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.
  Tenemos que

```
sum =_def lambda (f): lambda (n): if0 n then 0
                                          else (+ n (f(n-1)))
```

  Empezamos evaluando la expresión.
  Evaluamos (sum 5), al aplicar la definición anterior de sum como 5 no es cero entonces pasamos a la condición de else, por lo que tenemos:

  $sum(5) = 5 + sum(4)$
  $sum(4) = 4 + sum(3)$
  $sum(3) = 3 + sum(2)$
  $sum(2) = 2 + sum(1)$
  $sum(1) = 1 + sum(0)$

  Sumando los valores tenemos $5 + 4 + 3 + 2 + 1 + 0 = 15$

- Modificarla usando el combinador de punto fijo Y, volver a ejecutarla y explicar el resultado.
  Tomemos en cuenta que :

```
Y =_def lambda f. ( lambda x. f(xx))(lambda x. f(xx))

sum = lambda f. lambda n. if0 n 0 (+ n (f (- n 1)))
```

  Reescribimos la función sum utilizando Y:

```
(Y sum 5)
=_def ((lambda f.(lambda x.f(xx))(lambda x.f(xx)))sum)5
```

  Aplicamos Y a sum

```
=(lambda x.sum(xx))(lambda x.sum(xx))5
```

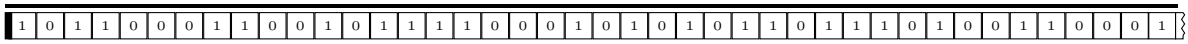
  Aplicamos la definición de sum

```
sum=lambda n.if0 n 0(+ n(f(n1)))

=(lambda n.if0 n 0(+n(Y sum(n1))))5

(lambda n: if0 n then 0 else (+n( Y sum)(n-1)))5
```


Figueroa Barrientos Andrea Valeria 31705346
Martínez Hidalgo Paola Mildred 319300217
Hazel Torres Nava 319158496



Ahora evaluamos la expresión:
$(Ysum(5)), 5 + (Ysum(4))$
$(Ysum(4)), 4 + (Ysum(3))$
$(Ysum(3)), 3 + (Ysum(2))$
$(Ysum(2)), 2 + (Ysum(1))$
$(Ysum(1)), 1 + (Ysum(0))$

Sumamos lo anterior
$5 + 4 + 3 + 2 + 1 + 0 = 15$

Con lo anterior podemos ver que en ambas evaluaciones da el mismo resultado.

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 :

```
> 15
> 21
```

Explicación :

```
> (define c #f)                  // Define la variable c como falso
> (+ 1 (+ 2 (+ 3                 // Sumas anidadas
   (+  (let/cc k (set! c k) 4)   // Aquí captura una continuación k, y regresa 4
   5))))                         // Fin de las sumas anidadas. Al final se
                                 // evalua como :
                                 //  >(+ 1 (+ 2 (+ 3 (+ 4 5)))) = 15
 // Notemos que el cuerpo de la continuación indica que k (la continuación)
 // se guarda en la variable c
```

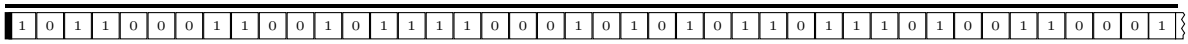
Veasé que la continuación asociada es :

$$(\lambda \uparrow v\ (+1(+2(+3(+v\ 5))))\ )((set!\ c\ k)4)$$

Continuación explicación :

```
> (c 10)    \\Como en la continuación k  se guardo en la
            \\ variable c, entonces se ejecuta la continuación con
            \\ argumento 10. Se evalua como :
            \\  >(+ 1 (+ 2 (+ 3 (+ 10 5)))) = 21
```


Figueroa Barrientos Andrea Valeria 31705346
Martínez Hidalgo Paola Mildred 319300217
Hazel Torres Nava 319158496



Ejercicio 3:

Solución recursiva

Consideremos la función `ocurrenciasElementosNoTail` que cuenta las ocurrencias de los elementos de una segunda lista en una primera lista. Examinemos los registros de activación generados por la llamada:

```
ocurrenciasElementosNoTail [1,2,3] [1,2]
```

Paso a paso

Primera llamada:

Argumentos:

```
lista = [1,2,3], x = 1, xs = [2]
```

La función evalúa:

```
length (filter (== 1) [1,2,3]) = 1
```

Registro de activación pendiente:

$[(1,1)]$: `ocurrenciasElementosNoTail [1,2,3] [2]`

Segunda llamada:

Argumentos:

```
lista = [1,2,3], x = 2, xs = []
```

La función evalúa:

```
length (filter (== 2) [1,2,3]) = 1
```

Registro de activación pendiente:

$[(2,1)]$: `ocurrenciasElementosNoTail [1,2,3] []`

Tercera llamada (caso base):

Argumentos:

```
lista = [1,2,3], elems = []
```

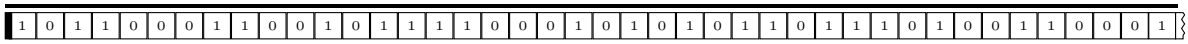
Como la lista está vacía, la función devuelve:

```
[]
```

Retornos:

- La tercera llamada retorna `[]` a la segunda llamada, que ahora puede devolver `[(2, 1)]`.
- La primera llamada recibe el resultado `[(2, 1)]` y puede devolver `[(1, 1), (2, 1)]`.

Figueroa Barrientos Andrea Valeria 31705346
Martínez Hidalgo Paola Mildred 319300217
Hazel Torres Nava 319158496



Estructura de la pila

Primera llamada:	$[(1,1)]$: `ocurrenciasElementosNoTail [1,2,3] [2]`
Segunda llamada:	$[(1,1)]$: $[(2,1)]$: `ocurrenciasElementosNoTail [1,2,3] []`
Caso base:	$[(1,1)]$: $[(2,1)]$: $[]$
Resultado final:	$[(1,1),(2,1)]$

Solución usando Tail recursion

Descripción

En la versión con recursión de cola, los resultados parciales se acumulan en el argumento `acc`, y la pila no se llena con cada llamada recursiva. Examinemos los registros de activación generados por la llamada:

```
ocurrenciasElementosTail [1,2,3] [1,2]
```

Paso a paso

Primera llamada:

Argumentos:

```
lista = [1,2,3], elems = [1,2], acc = []
```

La función evalúa:

```
length (filter (== 1) [1,2,3]) = 1
```

Actualiza el acumulador:

```
acc = [(1, 1)]
```

Registro de activación pendiente:

```
ocurrenciasAux [1,2,3] [2] [(1, 1)]
```

Segunda llamada:

Argumentos:

```
lista = [1,2,3], elems = [2], acc = [(1, 1)]
```

La función evalúa:

```
length (filter (== 2) [1,2,3]) = 1
```

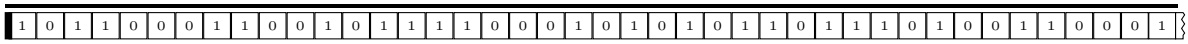
Actualiza el acumulador:

```
acc = [(2, 1), (1, 1)]
```

Registro de activación pendiente:

```
ocurrenciasAux [1,2,3] [] [(2, 1), (1, 1)]
```


Figueroa Barrientos Andrea Valeria 31705346
Martínez Hidalgo Paola Mildred 319300217
Hazel Torres Nava 319158496



Tercera llamada (caso base):

Argumentos:

```
lista = [1,2,3], elems = [], acc = [(2, 1), (1, 1)]
```

Como la lista está vacía, la función devuelve:

```
reverse acc = [(1, 1), (2, 1)]
```

Estructura de la pila

Primera llamada: `ocurrenciasAux [1,2,3] [2] [(1, 1)]`
Segunda llamada: `ocurrenciasAux [1,2,3] [] [(2, 1), (1, 1)]`
Caso base: `reverse [(2, 1), (1, 1)] = [(1, 1), (2, 1)]`

Resultado final:

```
[(1, 1), (2, 1)]
```


Figueroa Barrientos Andrea Valeria 31705346
Martínez Hidalgo Paola Mildred 319300217
Hazel Torres Nava 319158496