UNIVERSIDAD NACIONAL AUTÓNOMA DE
MÉXICO
FACULTAD DE CIENCIAS
LENGUAJES DE PROGRAMACIÓN

Tarea 07

Condiciones de entrega:

1. La tarea se realizará en equipos de 3
   - Castillo Chora Paola
   - Ángel Moises González Corrales
2. Se debe entregar en un pdf generado con LaTeX

Ejercicio 1

La expresión en MiniLisp es la siguiente:

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

Resultado al ejecutar

1. **sum**: Se define una función llamada `sum` con `let`. Esta función recibe un número `n`. Si `n` es 0, regresa 0. Si no, suma `n` con el resultado de llamar a `sum` con `n-1`.
2. **Evaluación de** `(sum 5)`:

$$\begin{aligned}
\texttt{sum(5)} &= 5 + \texttt{sum(4)} \\
\texttt{sum(4)} &= 4 + \texttt{sum(3)} \\
\texttt{sum(3)} &= 3 + \texttt{sum(2)} \\
\texttt{sum(2)} &= 2 + \texttt{sum(1)} \\
\texttt{sum(1)} &= 1 + \texttt{sum(0)} \\
\texttt{sum(0)} &= 0
\end{aligned}$$

Y ahora se resuelve hacia atrás:

$$\begin{aligned}
\texttt{sum(1)} &= 1 + 0 = 1 \\
\texttt{sum(2)} &= 2 + 1 = 3 \\
\texttt{sum(3)} &= 3 + 3 = 6 \\
\texttt{sum(4)} &= 4 + 6 = 10 \\
\texttt{sum(5)} &= 5 + 10 = 15
\end{aligned}$$

El resultado final es **15**.

La versión modificada con el combinador de punto fijo Y

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

Caracteristicas

1. **Combinador Y**: Permite hacer recursión sin nombres directos. Se pasa una función a `Y`, y él la convierte en recursiva.

2. **Aplicamos Y a `sum`**: Usamos `Y` para definir la versión recursiva de `sum`.

3. **Evaluamos `((Y ...) 5)`**:

$$\begin{aligned}
\texttt{sum(5)} &= 5 + \texttt{sum(4)} \\
\texttt{sum(4)} &= 4 + \texttt{sum(3)} \\
\texttt{sum(3)} &= 3 + \texttt{sum(2)} \\
\texttt{sum(2)} &= 2 + \texttt{sum(1)} \\
\texttt{sum(1)} &= 1 + \texttt{sum(0)} \\
\texttt{sum(0)} &= 0
\end{aligned}$$

   Y el resultado final también es **15**.

Ejercicio 2

Evaluar la siguiente expresión en Racket, explicar su resultado y dar la continuación asociada a evaluar usando la notación $\lambda\uparrow$.

```
1 -- Esta funcion cuenta cuantas veces aparece un elemento en la lista
2 (define c #f)
3 (+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
4 (c 10)
```

La primera parte de la expresión devuelve 15, pues evalua la suma aninada de $1+2+3+4+5$, mientras la segunda parte de la linea 4 devuelve 21, pues remplaza el valor de 4 con el de 10 en la misma suma, reanudando la evaluación.
La continuación $\lambda\uparrow$ asociada es

$$(\lambda\uparrow(\text{v})\ (+\ 1\ (+\ 2\ (+\ 3\ (+\ \text{v}\ 5))))))$$

En la primera linea, corresponde a la siguiente evaluación.

$$\begin{gathered}
(\lambda\uparrow(\text{v})\ (+\ 1\ (+\ 2\ (+\ 3\ (+\ \text{v}\ 5))))))\ (\text{c}\ 10) \\
(+\ 1\ (+\ 2\ (+\ 3\ (+\ 10\ 5)))))) \\
21
\end{gathered}$$

En el segundo bloque, se evalua como sigue.

$$\begin{gathered}
(\lambda\uparrow(\text{v})\ (+\ 1\ (+\ 2\ (+\ 3\ (+\ \text{v}\ 5))))))\ (\text{c}\ 4) \\
(+\ 1\ (+\ 2\ (+\ 3\ (+\ 4\ 5)))))) \\
15
\end{gathered}$$

Ejericicio 3

Parte 1: Definición de la función `ocurrenciasElementos`

La función `ocurrenciasElementos` toma dos listas como argumentos. La primera lista contiene los elementos de los que se contarán las ocurrencias, y la segunda lista contiene los elementos que queremos contar. La función devuelve una lista de pares, donde cada par contiene el elemento de la segunda lista y la cantidad de veces que aparece en la primera.

```haskell
-- Esta funcion cuenta cuantas veces aparece un elemento en la lista
contarOcurrencias :: Eq a => a -> [a] -> Int
contarOcurrencias x = length . filter (== x)

-- Funcion principal que arma la lista de pares
ocurrenciasElementos :: Eq a => [a] -> [a] -> [(a, Int)]
ocurrenciasElementos lista1 lista2 = [(y, contarOcurrencias y lista1) | y <-
    lista2]
```

Ejemplo de uso:

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

Parte 2: Registros de Activación

Cuando llamamos `ocurrenciasElementos [1,3,6,2,4,7,3,9,7] [5,2,3]`, estos son los pasos que se ejecutan:

1. Se llama la función `ocurrenciasElementos` con las listas `[1,3,6,2,4,7,3,9,7]` y `[5,2,3]`.
2. Para el primer elemento de la segunda lista (`5`):
   - Se filtran los elementos de `[1,3,6,2,4,7,3,9,7]` que sean `5`, te queda `[]`.
   - El resultado es `0`, entonces se arma el par `(5,0)`.
3. Para el segundo elemento de la segunda lista (`2`):
   - Se filtran los `2` en `[1,3,6,2,4,7,3,9,7]`, te queda `[2]`.
   - El resultado es `1`, entonces se genera el par `(2,1)`.
4. Para el tercer elemento de la segunda lista (`3`):
   - Se filtran los `3` en `[1,3,6,2,4,7,3,9,7]`, te queda `[3,3]`.
   - El resultado es `2`, entonces se genera el par `(3,2)`.
5. El resultado final es `[(5,0),(2,1),(3,2)]`.

Parte 3: Optimización con Recursión de Cola

Para optimizar la función y usar recursión de cola, creamos una versión que acumule los resultados a medida que se procesan los elementos.

```haskell
-- Funcion para contar ocurrencias, pero con recursion de cola
contarOcurrenciasTail :: Eq a => a -> [a] -> Int
contarOcurrenciasTail x = contar 0
  where
    contar acc [] = acc
    contar acc (y:ys)
      | x == y    = contar (acc + 1) ys
      | otherwise = contar acc ys

-- Version de ocurrenciasElementos
ocurrenciasElementosTail :: Eq a => [a] -> [a] -> [(a, Int)]
ocurrenciasElementosTail lista1 lista2 = ocurrenciasAux lista2 []
  where
    ocurrenciasAux [] acc = reverse acc
    ocurrenciasAux (y:ys) acc = ocurrenciasAux ys ((y, contarOcurrenciasTail
        y lista1) : acc)
```

Ejemplo de uso:

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

Parte 4: Registros de Activación por la version de cola

Con la llamada `ocurrenciasElementosTail [1,3,6,2,4,7,3,9,7] [5,2,3]`:

1. Empiezas con la llamada a `ocurrenciasElementosTail` y un acumulador vacío `[]`.
2. Para el primer elemento (`5`):
   - Se cuenta cuántas veces aparece el `5` en `[1,3,6,2,4,7,3,9,7]`.
   - Como no hay ningún `5`, se genera `(5,0)` y se guarda en el acumulador.
3. Para el segundo elemento (`2`):
   - Se cuenta cuántas veces aparece el `2` en `[1,3,6,2,4,7,3,9,7]`.
   - Se genera `(2,1)` y se guarda en el acumulador.
4. Para el tercer elemento (`3`):
   - Se cuenta cuántas veces aparece el `3` en `[1,3,6,2,4,7,3,9,7]`.
   - Se genera `(3,2)` y se guarda en el acumulador.
5. El resultado final es `[(5,0),(2,1),(3,2)]`.