UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE CIENCIAS
LENGUAJES DE PROGRAMACIÓN

# Semanal 07

Salazar Gonzalez Pedro Yamil 306037445

## 1 Ejercicio 1

La expresión original define una función recursiva `sum` que calcula la suma de los $n$ primeros números:

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

1. **Inicio de la evaluación de `let`:**
   - Definimos `sum` como una función lambda que toma un argumento $n$.
   - La función `sum` tiene la siguiente forma:

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

2. **Primera llamada a `sum 5`:**
   - Evaluamos `(sum 5)`.
   - Sustituimos $n = 5$ en la función `sum`:

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

   - Dado que $5 \neq 0$, evaluamos la parte de la suma: `(+ 5 (sum 4))`.
3. **Evaluación de `sum 4`:**
   - Evaluamos `(sum 4)`.
   - Sustituimos $n = 4$ en la función `sum`:

     ```
     (if0 4 0 (+ 4 (sum (- 4 1))))
     ```

   - Dado que $4 \neq 0$, evaluamos `(+ 4 (sum 3))`.
4. **Evaluación de `sum 3`:**
   - Evaluamos `(sum 3)`.
   - Sustituimos $n = 3$ en la función `sum`:

     ```
     (if0 3 0 (+ 3 (sum (- 3 1))))
     ```

   - Dado que $3 \neq 0$, evaluamos `(+ 3 (sum 2))`.
5. **Evaluación de `sum 2`:**
   - Evaluamos `(sum 2)`.
   - Sustituimos $n = 2$ en la función `sum`:

     ```
     (if0 2 0 (+ 2 (sum (- 2 1))))
     ```

   - Dado que $2 \neq 0$, evaluamos `(+ 2 (sum 1))`.
6. **Evaluación de `sum 1`:**

- Evaluamos `(sum 1)`.
- Sustituimos $n = 1$ en la función `sum`:

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

- Dado que $1 \neq 0$, evaluamos `(+ 1 (sum 0))`.

7. **Evaluación de `sum 0`**:

   - Evaluamos `(sum 0)`.
   - Sustituimos $n = 0$ en la función `sum`:

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

   - Dado que $0 = 0$, devolvemos 0.

   con esto tendríamos una expresión (+ 5 ( + 4 ( + 3 ( + 2 ( +1 0)))))

8. **Volviendo hacia atrás en las llamadas recursivas**:

   - `sum 1` devuelve $1 + 0 = 1$.
   - `sum 2` devuelve $2 + 1 = 3$.
   - `sum 3` devuelve $3 + 3 = 6$.
   - `sum 4` devuelve $4 + 6 = 10$.
   - `sum 5` devuelve $5 + 10 = 15$.

Por lo tanto, el resultado de la suma es 15.

A continuación, se muestra la expresión modificada usando el combinador de punto fijo $Y$: tenemos que:

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

El combinador $Y$ nos permite definir funciones recursivas sin necesidad de hacer referencia directa a su nombre. Aquí:

- $Y$ toma una función $f$ y devuelve una versión recursiva de $f$.
- La función *sum* ahora se define en términos de un parámetro *sum*, que es la referencia recursiva proporcionada por $Y$.

El cálculo de `(sum 5)` sigue la misma lógica recursiva, y el resultado sigue siendo 15.

## 2 Ejercicio 2

La expresión es la siguiente:

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

La primera línea:

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

simplemente define la variable `c` con el valor inicial `#f`. Esto no tiene efecto inmediato en el resto de la evaluación.

La segunda parte de la expresión es una serie de sumas anidadas que contienen una llamada a `let/cc` para capturar una continuación. La expresión es:

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

Al llegar a `(let/cc k ...)`, la continuación capturada es lo que sigue después de la llamada, es decir, la evaluación de:

$$(+45)$$

Esta continuación es asignada a la variable `c` mediante `(set! c k)`. Después de capturar la continuación, la expresión retorna 4, por lo que el resto de la expresión se convierte en:

$$(+1(+2(+3(+45))))$$

Continuamos evaluando las sumas:

$$\begin{aligned}(+\ 4\ 5) &\rightarrow 9\\ (+\ 3\ 9) &\rightarrow 12\\ (+\ 2\ 12) &\rightarrow 14\\ (+\ 1\ 14) &\rightarrow 15\end{aligned}$$

Por lo tanto, la evaluación completa de esta expresión da como resultado **15**.

Ahora se invoca la continuación capturada almacenada en `c` con el valor 10:

```
(c 10)
```

Dado que `c` es la continuación capturada que se evalúa como `(+ 4 5)`, al invocar `(c 10)` reemplazamos la subexpresión capturada `(let/cc k ...)` por 10. La continuación es entonces:

$$(+1(+2(+3(+105))))$$

Ahora evaluamos la nueva expresión:

$$\begin{aligned}(+\ 10\ 5) &\rightarrow 15\\ (+\ 3\ 15) &\rightarrow 18\\ (+\ 2\ 18) &\rightarrow 20\\ (+\ 1\ 20) &\rightarrow 21\end{aligned}$$

El resultado final de invocar la continuación es **21**.

La continuación capturada por `let/cc k` es la parte de la expresión que aún queda por evaluarse al momento de la captura. En notación $\lambda^{\uparrow}$, la continuación es:

$$k = \lambda^{\uparrow} v.(+1(+2(+3(+v5))))$$

Esto significa que, cuando se invoca `c` con algún valor $v$, se reemplaza la expresión capturada (la que sigue después de `let/cc`) por el valor $v$, reanudando la evaluación desde ese punto.

El uso de continuaciones con `let/cc` en Racket permite capturar el estado de la ejecución en un punto dado y reanudarla con un nuevo valor. En este caso, la evaluación inicial da como resultado **15**, mientras que al invocar la continuación con `(c 10)`, la evaluación continúa desde el punto capturado, resultando en **21**.

# 3 Ejercicio 3

## 3.1 Ejercicio 1: Definición de la función `ocurrenciasElementos`

La función `ocurrenciasElementos` toma dos listas como argumentos y devuelve una lista de pares, donde cada par contiene un elemento de la segunda lista y el número de veces que aparece dicho elemento en la primera lista.

```
-- Funci n auxiliar para contar las ocurrencias de un elemento en una lista
contarOcurrencias :: Eq a => a -> [a] -> Int
contarOcurrencias x [] = 0
contarOcurrencias x (y:ys)
  | x == y    = 1 + contarOcurrencias x ys
```

```
  | otherwise = contarOcurrencias x ys

-- Funci n principal
ocurrenciasElementos :: Eq a => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos _ [] = []
ocurrenciasElementos lista (x:xs) = (x, contarOcurrencias x lista) : ocurrenciasElementos lis
```

Por ejemplo, al ejecutar:

```
-- Ejemplo de uso:
-- ocurrenciasElementos [1,3,6,2,4,7,3,9,7] [5,2,3]
-- [(5,0),(2,1),(3,2)]
```

### 3.2 Ejercicio 2: Registros de activación generados

Vamos a mostrar los registros de activación generados para la llamada `ocurrenciasElementos [1,2,3] [1,2]`.

- `ocurrenciasElementos [1,2,3] [1,2]`
- Se llama a `contarOcurrencias 1 [1,2,3]`:
  - `contarOcurrencias 1 [1,2,3]` → 1+ `contarOcurrencias 1 [2,3]` → 0
- Luego se llama a `contarOcurrencias 2 [1,2,3]`:
  - `contarOcurrencias 2 [1,2,3]` → `contarOcurrencias 2 [2,3]` → 1+ `contarOcurrencias 2 [3]` → 0
- El resultado es `[(1,1), (2,1)]`.

### 3.3 Ejercicio 3: Optimización con recursión de cola

Ahora optimizamos la función para usar recursión de cola. Transformamos la función auxiliar `contarOcurrencias` en una versión con acumulador, y adaptamos la función principal.

```
-- Funci n auxiliar optimizada para contar las ocurrencias usando recursi n de cola
contarOcurrenciasCola :: Eq a => a -> [a] -> Int
contarOcurrenciasCola x lista = contarAux x lista 0
  where
    contarAux _ [] acc = acc
    contarAux x (y:ys) acc
      | x == y    = contarAux x ys (acc + 1)
      | otherwise = contarAux x ys acc

-- Funci n principal optimizada
ocurrenciasElementosCola :: Eq a => [a] -> [a] -> [(a, Int)]
ocurrenciasElementosCola _ [] = []
ocurrenciasElementosCola lista (x:xs) =
  (x, contarOcurrenciasCola x lista) : ocurrenciasElementosCola lista xs
```

Por ejemplo, al ejecutar:

```
-- Ejemplo optimizado:
-- ocurrenciasElementosCola [1,3,6,2,4,7,3,9,7] [5,2,3]
-- [(5,0),(2,1),(3,2)]
```

### 3.4 Ejercicio 4: Registros de activación en la versión con recursión de cola

Mostramos los registros de activación para la versión optimizada con la misma llamada `ocurrenciasElementosCola [1,2,3] [1,2]`.

- `ocurrenciasElementosCola [1,2,3] [1,2]`
- Se llama a `contarOcurrenciasCola 1 [1,2,3]`:
  - `contarAux 1 [1,2,3] 0` → `contarAux 1 [2,3] 1` → `contarAux 1 [3] 1` → `contarAux 1 [] 1`
- Luego se llama a `contarOcurrenciasCola 2 [1,2,3]`:
  - `contarAux 2 [1,2,3] 0` → `contarAux 2 [2,3] 0` → `contarAux 2 [3] 1` → `contarAux 2 [] 1`
- El resultado es `[(1,1), (2,1)]`.

La función original `ocurrenciasElementos` fue transformada para utilizar recursión de cola, mejorando la eficiencia en el manejo de registros de activación. Ambas versiones producen los mismos resultados, pero la versión optimizada tiene un mejor manejo de los recursos.