Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación
2025-1
Semanal 7
Victoria Morales Ricardo Maximiliano
Sánchez Estrada Alejandro
Suárez Ortiz Joshua Daniel

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.

  Al ejecutarla nos topamos con un error ya que 'sum' es un variable libre
- Modificarla usando el combinador de punto fijo Y, volver a ejecutarla y explicar el resultado.

  Usamos la def de $Y_{def} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
  sum nos queda de la siguiente forma:

```
sum := lambda(n) : if0 n then 0 else (+ n (sum(n-1)))
```

```
(Y sum) 3 = def [(lambda f.(lambda x.f(xx))(lambda x.f(xx)) sum] 3
```

  Lo β-reducimos y nos queda:

$$\rightarrow_\beta (\lambda x.sum(xx))\ (\lambda x.sum(xx))\ 3)$$
$$\rightarrow_\beta (sum[(\lambda x.sum(xx))\ (\lambda x.sum(xx)]]\ 3\ (Y sum)$$

```
sum = def lambda(f) : lambda(n) if0 n then 0 else (+ n (f(n-1)))
lambda(f) : lambda(n) if0 n then 0 else (+ n (f(n-1))) 3
```

  β-reducimos:

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

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

  Checamos $if0$, si n es 0 ejecutamos el primer caso $n = 3$, no es 0 por lo tanto continuamos hasta que lo sea:

  $(sum\ 0) = (\lambda n.if0\ n\ 0\ (+n(sum(-\ n\ 1)))))0$ esto es sum(0) los anteriores ejecuciones quedaban como $1 + sum(0)$ y $2 + sum(1)$

  Evaluación de $(sum\ 5)$ comienza llamando a sum con el argumento 5. Como 5 no es 0, la función retorna 5 + sum(4). Este proceso continúa recursivamente hasta que n sea 0, momento en el que la función regresa 0 y se empiezan a sumar los valores. Por lo tanto la evaluación da como resultado $5 + 4 + 3 + 2 + 1 + 0 = 15$ o tambien se puede ver como $(+5(+4(+3(+2(+1(if0\ 0\ 0\ (+\ 0((Y\ Y)(-\ 0\ 1)))))))))) = (+5(+4(+3(+2(+\ 1\ 0))))) = 15$.

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)
```

Primero evaluaremos en Racket:

```
(define c #f)
```

Definimos la variable c como falso.
Seguimos con la expresion:

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

Empezamos desde lo mas interno de la expresion:

```
(let/cc k (set! c k) 4)
```

letcc continua con la suma de los valores y le es asignado a k y c, seguimos evaluando el cuerpo de la expresion.

```
(+ 1 (+ 2 (+ 3 (+ 4 5))))
```

Lo que es igual a 15, seguimos con:

```
(c 10)
```

Lo cual nos indica que invoca la continuacion que le habiamos asignado a c pero ahpra con 10 en lugar de 4, es decir tendriamos:

```
(+ 1 (+ 2 (+ 3 (+ 10 5))))
```

Como resultado obtendriamos 21.

Continuacion asociada $\lambda \uparrow$
$c = \lambda \uparrow (v) \cdot = (+1(+2(+3(+\ v\ 5))))$
$c(10) = (+1(+2(+3(+\ 10\ 5))))$
Lo que resulta en 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 derechael 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)]
     ```

     Función principal:

```haskell
ocurrenciasElementos :: (Eq a) => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos _ [] = []
ocurrenciasElementos xs (y:ys) = (y, cuentaOcurrencias y xs) :
    ocurrenciasElementos xs ys
```

Funcion auxiliar:

```haskell
cuentaOcurrencias :: (Eq a) => a -> [a] -> Int
cuentaOcurrencias _ [] = 0
cuentaOcurrencias elem (z:zs)
  | elem == z  = 1 + cuentaOcurrencias elem zs
  | otherwise  = cuentaOcurrencias elem zs
```

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

  Los registros de activación se ven a continuación:

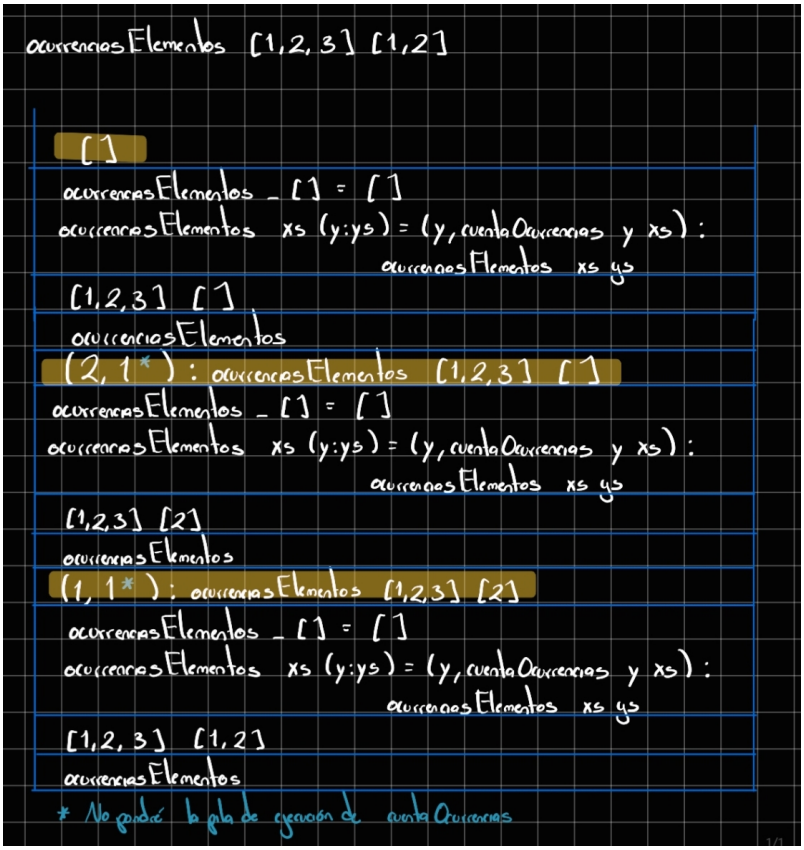

Figure 1: Registros de activación. Se omitió la pila de ejecución de cuentaOcurrencias.

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

  Función auxiliar:

```
cuentaOcurrencias :: (Eq a) => a -> [a] -> Int
cuentaOcurrencias a xs = colaCuentaOcurrencias a xs 0
```

```
colaCuentaOcurrencias :: (Eq a) => a -> [a] -> Int -> Int
colaCuentaOcurrencias _ [] acc = acc
cuentaOcurrencias elem (z:zs) acc
  | elem == z  = colaCuentaOcurrencias elem zs (1+acc)
  | otherwise  = cuentaOcurrencias elem zs acc
```

  Función principal:

```
ocurrenciasElementos :: (Eq a) => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos xs ys = colaOcurrenciasElementos xs ys []
```

```
colaOcurrenciasElementos :: (Eq a) => [a] -> [a] -> [(a, Int)] -> [(a, Int)]
colaOcurrenciasElementos _ [] acc = acc
colaOcurrenciasElementos xs (y:ys) acc = colaOcurrenciasElementos xs ys ((y,
    cuentaOcurrencias y xs): acc)
```

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

  Los registros de activación se ven a continuación:

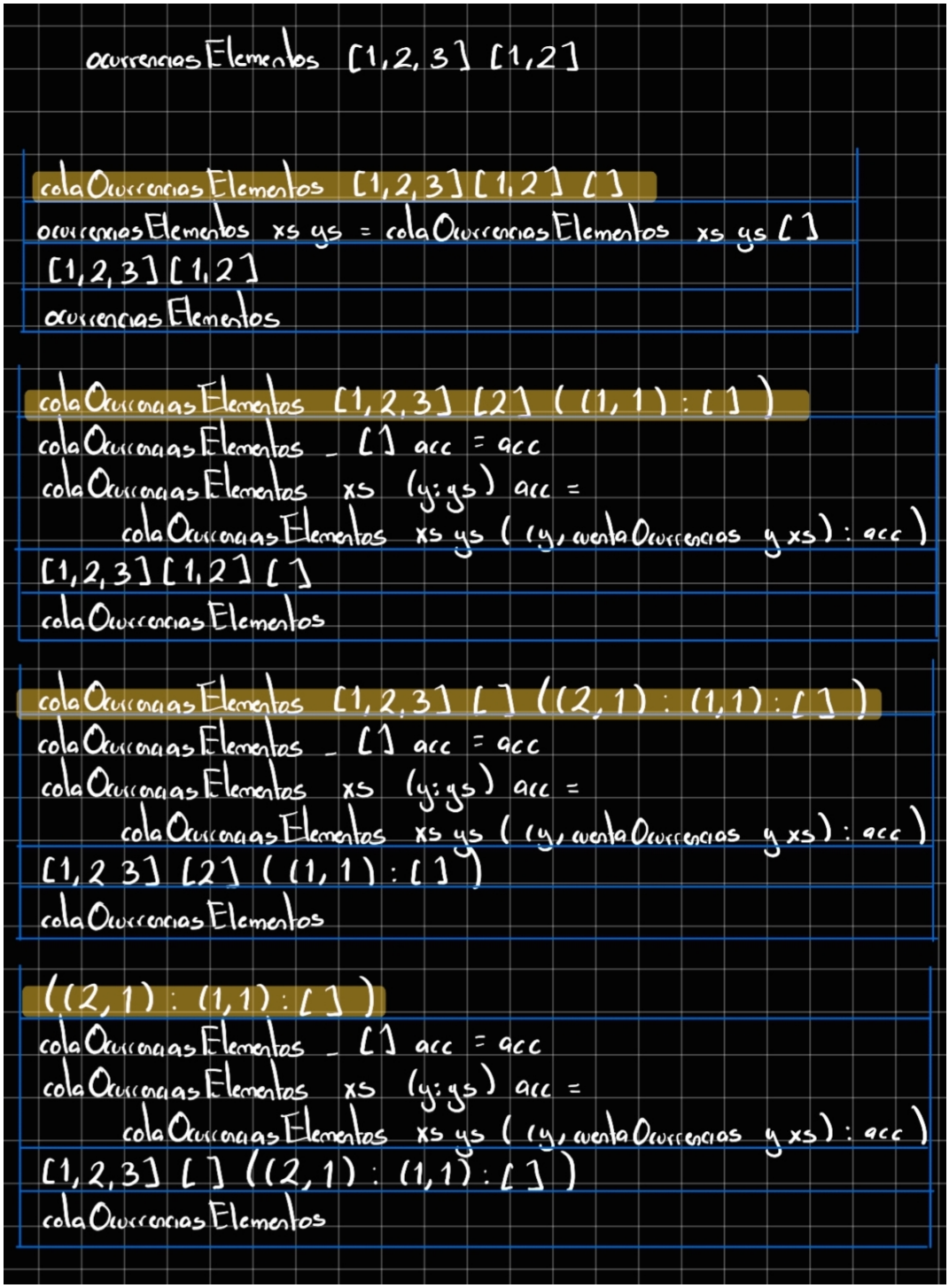

Figure 2: Registros de activación con las funciones que usan recursión de cola. Se omitió la pila de ejecución de cuentaOcurrencias. Son varias pilas, tenemos un registro de activación a la vez.