Ayudantía 10:
Mas Scheme!

Profesores: José Luis Martí Lara, Roberto Díaz Urra
Ayudantes:
Hugo Sepúlveda Arriaza
Gabriela Acuña Benito
Lucio Fondón Rebolledo
lucio.fondon@sansano.usm.cl

Universidad Técnica Federico Santa María
Departamento de Informática

Definición

Definición

La *recursividad* es un concepto y técnica fundamental dentro del mundo de la programación. Se define la **recursión** como una manera de resolver un problema con un procedimiento (*función*) que se llama a sí mismo. En general, en una solución recursiva se pueden encontrar dos componentes principales:

- **Caso Base**: Es la solución a la instancia más pequeña del problema. Sirve como condición de término para el algoritmo
- **Paso Recursivo**: Llamada a la misma función, pero con el problema un poco menos complejo que el original.

Función `factorial()` **en** `Scheme`

```
(define (factorial n)
  (if (= n 0) ;; Caso Base
      1
      (* n (factorial (- n 1)))) ;; Paso Recursivo
  )
```

Recursión Simple vs Recursión de Cola

- **Recursión Simple**: Una función utiliza **recursión simple** si quedan instrucciones por ejecutar al retornar
- **Recursión de Cola**: Una función utiliza **recursión simple** si no ejecuta ninguna instrucción luego de retornar

Función `factorial()` con recursión simple

```
(define (factorial-simple n)
  (if (= n 0) ;; Caso Base
      1
      (* n (factorial (- n 1)))) ;; Paso Recursivo
  )
```

Función `factorial()` con recursión de cola

```
(define (factorial-cola numero)
  (let rec ((n numero) (cant 1))
    (if (= n 0) ;; Caso Base
        cant
        (rec (- n 1) (* cant n)) ;; Paso Recursivo
        )
    )
  )
```

Recursión Simple vs Recursión de Cola

Stack de llamadas de `factorial()` con recursión simple

```
(factorial-simple 3)
(* 3 (factorial-simple 2))
(* 3 (* 2 (factorial-simple 1)))
(* 3 (* 2 (* 1 (factorial-simple 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Stack de llamadas de `factorial()` con recursión de cola

```
(factorial-cola 3)
(factorial-cola 3 1)
(factorial-cola 2 3)
(factorial-cola 1 6)
(factorial-cola 0 6)
6
```

Ejercicios

Ejercicio 1

Dada una lista de números, debe realizar una función que calcule la suma total de los cubos de cada uno de los elementos de la lista utilizando recursión de cola. En caso que la lista se encuentre vacía, se debe devolver el valor 0.

```
>(sumar-cubos) '(5 6 4 10 6))
1432
>(sumar-cubos) '(9 3 4 2 1))
829
```

Solución

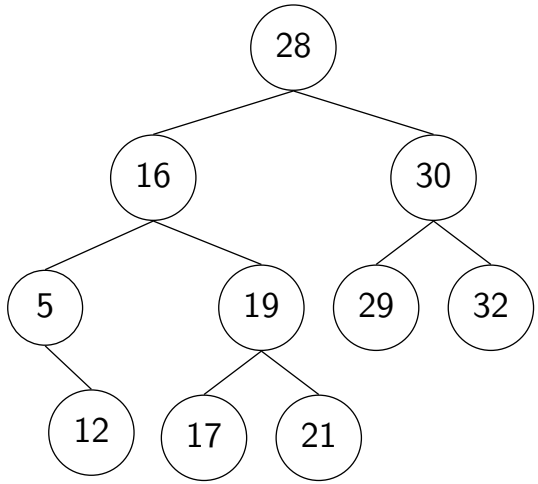
```
(define (cubo n)
  (* (* n n) n))

(define (sumar-cubos lista)
  (let rec ( (l lista) (suma 0) )
    (if (null? l)
        suma
        (rec (cdr l) (+ suma (cubo (car l))) )
        )
    )
  )
```

Ejercicio 2

Un árbol binario puede ser representado por una lista mediante:
`'(valor-nodo árbol-izquierdo árbol-derecho)`
Por lo tanto, una hoja sería un nodo con dos hijos nulos:
`'(valor-nodo () ())`
Teniendo en cuenta lo anterior, realice una función que recorra un árbol binario de búsqueda (ABB) en recorrido *InOrder*

que retorne una lista con el orden de visita de los nodos




```
>(define arbol '( 28 (16 (5 () (12 () ())) (19 (17 () ()) (21 () ()))) (30 (29 () ()) (32 () ()))))
>(in-order arbol)
(5 12 16 17 19 21 28 30 32)
```

Solución

```
;; valor del nodo
(define (valor nodo)
  (if (null? nodo)
      '()
      (car nodo)
      )
  )

;; arbol izq
(define (izq nodo)
  (if (null? nodo)
      '()
      (car(cdr nodo))
      )
  )

;; arbol der
(define (der nodo)
  (if (null? nodo)
      '()
      (car(cdr(cdr nodo)))
      )
  )

(define (in-order arbol)
  (if (null? arbol)
      '()
       (append(in-order (izq arbol))
              (list (valor arbol))
              (in-order (der arbol)))
       )
  )
```

Ejercicio 3

Utilizando recursión de cola, desarrolle la función `cortar` que recibe una lista con números enteros y un número entero `x`. La función debe retornar una lista que contenga tres listas: una lista con los elementos menores a `x`, una lista con los elementos iguales a `x` y una lista con los elementos mayores a `x`. En cada lista se debe mantener el orden de los elementos en la lista original.

```
>(cortar  6 '(1 4 2 6 4 8 6 2 1 7 4 5 1))
(1 4 2 4 2 1 4 5 1) (6 6) (8 7))
```

Solución

```
(define (cortar n lista)
  (let rec ( (l lista) (igual '()) (mayor '()) (menor '()) )
    (if (null? l)
        (append (list menor) (append (list igual) (list mayor)))
        (cond
          ((= (car l) n) (rec (cdr l) (append igual (list n)) mayor menor))
          ((> (car l) n) (rec (cdr l) igual (append mayor (list (car l))) menor))
          (else (rec (cdr l) igual mayor (append menor (list (car l)))))
        )
      )
    )
  )
```