UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Lenguajes de Programación

Semanal 07

Integrantes del Equipo:

- Eulogio Sánchez Christian
- López García Luis Norberto

Profesor: Manuel Soto Romero

16 de octubre de 2024

Ejercicio 1

a. Ejecutar la expresión y explicar el resultado

Se nos da la siguiente expresión en **MiniLisp**:

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

Al ejecutar `(sum 5)`, calculamos:

$$\begin{aligned}&\lambda n.if0\ n\ 0\ (+\ n\ sum(-\ n\ 1))5\\ &\quad = if0\ 5\ 0\ (+\ 5\ sum(-\ 5\ 1))\\ &\quad = (+\ 5\ sum\ 4)\end{aligned}$$

Pero `sum` queda como variable libre, entonces no se puede seguir ejecutando.

b. Modificarla usando el combinador de punto fijo Y, ejecutarla y explicar el resultado

El combinador de punto fijo Y nos permite definir funciones recursivas sin referenciarse a sí mismas directamente. En **MiniLisp**, podemos definir el combinador Y como:

```
(define Y (lambda (f) ((lambda (x) (f (x x))) (lambda(x) (f (x x))))))
```

Reescribimos la función `sum` usando el combinador Y:

```
(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))
```

Esta expresión define una función anónima que se aplica a sí misma para lograr la recursión necesaria.

Ejercicio 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)
```

Nos da como resultado 15 y 21, esto es debido a que primero se guarda el valor de k como 4, entonces se realiza

```
(+ 1 (+ 2 (+ 3 (+ 4 5)))) = 15
```

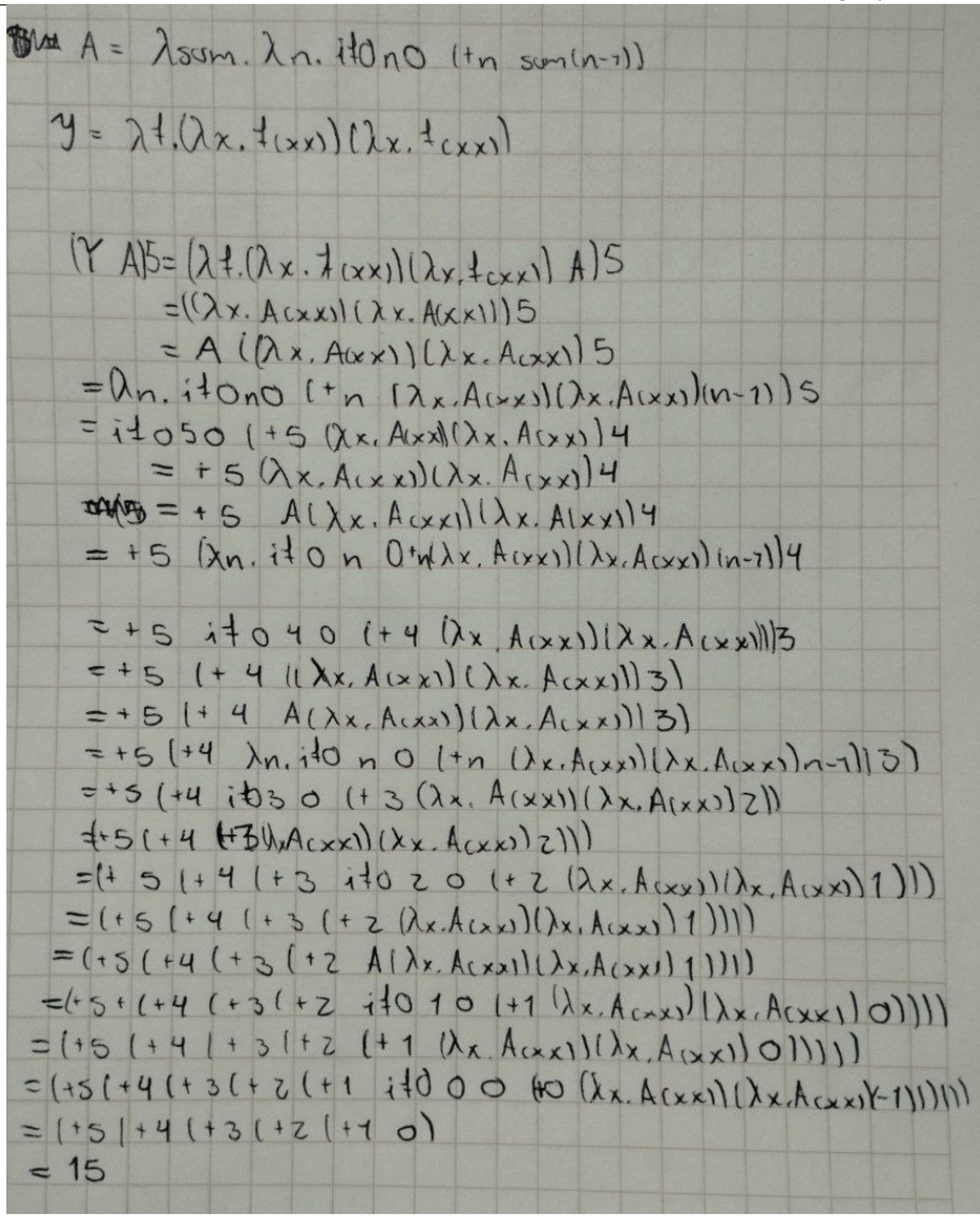
Sin embargo, en la continuación del `let/cc` tenemos `c 10`, por lo tanto en vez de usar 4 se usa 10, evaluando:

```
(+ 1 (+ 2 (+ 3 (+ 10 5)))) = 21
```

La continuación asociada es:

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

    anterior: (+ v 5)
    actual: v
    siguiente: (c 10)
```



A = λsum. λn. if0 n 0 (+n sum(n-1))

Y = λf.(λx. f(xx))(λx. f(xx))

(Y A)5 = (λf.(λx. f(xx))(λx. f(xx)) A)5

=((λx. A(xx))(λx. A(xx)))5

= A ((λx. A(xx))(λx. A(xx)))5

=λn. if0 n 0 (+n ((λx. A(xx))(λx. A(xx)))(n-1))5

= if0 5 0 (+5 (λx. A(xx))(λx. A(xx))4

= +5 (λx. A(xx))(λx. A(xx))4

= +5 A(λx. A(xx))(λx. A(xx))4

= +5 (λn. if0 n 0 (+n (λx. A(xx))(λx. A(xx))(n-1))4

= +5 if0 4 0 (+4 (λx. A(xx))(λx. A(xx)))3

= +5 (+4 ((λx. A(xx))(λx. A(xx)))3)

= +5 (+4 A(λx. A(xx))(λx. A(xx)))3)

= +5 (+4 λn. if0 n 0 (+n (λx. A(xx))(λx. A(xx))n-1))3)

= +5 (+4 if0 3 0 (+3 (λx. A(xx))(λx. A(xx))2))

= +5 (+4 (+3 (λx. A(xx))(λx. A(xx))2)))

=(+5 (+4 (+3 if0 2 0 (+2 (λx. A(xx))(λx. A(xx))1)))

=(+5 (+4 (+3 (+2 (λx. A(xx))(λx. A(xx))1))))

=(+5 (+4 (+3 (+2 A(λx. A(xx))(λx. A(xx))1))))

=(+5 (+4 (+3 (+2 if0 1 0 (+1 (λx. A(xx))(λx. A(xx))0)))))

=(+5 (+4 (+3 (+2 (+1 (λx. A(xx))(λx. A(xx))0)))))

=(+5 (+4 (+3 (+2 (+1 if0 0 0 (+0 (λx. A(xx))(λx. A(xx))(-1)))))))

=(+5 (+4 (+3 (+2 (+1 0)

= 15

Figura 1: Aplicación con combinador Y.

Ejercicio 3

a. Definir la función recursiva `ocurrenciasElementos`

Definimos la función `ocurrenciasElementos` en **Haskell** de la siguiente manera:

```
ocurrenciasElementos :: (Eq a) => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos xs ys = [(y, cuenta y xs) | y <- ys]
  where
    cuenta :: (Eq a) => a -> [a] -> Int
```

```
cuenta _ [] = 0
cuenta y (x:xs)
  | y == x    = 1 + cuenta y xs
  | otherwise = cuenta y xs
```

Esta función toma dos listas: la primera con los elementos donde buscar (`xs`) y la segunda con los elementos a contar (`ys`). Para cada elemento `y` en `ys`, calcula cuántas veces aparece en `xs` usando la función auxiliar `cuenta`.

b. Mostrar los registros de activación generados por la función con la llamada `ocurrenciasElementos [1,2,3] [1,2]`

Llamada:

```
ocurrenciasElementos [1,2,3] [1,2]
```

Análisis de la ejecución:

1. **Primera iteración ($y = 1$):**
   - `cuenta 1 [1,2,3]`
     - $1 == 1$: suma 1 y continúa con `cuenta 1 [2,3]` (acumulado: 1)
     - $1 \neq 2$: continúa con `cuenta 1 [3]` (acumulado: 1)
     - $1 \neq 3$: continúa con `cuenta 1 []` (acumulado: 1)
     - Lista vacía: retorna 0
   - Retorna total: $1 + 0 = 1$
2. **Segunda iteración ($y = 2$):**
   - `cuenta 2 [1,2,3]`
     - $2 \neq 1$: continúa con `cuenta 2 [2,3]` (acumulado: 0)
     - $2 == 2$: suma 1 y continúa con `cuenta 2 [3]` (acumulado: 1)
     - $2 \neq 3$: continúa con `cuenta 2 []` (acumulado: 1)
     - Lista vacía: retorna 0
   - Retorna total: $1 + 0 = 1$

Registros de activación:

- `ocurrenciasElementos [1,2,3] [1,2]`
  - `cuenta 1 [1,2,3]`
    - `cuenta 1 [2,3]`
      - `cuenta 1 [3]`
      - `cuenta 1 []`
  - `cuenta 2 [1,2,3]`
    - `cuenta 2 [2,3]`
      - `cuenta 2 [3]`
      - `cuenta 2 []`

c. Optimizar la función usando recursión de cola

```
cuenta :: (Eq a) => a -> [a] -> Int
cuenta y xs = cuentaAux y xs 0
  where
    cuentaAux :: (Eq a) => a -> [a] -> Int -> Int
    cuentaAux _ [] acc = acc
    cuentaAux y (x:xs) acc
      | y == x    = cuentaAux y xs (acc + 1)
      | otherwise = cuentaAux y xs acc
```

Actualizamos `ocurrenciasElementos`:

```
ocurrenciasElementos :: (Eq a) => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos xs ys = [(y, cuenta y xs) | y <- ys]
```

Ahora, `cuenta` utiliza un acumulador (`acc`) y es recursiva de cola.

d. Mostrar los registros de activación generados por la versión de cola

Con la llamada `ocurrenciasElementos [1,2,3] [1,2]`, los registros de activación son:

- `ocurrenciasElementos [1,2,3] [1,2]`
  - `cuenta 1 [1,2,3]` inicia con `acc = 0`
    - `cuentaAux 1 [1,2,3] 0`
      - $1 == 1$: `acc = 1`, continúa con `cuentaAux 1 [2,3] 1`
      - $1 \neq 2$: `acc = 1`, continúa con `cuentaAux 1 [3] 1`
      - $1 \neq 3$: `acc = 1`, continúa con `cuentaAux 1 [] 1`
      - Lista vacía: retorna `acc = 1`
  - `cuenta 2 [1,2,3]` inicia con `acc = 0`
    - `cuentaAux 2 [1,2,3] 0`
      - $2 \neq 1$: `acc = 0`, continúa con `cuentaAux 2 [2,3] 0`
      - $2 == 2$: `acc = 1`, continúa con `cuentaAux 2 [3] 1`
      - $2 \neq 3$: `acc = 1`, continúa con `cuentaAux 2 [] 1`
      - Lista vacía: retorna `acc = 1`

Cada llamada sólo se realiza una vez.