Semanal 07

- del Valle Vera Nancy Elena
- Juárez Cruz Joshua
- Sánchez Victoria Leslie Paola

Dada la siguiente expresión en MiniLisp

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

- **Ejecutarla y explicar el resultado.**

Añadimos sum a la pila con su respectiva cerradura.

sum	<n, (if0 n 0 (+ n (sum (- n 1)))), []>

Evaluamos el cuerpo del let.

sum 5 = (lambda (n) (if0 n 0 (+ n (sum (- n 1))))) 5

Añadimos [n:=5] al subambiente

sum	<n, (if0 n 0 (+ n (sum (- n 1)))), [n ← 5]>

Evaluamos (if0 n 0 (+ n (sum (- n 1)))) buscando el valor de los identificadores en el ambiente de sum

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

Tenemos un error de variable libre

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

Definimos al combinador Y de la siguiente manera:

$$Y =_{def} \lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$$
$$func =_{def} \lambda f.\lambda n.\,\text{if } n = 0 \text{ then } 0 \text{ else } n + f(n-1)$$

Realizamos la ejecucion usando el combinador Y. Primero notemos que:

$$
\begin{aligned}
Y\ func &=_{def}\ (\lambda f.(\lambda x.f(x\,x))\,)\,(\lambda x.f(x\,x))\,)\,func \\
&\to (\lambda x.func\,(x\,x)\,)\,(\lambda x.func\,(x\,x)) \\
&\to func((\lambda x.func\,(x\,x)\,)\,(\lambda x.func\,(x\,x))) =_{def}\ func\,(Y\ func)
\end{aligned}
$$

Así la evaluación es la siguiente:

$$
\begin{aligned}
&(Y\ \ func)\,5 \to^* (func\ (Y\ func))\,5 \\
&=_{def}\ ((\lambda f.\lambda n.\,\text{if } n = 0 \text{ then } 0 \text{ else } n + f(n-1))(Y\ func))\ 5 \\
&\to (\lambda n.\,\text{if } n = 0 \text{ then } 0 \text{ else } n + (Y\ \ func)(n-1))\ 5 \\
&\to \text{if } 5 = 0 \text{ then } 0 \text{ else } 5 + (Y\ \ func)\ 4 \\
&\to 5 + (Y\ \ func)4 \\
&\to^* 5 + (func\,(Y\ \ func))\ 4 \\
&\to^* 5 + (\text{if } n = 0 \text{ then } 0 \text{ else } n + (Y\ \ func)(n-1))\ 4 \\
&\to 5 + (\text{if } 4 = 0 \text{ then } 0 \text{ else } 4 + (Y\ \ func)\ 3) \\
&\to 5 + 4 + (Y\ \ func)\ 3 \\
&\to^* 5 + 4 + (func\ (Y\ \ fucn))\ 3 \\
&\to^* 5 + 4 + (\text{if } n = 0 \text{ then } 0 \text{ else } n + (Y\ \ func)(n-1))\ 3 \\
&\to 5 + 4 + (\text{if } 3 = 0 \text{ then } 0 \text{ else } 3 + (Y\ \ func)\ 2) \\
&\to 5 + 4 + 3 + (Y\ \ func)\ 2 \\
&\to^* 5 + 4 + 3 + (func\ (Y\ \ fucn))\ 2 \\
&\to^* 5 + 4 + 3(\text{if } n = 0 \text{ then } 0 \text{ else } n + (Y\ \ func)(n-1))\ 2 \\
&\to 5 + 4 + 3(\text{if } 2 = 0 \text{ then } 0 \text{ else } 2 + (Y\ \ func)\ 1) \\
&\to 5 + 4 + 3 + 2 + (Y\ \ func)\ 1 \\
&\to^* 5 + 4 + 3 + 2 + (func\ (Y\ \ fucn))\ 1 \\
&\to^* 5 + 4 + 3 + 2(\text{if } n = 0 \text{ then } 0 \text{ else } n + (Y\ \ func)(n-1))\ 1 \\
&\to 5 + 4 + 3 + 2(\text{if } 1 = 0 \text{ then } 0 \text{ else } 1 + (Y\ \ func)\ 0) \\
&\to 5 + 4 + 3 + 2 + 1 + (Y\ \ func)\ 0 \\
&\to^* 5 + 4 + 3 + 2 + 1 + (func\ (Y\ \ fucn))\ 0 \\
&\to^* 5 + 4 + 3 + 2 + 1(\text{if } n = 0 \text{ then } 0 \text{ else } n + (Y\ \ func)(n-1))\ 0 \\
&\to 5 + 4 + 3 + 2 + 1(\text{if } 0 = 0 \text{ then } 0 \text{ else } 0 + (Y\ \ func)\ \ -1) \\
&\to 5 + 4 + 3 + 2 + 1 + 0 \to 15
\end{aligned}
$$

(Evaluar la siguiente expresión en Racket, explicar su resultado y dar la continuación asociada a evaluar usando la notación λ↑.

```
> (define c #f)
> (+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
```

```
> (c 10)
```

El identificador k guarda la continuación asociada en el momento actual. Dicha continuación es:

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

Luego, este valor se guarda en c y se realizamos la primera evaluación de la continuación guardada en c con el valor 4. El resultado es el siguiente:

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

Luego realizamos la evaluación de la continuación guardada en la viriable c con el valor 10 y obtemos como resultado:

```
> (+ 1 (+ 2 (+ 3 (+ 10 5))))
> (+ 1 (+ 2 (+ 3 15)))
> (+ 1 (+ 2 18))
> (+ 1 20)
>  21
```

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.**

```haskell
ocurrenciasElementos :: Eq a => [a] -> [a] -> [(a,Int)]
ocurrenciasElementos l [] = []
ocurrenciasElementos l (x:xs) = (x,count l x):(ocurrenciasElementos l xs)

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

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

[] ocurrenciasElementos l [] = [] ocurrenciasElementos l (x:xs) = (x,count l x) (ocurrenciasElementos l xs) [1,2,3] [] ocurrenciasElementos
0 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [] 2 count
count [] 2 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [3] 1 count
1+count [3] 2 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [2,3] 2 count
count [2,3] 2 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [1,2,3] 2 count
(2,count [1,2,3] 2):(ocurrenciasElementos [1,2,3] []) ocurrenciasElementos l [] = [] ocurrenciasElementos l (x:xs) = (x,count l x) (ocurrenciasElementos l xs) [1,2,3] [2] ocurrenciasElementos

0 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [] 1 count
count [] 1 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [3] 1 count
count [3] 1 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [2,3] 1 count
1+count [2,3] 1 count [] n = 0 count (x:xs) n | x == n = 1 + count xs n | otherwise = count xs n [1,2,3] 1 count
(1,count [1,2,3] 1):(ocurrenciasElementos [1,2,3] [2]) ocurrenciasElementos l [] = [] ocurrenciasElementos l (x:xs) = (x,count l x) (ocurrenciasElementos l xs) [1,2,3] [1,2] ocurrenciasElementos

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

Transformación a recursión de cola de la función auxiliar `count`. Queremos una nueva función `countRC` , la definimos como `countRC xs e acc = acc + count xs e` .

Caso base:

En la versión optimizada, solo regresamos el acumulador

`countRC [] n acc` = `acc + count [] n`

= `acc + 0`

= `acc`

Hipótesis de inducción: Supongamos que `countRC xs n acc = acc + count xs n`

Paso Inductivo:

Tenemos dos casos:

- Si x==n.

`countRC (x:xs) n acc` = `acc + count (x:xs) n` (definición de countRC)

= `acc + (1 + count xs n)` (definición de count cuando x==n)

= `(acc + 1) + count xs n` (conmutatividad de la suma)

= `countRC xs n (acc + 1)` (H.I)

Como la función `count` devuelve un entero, el acumulador también es un entero, así que podemos sumar uno al acumulador.

- Si x/=n.

`countRC (x:xs) n acc` = `acc + count (x:xs) n` (definición de countRC)

= `acc + count xs n` (definición de count cuando x/=n)

= `countRC xs n acc` (H.I)

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

Transformación a recursión de cola de la función `ocurrenciasElementos`

Caso base:

En la versión optimizada, solo regresamos el acumulador.

`ocurrenciasElementosRC l [] acc = acc`

Hipótesis de inducción:

Supongamos `ocurrenciasElementosRC l xs acc = ocurrenciasElementos l xs`

Paso inductivo:

`ocurrenciasElementosRC l (x:xs) acc` = `ocurrenciasElementos l (x:xs)` (def. ocurrenciasElementosRC)

= `(x,count l x):(ocurrenciasElementos l xs)` (def. ocurrenciasElementos)

= `(x,count l x):(ocurrenciasElementosRC l xs acc)` (H.I)

= `[(x,count l xs)] ++ (ocurrenciasElementosRC l xs acc)` ((x:xs) ≡ [x]++xs)

= `ocurrenciasElementosRC l xs (acc ++ [(x,count l xs)])`

El acumulador debe ser una lista de duplas, así que podemos concatenar el acumulador con la última dupla que se ha calculado.

= `ocurrenciasElementosRC l xs (acc ++ [(x,countRC l x 0)]` (Transformación de cola de la función `count`)

Finalmente:

`ocurrenciasElementosRC l (x:xs) acc` = `ocurrenciasElementosRC l xs (acc++[(x, countRC l x 0)])`

```
ocurrenciasElementosRC :: Eq a => [a] -> [a] -> [(a, Int)] -> [(a,Int)]
ocurrenciasElementosRC l [] acc = acc
ocurrenciasElementosRC l (x:xs) acc = ocurrenciasElementosRC l xs (acc++[(x,countRC l x 0)])
```

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

```
ocurrenciasElementosRC [1,2,3] [2] ([]++[(1,countRC [1,2,3] 1 0)])
ocurrenciasElementosRC l [] acc = acc ocurrenciasElementosRC l (x:xs) acc =
ocurrenciasElementosRC l xs (acc++[(x,countRC l x 0)])
[1,2,3] [1,2] []
ocurrenciasElementosRC
```

```
ocurrenciasElementosRC [1,2,3] [] ([]++[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])
ocurrenciasElementosRC l [] acc = acc ocurrenciasElementosRC l (x:xs) acc =
ocurrenciasElementosRC l xs (acc++[(x,countRC l x 0)])
[1,2,3] [2] ([]++[(1,countRC [1,2,3] 1 0)])
ocurrenciasElementosRC
```

```
([]++[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])
ocurrenciasElementosRC l [] acc = acc ocurrenciasElementosRC l (x:xs) acc =
ocurrenciasElementosRC l xs (acc++[(x,countRC l x 0)])
[1,2,3] [] ([]++[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])
ocurrenciasElementosRC
```

countRC [2,3] (0+1)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise = countRC xs n acc
[1,2,3] 1 0
countRC

[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])

countRC [3] (0+1)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise = countRC xs n acc
[2,3] 1 (0+1)
countRC

[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])

countRC [] (0+1)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise = countRC xs n acc
[3] 1 (0+1)
countRC

[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])

0+1

countRC [] (0+1)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise = countRC xs n acc
[] 1 (0+1)
countRC

[(1,countRC [1,2,3] 1 0)] ++ [(2,countRC [1,2,3] 2 0)])

countRC [2,3] 0
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise = countRC xs n acc

```
[1,2,3] 2 0
countRC
```

```
[(1,1), (2,countRC [1,2,3] 2 0)])
```

```
countRC [3] (1+0)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise =
countRC xs n acc
[2,3] 2 0
countRC
```

```
[(1,1), (2,countRC [1,2,3] 2 0)])
```

```
countRC [] (1+0)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise =
countRC xs n acc
[3] 2 (1+0)
countRC
```

```
[(1,1), (2,countRC [1,2,3] 2 0)])
```

```
(1+0)
count [] n = 0 countRC (x:xs) n acc | x == n = countRC xs n (acc + 1)| otherwise =
countRC xs n acc
[3] [] (1+0)
countRC
```

```
[(1,1), (2,countRC [1,2,3] 2 0)])
```

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