UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS
LENGUAJES DE PROGRAMACIÓN

Semanal 07

Ana Lilia Carballido Camacateco - 315314601

Oscar Fernando Frias Dominguez - 314255662

1 Ejercicios

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.
     Resultado de la ejecución: "Error de variable libre: Sum" porque sum es una función que recibe un número y en éste caso espera una expresión.

   - Modificarla usando el combinador de punto fijo Y, volver a ejecutarla y explicar el resultado.
     Recordemos que el combinador Y se define como sigue:

$$Y = def\lambda f(\lambda x \cdot f(xx)(\lambda x \cdot f(xx)))$$

     La definición de sum es:

$$sum := \lambda(n) : If0ntheen0else(+n(sum(n-1)))$$

     Si aplicamos el combinador Y a la función sum con un argumento dígamos 5, entonces:

$$(Ysum)5 = [(\lambda f(\lambda x \cdot f(xx))(\lambda x \cdot f(xx)))sum]5$$

$$\rightarrow (\lambda x \cdot sum(xx))(\lambda x \cdot sum(xx))5$$

$$\rightarrow (sum[(\lambda x \cdot sum(xx))(\lambda x \cdot sum(xx))])5$$

     Pero sum recibe un número y no una aplicación de función por lo que modificamos la función sum para recibir funciones de la siguiente manera:

$$sum' =_{def} \lambda(f) : \lambda(n) : if0nthen0else(+n(F(n-1)))$$

     Entonces, ejecutando la expresión con la nueva definición de sum' que permite tener una función como argumento la cual se puede resolver en cada paso, en éste caso la resta de $(-n1)$, tenemos la expresión:

$$(let(sum'(\lambda(f) : \lambda(n)(if0n0(+n(sum'(-n1))))))(sum5))$$

$$= sum'(5) = 5 + sum'(4)$$

$$= 5 + 4 + sum'(3)$$

$$= 5 + 4 + 3 + sum'(2)$$

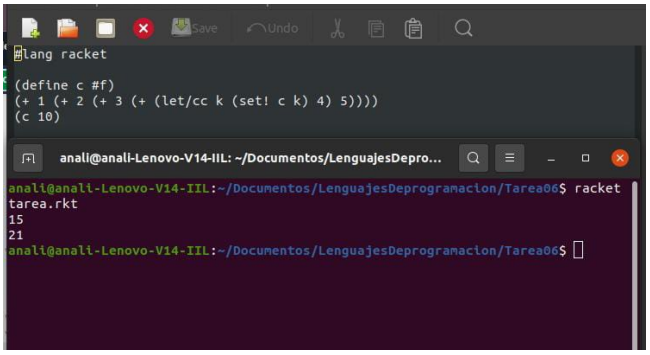
$$= 5 + 4 + 3 + 2 + sum(1)$$

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

$$= 5 + 4 + 3 + 2 + 1 + 0 = 15$$

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)
```



Explicación:
Primero se imprime el número 15 porque es el resultado cuando la continuación aún no se aplica con $c = 10$. Entonces es el resultado de calcular:

$$(+1(+2(+3(+4)5)))) = 15$$

Luego se imprime 21 porque la continuación nos permite darle un valor a k en ése momento de la suma, entonces cuando se aplica $c = 10$, el resultado se calcula como :

$$(+1(+2(+3(+10)5)))) = 21$$

Continucaciòn asociada:

$$(\lambda \uparrow (v)(+1(+2(+3(+v5))))((set!ck)4)$$

3. Realizar los siguientes ejercicios en Haskell:

   - Definir la función recursiva `ocurreciasElementos` 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)]
     ```

     ```haskell
     ocurrenciasElementos :: [Int] -> [Int] -> [(Int, Int)]
     ocurrenciasElementos _ [] = []
     ocurrenciasElementos xs (y:ys) = [(y, contarOcurrencias y xs)] ++ ocurrenciasElementos xs ys



     contarOcurrencias :: Int -> [Int] -> Int
     contarOcurrencias _ [] = 0
     contarOcurrencias a (x:xs)
         | a == x = 1 + contarOcurrencias a xs
         | otherwise = contarOcurrencias a xs
     ```

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

     Ejecución de `ocurrenciasElementos [1,2,3] [1,2]`

     Al ejecutar la función `ocurrenciasElementos [1,2,3] [1,2]`, se generan los siguientes registros de activación.

Primer paso de la ejecución

(a) Llamada a `ocurrenciasElementos [1,2,3] [1,2]`:
  - `(y:ys) = (1:2)`
  - Llama a `contarOcurrencias 1 [1,2,3]`

(b) Llamada a `contarOcurrencias 1 [1,2,3]`:
  - El primer elemento coincide con 1.
  - Llama a `contarOcurrencias 1 [2,3]`

(c) Llamada a `contarOcurrencias 1 [2,3]`:
  - El primer elemento no coincide ($2 \neq 1$).
  - Llama a `contarOcurrencias 1 [3]`

(d) Llamada a `contarOcurrencias 1 [3]`:
  - El primer elemento no coincide ($3 \neq 1$).
  - Llama a `contarOcurrencias 1 []`

(e) Llamada a `contarOcurrencias 1 []`:
  - Lista vacía, retorna 0.

(f) Resultado de `contarOcurrencias 1 [1,2,3]`:
  - El resultado final es 1, porque el número 1 aparece una vez en `[1,2,3]`.

(g) Continuación de la llamada a `ocurrenciasElementos [1,2,3] [1,2]`:
  - El primer par de la lista es `(1,1)`.

Segundo paso de la ejecución

(a) Llamada a `ocurrenciasElementos [1,2,3] [2]`:
  - `(y:ys) = (2:[])`
  - Llama a `contarOcurrencias 2 [1,2,3]`

(b) Llamada a `contarOcurrencias 2 [1,2,3]`:
  - El primer elemento no coincide ($1 \neq 2$).
  - Llama a `contarOcurrencias 2 [2,3]`

(c) Llamada a `contarOcurrencias 2 [2,3]`:
  - El primer elemento coincide ($2 = 2$).
  - Llama a `contarOcurrencias 2 [3]`

(d) Llamada a `contarOcurrencias 2 [3]`:
  - El primer elemento no coincide ($3 \neq 2$).
  - Llama a `contarOcurrencias 2 []`

(e) Llamada a `contarOcurrencias 2 []`:
  - Lista vacía, retorna 0.

(f) Resultado de `contarOcurrencias 2 [1,2,3]`:
  - El resultado final es 1, porque el número 2 aparece una vez en `[1,2,3]`.

(g) Continuación de la llamada a `ocurrenciasElementos [1,2,3] [2]`:
  - El segundo par de la lista es `(2,1)`.

Tercer paso de la ejecución

(a) Llamada a `ocurrenciasElementos [1,2,3] []`:
  - Lista vacía, retorna `[]`.

Resultado final

El resultado final es la lista de pares:

$$[(1,1),(2,1)]$$

- Optimizarla función definida usando recursión de cola. Debes transformar todas las funciones auxiliares que utilicen.

```haskell
ocurrenciasElementosTail :: [Int] -> [Int] -> [(Int,Int)] -> [(Int,Int)]
ocurrenciasElementosTail _ [] acc = acc
ocurrenciasElementosTail xs (y:ys) acc  = ocurrenciasElementosTail xs ys (acc ++ [(y, contarOcurrenciasTail y xs 0)])


contarOcurrenciasTail ::  Int -> [Int] -> Int -> Int
contarOcurrenciasTail _ [] acc = acc
contarOcurrenciasTail a (x:xs) acc
    | a == x = contarOcurrenciasTail a xs (acc+1)
    | otherwise = contarOcurrenciasTail a xs acc
```

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

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

Al ejecutar la función `ocurrenciasElementosTail [1,2,3] [1,2] []`, los siguientes registros de activación se generan paso a paso.

Primer paso de la ejecución

(a) Llamada a `ocurrenciasElementosTail [1,2,3] [1,2] []`:
  - Se extrae `y = 1`, `ys = [2]`.
  - Llama a `contarOcurrenciasTail 1 [1,2,3] 0`.

(b) Llamada a `contarOcurrenciasTail 1 [1,2,3] 0`:
  - El primer elemento coincide con 1.
  - Llama a `contarOcurrenciasTail 1 [2,3] 1`.

(c) Llamada a `contarOcurrenciasTail 1 [2,3] 1`:
  - El primer elemento no coincide (2 1).
  - Llama a `contarOcurrenciasTail 1 [3] 1`.

(d) Llamada a `contarOcurrenciasTail 1 [3] 1`:
  - El primer elemento no coincide ($3 \neq 1$).
  - Llama a `contarOcurrenciasTail 1 [] 1`.

(e) Llamada a `contarOcurrenciasTail 1 [] 1`:
  - Lista vacía, retorna 1.

(f) Continuación de `ocurrenciasElementosTail [1,2,3] [1,2] []`:
  - Agrega el par `(1, 1)` a la lista de acumulación.
  - Llama a `ocurrenciasElementosTail [1,2,3] [2] [(1, 1)]`.

Segundo paso de la ejecución

(a) Llamada a `ocurrenciasElementosTail [1,2,3] [2] [(1, 1)]`:
  - Se extrae `y = 2`, `ys = []`.
  - Llama a `contarOcurrenciasTail 2 [1,2,3] 0`.

(b) Llamada a `contarOcurrenciasTail 2 [1,2,3] 0`:
  - El primer elemento no coincide ($1 \neq 2$).
  - Llama a `contarOcurrenciasTail 2 [2,3] 0`.

(c) Llamada a `contarOcurrenciasTail 2 [2,3] 0`:
  - El primer elemento coincide ($2 = 2$).
  - Llama a `contarOcurrenciasTail 2 [3] 1`.

(d) Llamada a `contarOcurrenciasTail 2 [3] 1`:
  - El primer elemento no coincide ($3 \neq 2$).
  - Llama a `contarOcurrenciasTail 2 [] 1`.

(e) Llamada a `contarOcurrenciasTail 2 [] 1`:

- Lista vacía, retorna 1.

(f) Continuación de `ocurrenciasElementosTail [1,2,3] [2] [(1, 1)]`:
- Agrega el par `(2, 1)` a la lista de acumulación.
- Llama a `ocurrenciasElementosTail [1,2,3] [] [(1, 1), (2, 1)]`.

Tercer paso de la ejecución

(a) Llamada a `ocurrenciasElementosTail [1,2,3] [] [(1, 1), (2, 1)]`:
- La lista `ys` está vacía, retorna la lista acumulada `[(1, 1), (2, 1)]`.

Resultado final

El resultado final de la ejecución de `ocurrenciasElementosTail [1,2,3] [1,2] []` es la lista de pares:

$$[(1,1),(2,1)]$$