Evaluación semanal 7

Santiago González Tamariz
Lenguajes de Programación

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.

  Paso 1:

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

  $\epsilon =$ | sum | $\langle$n, (if0 n 0 (+ n (sum (- n 1)))), $\epsilon_1\rangle$ | $\quad \epsilon_1 =$ | | |
  Añadimos la cerradura de `sum` al ambiente y creamos el subambiente correspondiente

  Paso 2:

  ```
  (sum 5)
  ```

  $\epsilon =$ | $\Rightarrow$sum | $\langle$n, (if0 n 0 (+ n (sum (- n 1)))), $\epsilon_1\rangle$ | $\quad \epsilon_1 =$ | n | 5 |
  Es necesario evaluar `sum` para poder continuar

  Paso 3:

  ```
  (if0 n 0 (+ n (sum (- n 1)))),ϵ1
  ```

  $\epsilon =$ | sum | $\langle$n, (if0 n 0 (+ n (sum (- n 1)))), $\epsilon_1\rangle$ | $\quad \epsilon_1 =$ | $\Rightarrow$n | 5 |
  Es necesario evaluar la condición del `if0` para continuar

  Paso 4:

  ```
  (if0 5 0 (+ n (sum (- n 1)))),ϵ1
  ```

  $\epsilon =$ | sum | $\langle$n, (if0 n 0 (+ n (sum (- n 1)))), $\epsilon_1\rangle$ | $\quad \epsilon_1 =$ | n | 5 |
  Tomamos el else

  Paso 5:

  ```
  (+ 5 (sum (- n 1))),ϵ1
  ```

  $\epsilon =$ | sum | $\langle$n, (if0 n 0 (+ n (sum (- n 1)))), $\epsilon_1\rangle$ | $\quad \epsilon_1 =$ | $\Rightarrow$n | 5 |
  Es necesario evaluar la suma, iniciamos por el lado izquierdo

  Paso 6:

  ```
  (+ 5 (sum (- n 1))),ϵ1
  > error: variable libre
  ```

  $\epsilon =$ | sum | $\langle$n, (if0 n 0 (+ n (sum (- n 1)))), $\epsilon_1\rangle$ | $\quad \epsilon_1 =$ | n | 5 |
  Al buscar `sum` en el subambiente actual vemos que no está definido por lo que hay un error de variable libre

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

  Recordemos que $Y =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ y que $(Y\ fun)a = fun\ (Y\ fun)\ a$, tambien hay que añadirle un parámetro extra a nuestra función.

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

  Entonces aplicando el combinador de punto fijo nos queda

  $$((Y\ sum)\ 5) = sum\ (Y\ sum)\ 5 = (\lambda f.\lambda n.(if0\ n\ 0\ (+\ n\ (f\ (-\ n\ 1)))))\ (Y\ sum)\ 5$$

$$= (\lambda n.(if0\ n\ 0\ (+\ n\ ((Y\ sum)\ (-\ n\ 1))))) \ 5 = (if0\ 5\ 0\ (+\ 5\ ((Y\ sum)\ (-\ 5\ 1))))$$
$$= (+\ 5\ (sum\ (Y\ sum)\ 4)) = (+\ 5\ ((\lambda f.\lambda n.(if0\ n\ 0\ (+\ n\ (f\ (-\ n\ 1)))))\ (Y\ sum)\ 4))$$
$$= (+\ 5\ (if0\ 4\ 0\ (+\ 4\ ((Y\ sum)\ (-\ 4\ 1))))) = (+\ 5\ (+\ 4\ (sum\ (Y\ sum)\ 3)))$$
$$= (+\ 5\ (+\ 4\ ((\lambda f.\lambda n.(if0\ n\ 0\ (+\ n\ (f\ (-\ n\ 1)))))\ (Y\ sum)\ 3)))$$
$$= (+\ 5\ (+\ 4\ (if0\ 3\ 0\ (+\ 3\ ((Y\ sum)\ (-\ 3\ 1)))))) = (+\ 5\ (+\ 4\ (+\ 3\ (sum\ (Y\ sum)\ 2))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ ((\lambda f.\lambda n.(if0\ n\ 0\ (+\ n\ (f\ (-\ n\ 1)))))\ (Y\ sum)\ 2))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ (if0\ 2\ 0\ (+\ 2\ ((Y\ sum)\ (-\ 2\ 1))))))) = (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (sum\ (Y\ sum)\ 1)))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ ((\lambda f.\lambda n.(if0\ n\ 0\ (+\ n\ (f\ (-\ n\ 1)))))\ (Y\ sum)\ 1)))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (if0\ 1\ 0\ (+\ 1\ ((Y\ sum)\ (-\ 1\ 1)))))))) = (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ (sum\ (Y\ sum)\ 0))))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ ((\lambda f.\lambda n.(if0\ n\ 0\ (+\ n\ (f\ (-\ n\ 1)))))\ (Y\ sum)\ 0))))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ (if0\ 0\ 0\ (+\ 0\ ((Y\ sum)\ (-\ 0\ 1)))))))))) = (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ (+\ 1\ 0)))))$$
$$= (+\ 5\ (+\ 4\ (+\ 3\ (+\ 2\ 1)))) = (+\ 5\ (+\ 4\ (+\ 3\ 3))) = (+\ 5\ (+\ 4\ 6)) = (+\ 5\ 10) = 15$$

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)
```

Primero se define la variable `c` como falso, luego tenemos una suma, entonces evaluamos la suma iniciando por lo más anidado la parte izquierda. El `let/cc` nos devuelve 4, entonces ya podemos hacer la suma.

```
> (+ 1 (+ 2 (+ 3 (+ 4 5))))
> (+ 1 (+ 2 (+ 3 9)))
> (+ 1 (+ 2 12))
> (+ 1 14)
> 15
```

También tenemos que guarda la continuación en `c`

$$(\lambda_{\uparrow}(v)\ (+\ 1\ (+\ 2\ (+\ 3\ (+\ v\ 5)))))$$

Entonces al evaluar `c 10` nos queda

```
> (c 10)
> ((λ↑(v) (+ 1 (+ 2 (+ 3 (+ v 5))))) 10)
> (+ 1 (+ 2 (+ 3 (+ 10 5))))
> (+ 1 (+ 2 (+ 3 15)))
> (+ 1 (+ 2 18))
> (+ 1 20)
> 21
```

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

   - Definir la función recurisva `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)]
     ```

     Asumiendo que la segunda lista siempre tendrá un tamaño pequeño, podemos recorrer para cada uno de sus elementos a la primera lista e ir contando cuantas veces aparece. En caso de que la segunda lista tenga un tamaño parecido a la primera lista nos conviene más ordenar las listas e ir checando cuantos elementos contiguos del mismo hay y si cambia entonces pasar a contar las del sigiente elemento.

     ```haskell
     module Ej3_1 where
     -- ocurrenciasElementos
     oe :: [Int] -> [Int] -> [(Int, Int)]
     oe _ [] = []
     oe l1 (x:xs) = (x, ct x l1) : oe l1 xs

     -- Cuenta
     ct :: Int -> [Int] -> Int
     ct _ [] = 0
     ct y (x:xs) = if y == x then 1 + ct y xs else ct y xs

     ocurrenciasElementos = oe
     ```

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

     ```haskell
     ocurrenciasElementos [1,2,3] [1,2]
     = oe [1,2,3] [1,2]
     = (1, ct 1 [1,2,3]) : oe [1,2,3] [2]
     = (1, 1 + ct 1 [2,3]) : oe [1,2,3] [2]
     = (1, 1 + ct 1 [3]) : oe [1,2,3] [2]
     = (1, 1 + ct 1 []) : oe [1,2,3] [2]
     = (1, 1 + 0) : oe [1,2,3] [2]
     = (1,1) : oe [1,2,3] [2]
     = (1,1) : (2, ct 2 [1,2,3]) : oe [1,2,3] []
     = (1,1) : (2, ct 2 [2,3]) : oe [1,2,3] []
     = (1,1) : (2, 1 + ct 2 [3]) : oe [1,2,3] []
     = (1,1) : (2, 1 + ct 2 []) : oe [1,2,3] []
     = (1,1) : (2, 1 + 0) : oe [1,2,3] []
     = (1,1) : (2, 1) : oe [1,2,3] []
     = (1,1) : (2, 1) : oe [1,2,3]
     = (1,1) : (2, 1) : []
     = [(1,1), (2, 1)]
     ```

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

     ```haskell
     module Ej3_2 where
     -- ocurrenciasElementos
     oe :: [Int] -> [Int] -> [(Int, Int)] -> [(Int, Int)]
     oe _ [] acc = acc
     oe l1 (x:xs) acc = oe l1 xs (acc ++ [(x, ct x l1 0)])

     -- Cuenta
     ct :: Int -> [Int] -> Int -> Int
     ct _ [] acc = acc
     ct y (x:xs) acc = if y == x then ct y xs (acc+1) else ct y xs acc

     ocurrenciasElementos a b = oe a b []
     ```

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

```
ocurrenciasElementos [1,2,3] [1,2]
= oe [1,2,3] [1,2] []
= oe [1,2,3] [2] ([] ++ [(1, ct 1 [1,2,3] 0)])
= oe [1,2,3] [2] ([] ++ [(1, ct 1 [2,3] 1)])
= oe [1,2,3] [2] ([] ++ [(1, ct 1 [3] 1)])
= oe [1,2,3] [2] ([] ++ [(1, ct 1 [] 1)])
= oe [1,2,3] [2] ([] ++ [(1, 1)])
= oe [1,2,3] [2] [(1, 1)]
= oe [1,2,3] [] ([(1, 1)] ++ [(2, ct 2 [1,2,3] 0)])
= oe [1,2,3] [] ([(1, 1)] ++ [(2, ct 2 [2,3] 0)])
= oe [1,2,3] [] ([(1, 1)] ++ [(2, ct 2 [3] 1)])
= oe [1,2,3] [] ([(1, 1)] ++ [(2, ct 2 [] 1)])
= oe [1,2,3] [] ([(1, 1)] ++ [(2, 1)])
= oe [1,2,3] [] [(1, 1), (2, 1)]
= [(1, 1), (2, 1)]
```