Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de programación
Semanal 6
Jacome Delgado Alejandro (320011704)
Jimenez Sanchez Emma Alicia (320046155)
15 de octubre de 2023

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

- Ejecutarla y explicar el resultado. Ejecucuión:

```
Sustituimos [sum : = (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))]

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

Sustitumos [n := 5]

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

Evaluamos la función :

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

 Como 5 no es igual a  0, entonces tomamos:

 (+ 5 (sum (- 5 1)))

 (+ 5 (sum (4)))
```

La pila:

$$\Rightarrow \begin{array}{|c|c|}\hline \text{sum} & lambda(n)(\text{if0 } n \text{ 0 } (+n(\text{sum}(-\ n\ 1)))) \\ \hline n & 5 \\ \hline \end{array}$$

Al estar haciendo la evaluación con la pila que se muestra, vemos que la variable *sum* queda como variable libre por lo que daría un `error` la evaluación.

- Modificarla usando el combinador de punto fijo Y, volver a ejecutarla y explicar el resultado.
Se debe de hacer una modificación a nuestra función *sum*, ya que se debe de usar el combinador de punto de fijo Y, por lo que recordemos que $Yg =_{def} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) = g(Yg)$, por lo que al incoporarlo a nuestra función de *sum* para obtener la recursión, ademas tenemos que modificar la función suma para que funcione con nuestro combinador, lo que sería:

```
(let(Y(lambda(f)((lambda(x)(f(xx)))((lambda(x)(f(xx))))))
    (let(sum(Y(lambda(sum)(lambda (f) (lambda(n) ( if0 n 0 (+ n ( f (- n 1))
        (sum 5))
```

Por lo que al evaluar la función de *sum*5 sería:

- Evaluamos `sum` con 5:

```
((Y sum) 5)
```

- Evaluamos `Y` donde sustitumos f por (lambda(sum)):

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

Que se β-reduce:

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

como :
`(lambda(x)(lambda(sum)(xx)))(lambda(x)(lambda(sum)(xx))) = (lambda (Y) (lambda (sum)))` Por lo que tenemos:

```
((sum (Y sum)) 5)
```

Que se β-reduce:

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

Lo que nos deja lo siguiente:

```
(+ 5 ((Y sum) (- 5 1)))
(+ 5 ((Y sum) (4)))
```

en este punto, seguir la ejecución de: `(lambda (Y)(lambda (sum) (4)))` seguiremos exactamente los mismos pasos, lo que nos llevara a lo siguiente:

```
(+ 5 (+ 4 ((Y sum) 3)))
```

de nuevo, la ejecucion nos dejara lo siguiente:

```
(+ 5 (+ 4 (+ 3 ((Y sum) 2))))
```

siguiendo esa lógica, al final llegaremos a:

```
(+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (Y (sum 0)))))))
(+ 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)
```

El resultado en que da en racket es :

```
1 #lang racket
2 (define c #f)
3 (+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
4 (c 10)
```

```
Welcome to DrRacket, version 8.14 [cs].
Language: racket, with debugging; memory limit: 128 MB.
15
21
>
```

El primer resultado es 15 porque asignamos un let/cc el cuál una continuación que se va a guardar en k, y la $set!$ cambia el valor de c por el de k y como a k se le esta asignando 4 entonces tenemos:

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

Sin embrago como en la última línea le estamos asignando 10 a c, entonces en nuestro punto de continuación va a cambiar el valor de c y ya no se le va a asignar el valor de k, por lo que se tendrá:

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

Ahora la continuación asociada a evaluar usando la notación $\lambda \uparrow$ es:

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

Al evaluar tenemos primero $((\text{set!}\ \ c\ k)4)$ sería que $c = 4$, por lo que tendríamos la siguiente evaluación:

$$(\lambda\ \uparrow (v)(+\ 1(+\ 2(+\ 3(+\ v\ 5))))\ (c4)$$
$$(\lambda\ \uparrow (v)(+\ 1(+\ 2(+\ 3(+\ v\ 5))))\ (4)$$
$$(+\ 1(+\ 2(+\ 3(+\ 4\ 5)))) = 15$$

Como en la otra línea estamos definiendo a $c = 10$, entonces la evalución sería:

$$(\lambda\ \uparrow (v)(+\ 1(+\ 2(+\ 3(+\ v\ 5))))\ (c10)$$
$$(\lambda\ \uparrow (v)(+\ 1(+\ 2(+\ 3(+\ v\ 5))))\ (10)$$
$$(+\ 1(+\ 2(+\ 3(+\ 10\ 5)))) = 21$$

3. - 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. Por ejemplo:

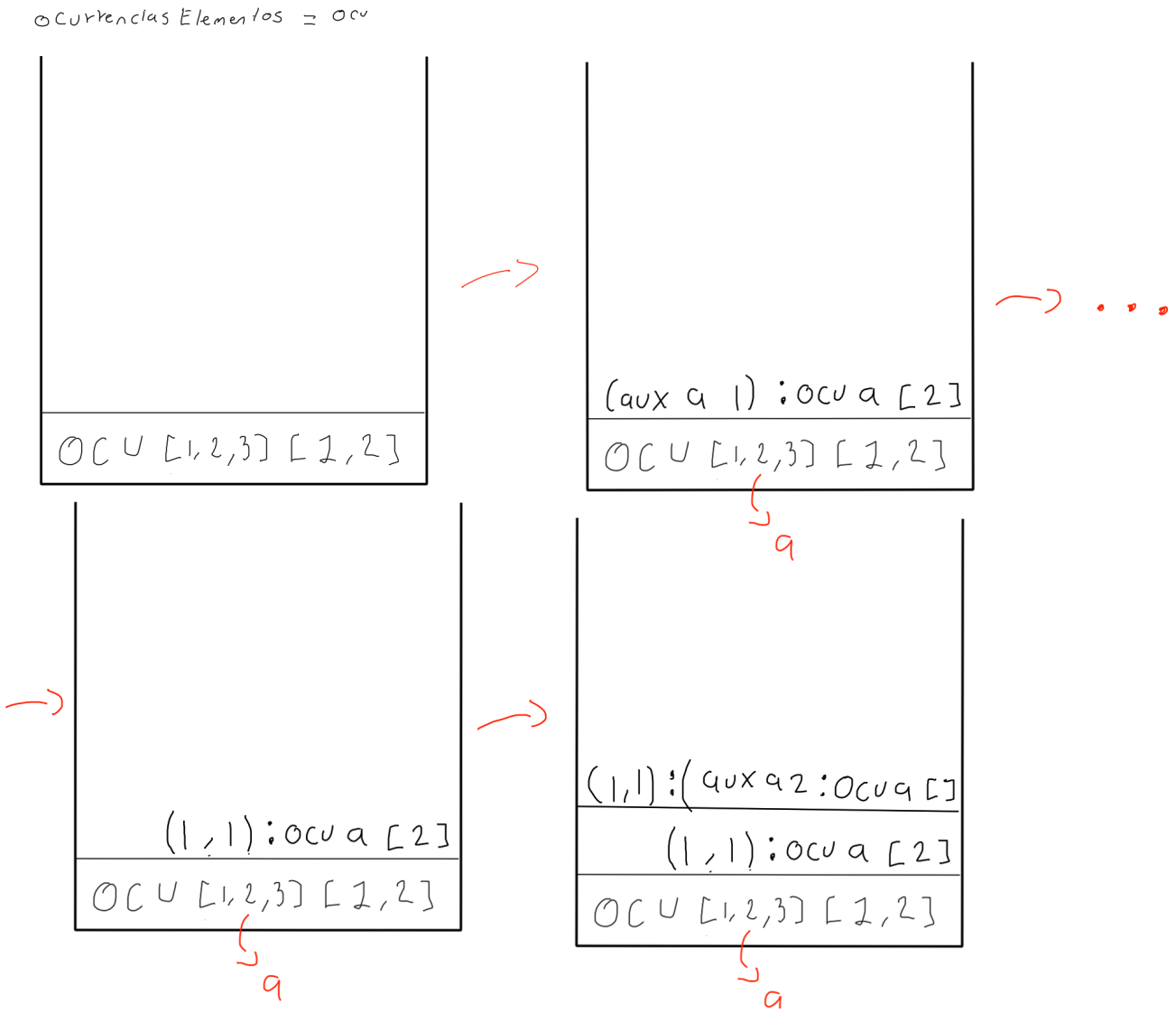
```
> ocurrenciasElementos [1,3,6,2,4,7,3,9,7] [5,2,3]
[(5,0),(2,1),(3,2)]
```

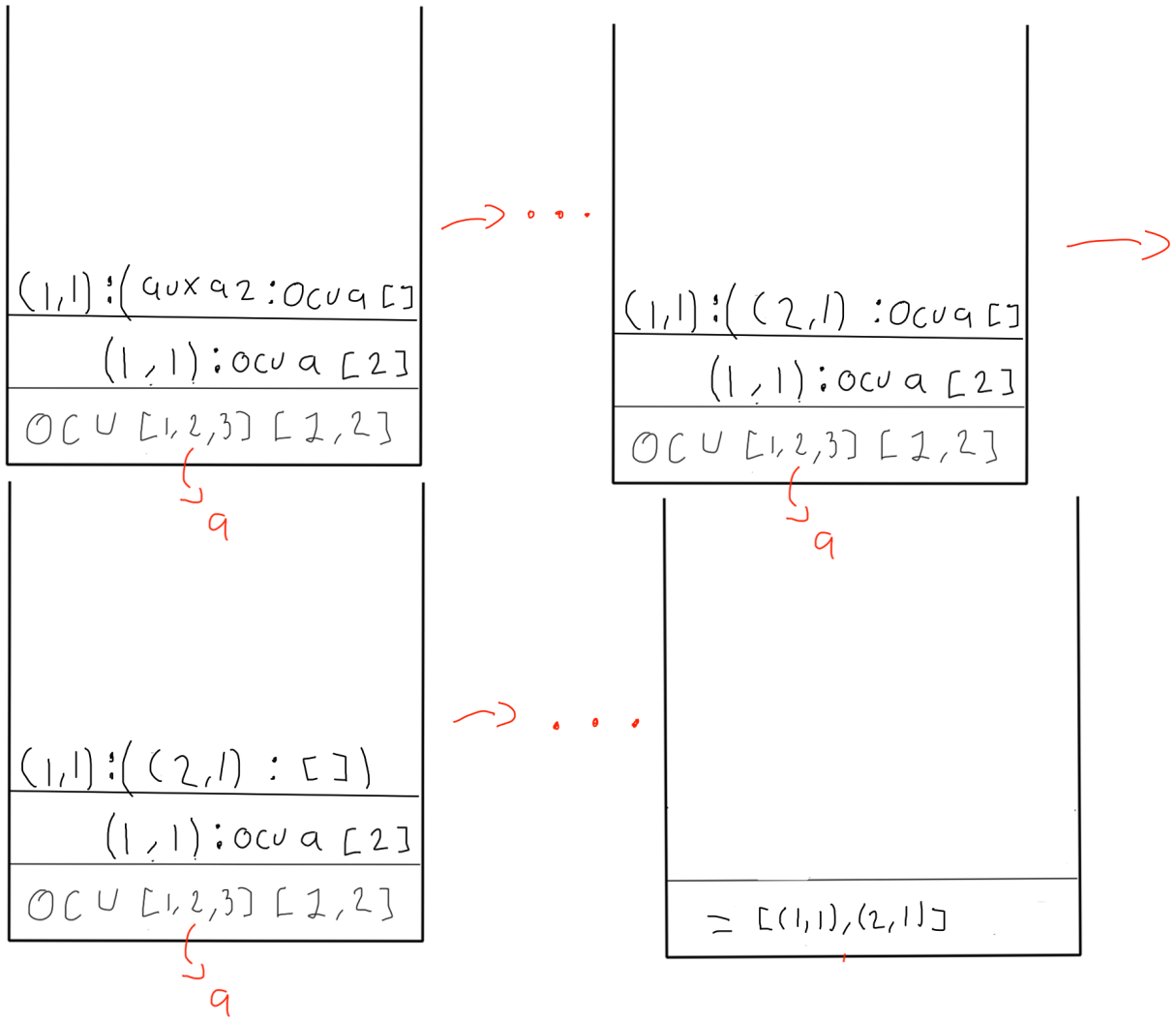
definimos la función de la siguiente forma:

```
ocurrenciasElementos :: [Int] -> [Int] -> [(Int,Int)]
ocurrenciasElementos a [] = []
ocurrenciasElementos a (x:xs) = (aux a x 0) : ocurrenciasElementos a xs


aux :: [Int] -> Int -> Int -> (Int,Int)
aux [] a n = (a, n)
aux (x:xs) a n = if a == x then (aux xs a (n+1)) else (aux xs a n)
```

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





se obviaron los pasos que no aumentaban el numero de registros como aux

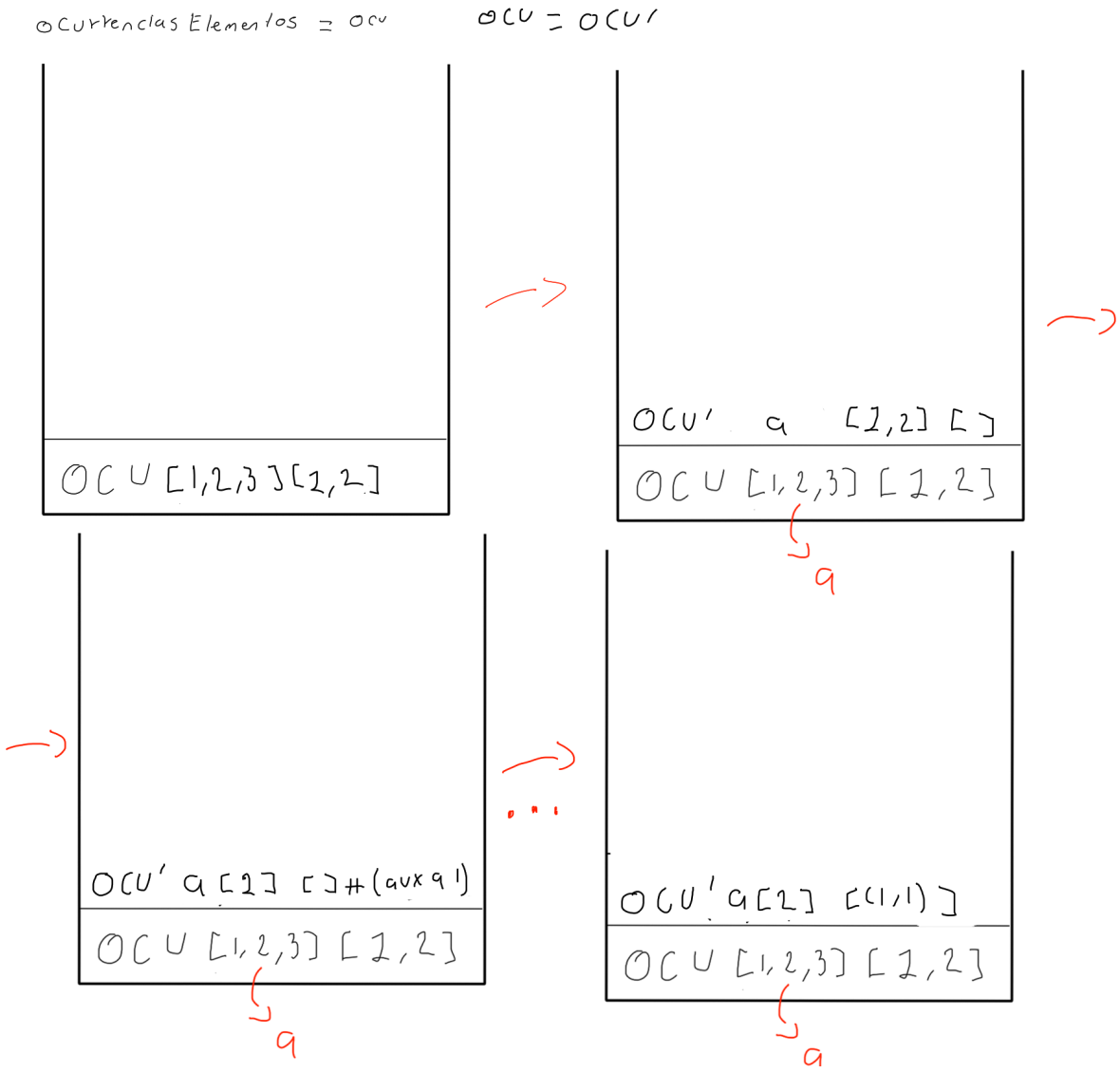
- Optimizar la función definida usando recursión de cola. Deben transformar todas las funciones auxiliares que utilicen.

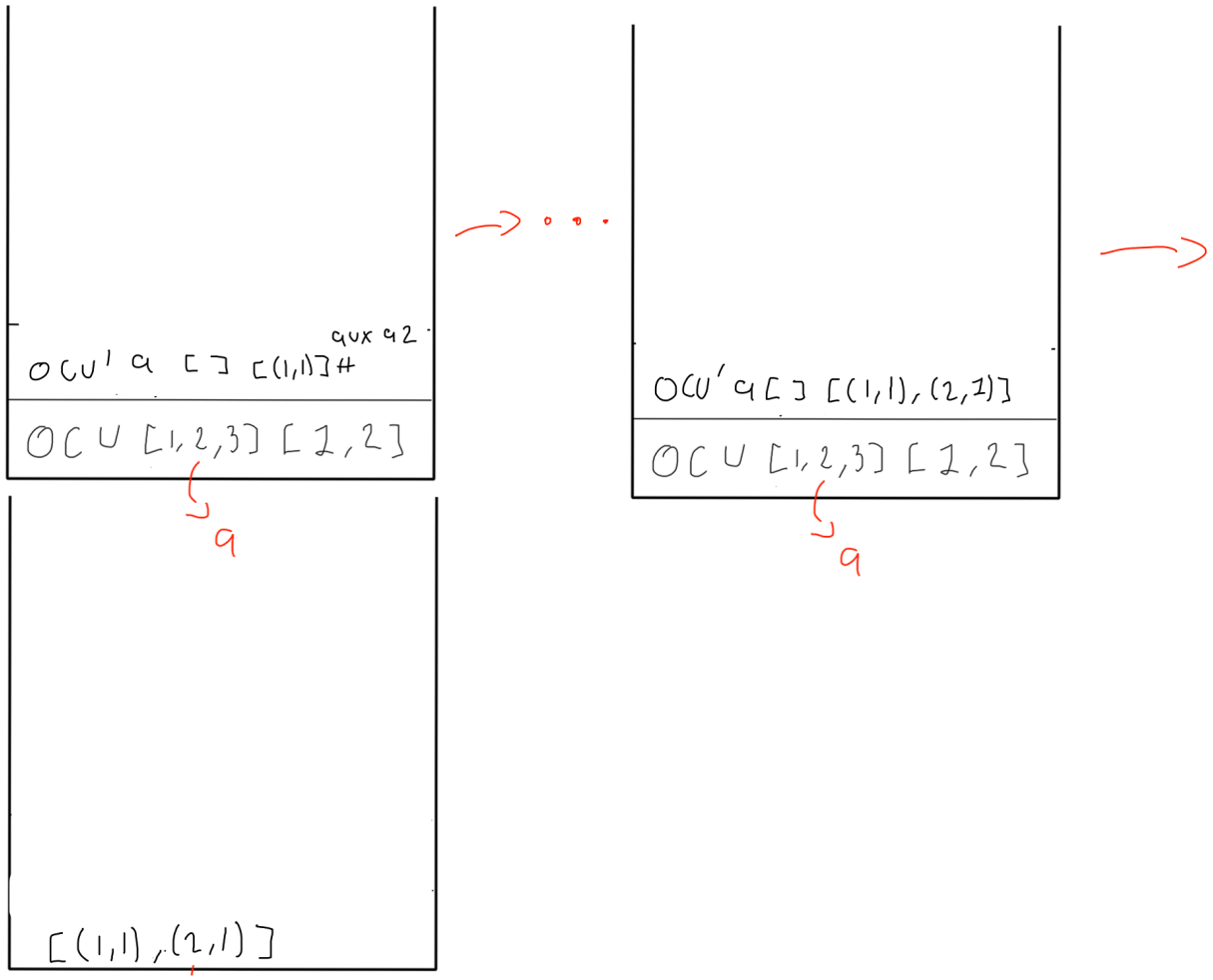
  como la función aux ya la definimos con recursión de cola, lo único que debemos modificar es la función ocurrenciasElementos para esto le agregaremos un acumulador en donde guardaremos las tuplas.

```
ocu :: [Int] -> [Int] -> [(Int,Int)] -> [(Int,Int)]
ocu a [] b = b
ocu a (x:xs) b = ocu a xs (b++[(aux a x 0)])

aux :: [Int] -> Int -> Int -> (Int,Int)
aux [] a n = (a, n)
aux (x:xs) a n = if a == x then (aux xs a (n+1)) else (aux xs a n)
```

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


OCurrenciasElementos = OCU
OCU = OCU'
OCU [1,2,3] [2,2]
OCU' a [],2] []
OCU [1,2,3] [2,2]
a
OCU' a [1] [] # (aux a 1)
OCU [1,2,3] [2,2]
a
OCU' a [2] [(1,1)]
OCU [1,2,3] [2,2]
a


aux q2
OCU' q [ ] [(1,1)]#
OC U [1,2,3] [1,2]
q
[(1,1),(2,1)]
OCU' q [ ] [(1,1),(2,1)]
OC U [1,2,3] [1,2]
q