Semanal 07

1. Dada la siguiente expresión en MiniLisp:

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

(a) **Ejecutarla y explicar el resultado.**

Haremos la evaluación perezosa y estática, entonces necesitamos cargar todos los ambienets de evaluación.

Primero empezamos por el let

$e =$

var	valor
sum	$< n, (if0\ \ n\ \ 0\ \ (+\ \ n\ \ (sum\ \ (-\ \ n\ \ 1)))), e_o >$

$e_0 =$

var	valor

Lo que queremos evaluar es:

```
sum 5
```

Ya no podemos pospononer más las evaluaciones, entonces comenzamos buscando *sum* en el ambiente principal.

Al encontrarlo sustituimos el cuerpo y metemos a su entorno el 5 con el ambiente e.

Quedando:

```
(if0 n 0 (+ n (sum (- n 1))))
```

$e =$

var	valor
sum	$< n, (if0\ \ n\ \ 0\ \ (+\ \ n\ \ (sum\ \ (-\ \ n\ \ 1)))), e_o >$

$e_0 =$

var	valor
n	$< 5, e >$

Ahora tampoco podemos posponer el IF0, entonces buscamos el valor de la n en el ambiente e_0, que es 5 que se evalúa con el ambiente e.

De esta forma obtenemos:

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

Y los ambientes quedan iguales, luego por el $if0$ obtenemos, se nos va al caso del else, entonces tomamos esa parte:

```
(+ n (sum (- n 1)))
```

Ya no podemos posponer la suma, entonces buscamos el valor de n, el cual al seguir en el ambiente e_0 sigue dando 5, recordando que el 5 trae su ambiente e, pero al evaluarse con ese ambiente queda igual.
Resultando en:

```
(+ 5 (sum (- n 1)))
```

$e =$

var	valor
sum	$< n, (if0\ \ n\ \ 0\ \ (+\ \ n\ \ (sum\ \ (-\ \ n\ \ 1)))), e_o >$

$e_0 =$

var	valor
n	$< 5, e >$

Lo siguiente que ya no se puede posponer al haber evaluado el 5 con su ambiente e, es ir a buscar a *sum* en el ambiente actual que es e_0, pero notamos que en e_0 no existe ninguna variable que se llame *sum*, por lo tanto tenemos un error, en particular un error de variable libre.
Y hasta ahí se detiene la ejecución.

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

Para poder usar el combinador de punto fijo Y, primero tenemos que aplicar un cambio importante.

Usando la notación de lambdas, tenemos que:

$$sum_{\texttt{def}} = \lambda n.(if0\ \ n\ \ 0\ \ (+\ \ n\ \ (sum\ \ (-\ \ n\ \ 1))))$$

El problema es que ese *sum* de adentro queda como variable libre, entonces lo que haremos es primero ligarla, agregando una lambda extra.

$$sum_{\texttt{def}} = \lambda f.\lambda n.(if0\ \ n\ \ 0\ \ (+\ \ n\ \ (f\ \ (-\ \ n\ \ 1))))$$

Además el combinador de punto fijo Y tiene la siguiente def:

$$Y_{\texttt{def}} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

Así que modificando el código para usar el combinador de punto fijo tenemos:

$$((Y\ \ sum)\ \ 5) =_{\texttt{def}} (((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ \ sum)\ \ 5)$$

$$\rightarrow_{\beta} (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ \ 5)$$

$$\rightarrow_\beta (((sum((\lambda x.sum(xx))(\lambda x.sum(xx))))) \ 5)$$

$$=_{\texttt{def}} ((((\lambda f.\lambda n.(if0 \ \ n \ \ 0 \ \ (+ \ \ n \ \ (f \ \ (- \ \ n \ \ 1)))))((\lambda x.sum(xx))(\lambda x.sum(xx))))) \ 5)$$

$$\rightarrow_\beta ((((\lambda n.(if0 \ \ n \ \ 0 \ \ (+ \ \ n \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ n \ \ 1))))))) \ 5)$$

$$\rightarrow_\beta (if0 \ \ 5 \ \ 0 \ \ (+ \ \ 5 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ 5 \ \ 1))))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ 5 \ \ 1)))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ (((sum((\lambda x.sum(xx))(\lambda x.sum(xx))))) \ \ (4)))$$

Por pasos anteriores sabemos que $sum((\lambda x.sum(xx))(\lambda x.sum(xx))$ se beta reduce a $(\lambda n.(if0 \ \ n \ \ 0 \ \ (+ \ \ n \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ n \ \ 1)))))$
Esta beta reducción la aplicaremos en repetidas ocasiones.

$$\rightarrow_\beta (+ \ \ 5 \ \ (\lambda n.(if0 \ \ n \ \ 0 \ \ (+ \ \ n \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ n \ \ 1))))) \ \ (4))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ ((if0 \ \ 4 \ \ 0 \ \ (+ \ \ 4 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ 4 \ \ 1))))))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ (+ \ \ 4 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ 4 \ \ 1))))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ (+ \ \ 4 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ 3)))$$

Nuevamente usamos la beta reducción previamente conocida.

$$\rightarrow_\beta (+ \ \ 5 \ \ (+ \ \ 4 \ \ ((\lambda n.(if0 \ \ n \ \ 0 \ \ (+ \ \ n \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ n \ \ 1))))) \ \ 3)))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ (+ \ \ 4 \ \ (((if0 \ \ 3 \ \ 0 \ \ (+ \ \ 3 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx))) \ \ (- \ \ 3 \ \ 1))))))))$$

$$\rightarrow_\beta (+ \ \ 5 \ \ (+ \ \ 4 \ \ (+ \ \ 3 \ \ (((\lambda x.sum(xx))(\lambda x.sum(xx)))2))$$

Otra vez aplicamos misma beta reducción

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ ((\lambda n.(if0\ n\ 0\ (+\ n\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ (-\ n\ 1)))))2))$$

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (((if0\ 2\ 0\ (+\ 2\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ (-\ 2\ 1)))))))$$

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ 1))))$$

Aplicando la misma beta reducción:

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2$$
$$((\lambda n.(if0\ n\ 0\ (+\ n\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ (-\ n\ 1)))))\ 1))))$$

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2$$
$$(((if0\ 1\ 0\ (+\ 1\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ (-\ 1\ 1)))))))))$$

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ 0))))))$$

Misma:

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ ((\lambda n.(if0\ n\ 0$$
$$(+\ n\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ (-\ n\ 1)))))\ 0))))))$$

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ (((if0\ 0\ 0$$
$$(+\ 0\ (((\lambda x.sum(xx))(\lambda x.sum(xx)))\ (-\ 0\ 1))))))))))$$

$$\to_\beta (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ 0)))))$$

Lo cual ya podremos evaluar y nos dará:

$$(+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ 1))))$$

$$(+\ 5\ (+\ 4\ (+\ 3\ 3)))$$

$$(+\ 5\ (+\ 4\ 6))$$

(+ 5 10)

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)
```

Define una variable `c` con valor *false*.
Se realiza una suma anidada y la vamos realizando de adentro hacia afuera.
`(+ (let/cc k (set! c k) 4) 5)`
La suma requiere que ambos elementos sean números, entonces tomamos lo que devuelve el `let/cc`, que es 4 y lo sumamos con 5.
`(+ 3 9)`
`(+ 2 12)`
`(+ 1 14)`
Y el resultado final de `(+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))` es 15.

Ahora, para evaluar *c 10* entonces evaluamos la continuación. `(+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))`.
Sumamos lo que está antes y lo que está después del `let/cc`, que es 11.
Entonces `11 10` da como resultado 21.

La continuación asociada es la siguiente:

```
(+ 1 (+ 2 (+ 3 (+
( λ↑(v) (+ 1 (+ 2 (+ 3 (+ v 5) ) ) ) )
5))))
```

3. Realizar los siguientes ejercicios en Haskell:

   (a) 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.

   Para esto vamos a utilizar la función ocElem, la cual va a generar la lista de parejas que contenga a cada elemento de la segunda lista y su conteo de apariciones en la primera lista, para obtener este conteo de apariciones ocupamos una función auxiliar contElem que nos dará el número de apariciones de un elemento en una lista

```
ocElem :: Eq a => [a] -> [a] -> [(a,Int)]
ocElem (x:xs) [] = []
ocElem xs (y:ys) = (y, contElem xs y) : ocElem xs ys

contElem :: Eq a => [a] -> a -> Int
contElem [] y = 0
contElem (x:xs) y
    | x == y = 1 + contElem xs y
    | otherwise = contElem xs y
```

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

```
ocElem [1,2,3] [1,2]
= (1, contElem [1,2,3] 1) : ocElem [1,2,3] [2]
= (1, 1 + contElem [2,3] 1) : ocElem [1,2,3] [2]
= (1, 1 + contElem [3] 1) : ocElem [1,2,3] [2]
= (1, 1 + contElem [] 1) : ocElem [1,2,3] [2]
= (1, 1 + 0) : ocElem [1,2,3] [2]
= (1,1) :  ocElem [1,2,3] [2]
= (1,1) : (2, contElem [1,2,3] 2) : ocElem [1,2,3] []
= (1,1) : (2, contElem [2,3] 2) : ocElem [1,2,3] []
= (1,1) : (2, 1 + contElem [3] 2) : ocElem [1,2,3] []
= (1,1) : (2, 1 + contElem [] 2) : ocElem [1,2,3] []
= (1,1) : (2, 1 + 0) : ocElem [1,2,3] []
= (1,1) : (2, 1) : ocElem [1,2,3] []
= (1,1) : (2, 1) : ocElem [1,2,3]
= (1,1) : (2, 1) :  []
= [(1,1), (2, 1)]
```

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

Transformamos entonces la función ocElem para que el acumulador mantenga la lista de las parejas, y la función contElem para que el acumulador lleve la cuenta de las apariciones.

```
ocElem :: Eq a => [a] -> [a] -> [Int] -> [(a,Int)]
ocElem (x:xs) [] acc = acc
ocElem xs (y:ys) acc = ocElem xs ys (acc ++ (y, contElem xs y))

contElem :: Eq a => [a] -> a -> Int -> Int
contElem [] y acc = acc
contElem (x:xs) y acc
    | x == y = contElem xs y (acc + 1)
    | otherwise = contElem xs y acc
```

(d)
```
ocElem [1,2,3] [1,2] []
= ocElem [1,2,3] [2] ([] ++ (1, contElem [1,2,3] 1 0))
= ocElem [1,2,3] [] ([] ++ (1, contElem [1,2,3] 1 0) ++ (2, contElem [1,2,3]
= ([] ++ (1, contElem [1,2,3] 1 0) ++ (2, contElem [1,2,3] 2 0))
= ([] ++ (1, contElem [2,3] 1 (0 + 1)) ++ (2, contElem [1,2,3] 2 0))
= ([] ++ (1, contElem [3] 1 (0 + 1)) ++ (2, contElem [1,2,3] 2 0))
= ([] ++ (1, contElem [] 1 (0 + 1)) ++ (2, contElem [1,2,3] 2 0))
= ([] ++ (1, (0 + 1)) ++ (2, contElem [1,2,3] 2 0))
= ([] ++ (1, 1) ++ (2, contElem [1,2,3] 2 0))
= ([] ++ (1, 1) ++ (2, contElem [2,3] 2 0))
= ([] ++ (1, 1) ++ (2, contElem [3] 2 (0 + 1)))
= ([] ++ (1, 1) ++ (2, contElem [] 2 (0 + 1)))
= ([] ++ (1, 1) ++ (2, (0+1)))
= ([] ++ (1, 1) ++ (2, 1))
= [(1, 1), (2, 1)]
```