] 1) : ocurrenciasElementos [1,2,3] [2]
> (1, 1 + count [3] 1) : ocurrenciasElementos [1,2,3] [2]
> (1, 1 + count [] 1) : ocurrenciasElementos [1,2,3] [2]
> (1, 1 + 0) : ocurrenciasElementos [1,2,3] [2]
> (1, 1) : ocurrenciasElementos [1,2,3] [2]
> (1, 1) : (2, count [1,2,3] 2) : ocurrenciasElementos [1,2,3] []
> (1, 1) : (2, count [2,3] 2) : ocurrenciasElementos [1,2,3] []
> (1, 1) : (2, 1 + count [3] 2) : ocurrenciasElementos [1,2,3] []
> (1, 1) : (2, 1 + count [] 2) : ocurrenciasElementos [1,2,3] []
```

```
> (1, 1) : (2, 1 + 0) : ocurrenciasElementos [1,2,3] []
> (1, 1) : (2, 1) : ocurrenciasElementos [1,2,3] []
> (1, 1) : (2, 1) : []
```

- Optimizar la función definida usando recursión de cola. Deben transformar todas las funciones auxiliares que utilicen.

```
ocurrenciasElementos :: Eq a => [a] -> [a] -> [(a,Integer)]
ocurrenciasElementos list search = aux [] list search

aux :: Eq a => [(a,Integer)] -> [a] -> [a] -> [(a,Integer)]
aux acc _ [] = acc
aux acc list (y:ys)
    = aux ((y, count 0 list y) : acc) list ys

count :: Eq a => Integer -> [a] -> a -> Integer
count acc [] _ = acc
count acc (x:xs) elem
    | x == elem = count (acc + 1) xs elem
    | otherwise = count acc xs elem
```

- Mostrar los registros de activación generados por la versión de cola con la misma llamada.

```
> ocurrenciasElementos [1,2,3] [1,2]
> aux [] [1,2,3] [1,2]
> aux ((1, count 0 [1,2,3] 1) : []) [1,2,3] [2]
> aux ((1, count 1 [2,3] 1) : []) [1,2,3] [2]
> aux ((1, count 1 [3] 1) : []) [1,2,3] [2]
> aux ((1, count 1 [] 1) : []) [1,2,3] [2]
> aux ((1, 1) : []) [1,2,3] [2]
> aux ((2, count 0 [1,2,3] 2) : (1, 1) : []) [1,2,3] []
> aux ((2, count 0 [2,3] 2) : (1, 1) : []) [1,2,3] []
> aux ((2, count 1 [3] 2) : (1, 1) : []) [1,2,3] []
> aux ((2, count 1 [] 2) : (1, 1) : []) [1,2,3] []
> aux ((2, 1) : (1, 1) : []) [1,2,3] []
> (2, 1) : (1, 1) : []
```