Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

•••

Karla Ramírez Pulido

Recursión de Cola
y
Procedimientos Mutuamente Recursivos

Función factorial

Implementación con recursión y con recursión de cola

Introducción

Recursión de cola ---> Optimización de la recursión

¿Dónde?

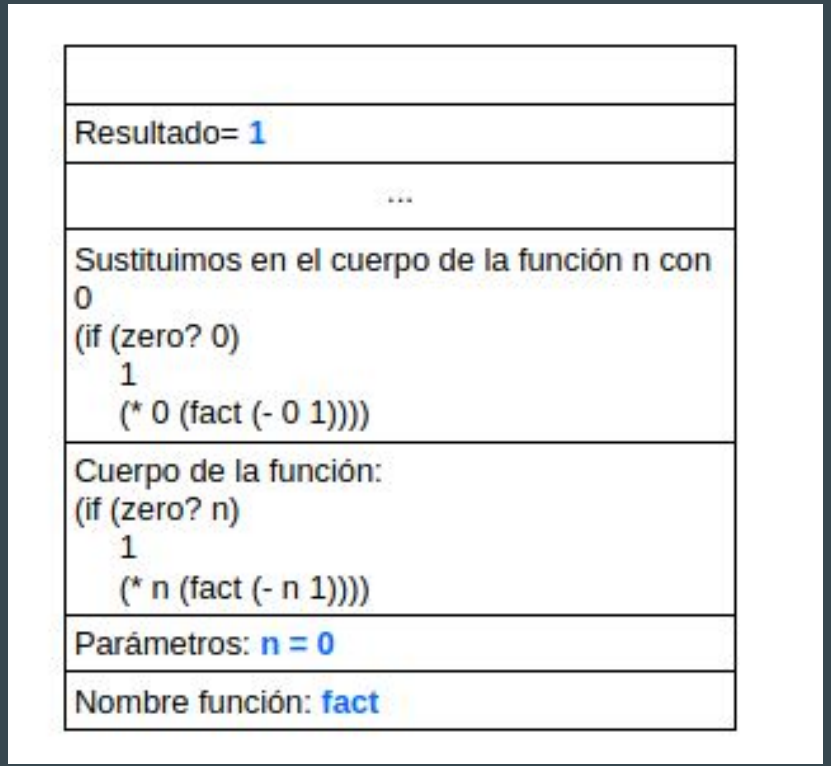
En manejo de espacio.

Recuerdan la función factorial ¿cuántos registros de activación se crean en tiempo de ejecución con una instancia de 0, 1, 2, 1,000,000, etc?

Ejemplo: función factorial

```
(define (fact n)
        (if (zero? n)
                1
            (* n (fact (- n 1)))))


> (fact 0)
```



Función factorial

> (fact 1)

Se crean 2 registros de activación de la llamada a función principal:

(fact 1) => (* 1 (fact 0)) => (* 1 1) = 1

> (fact 2)

Se crean 3 registros de activación de la llamada a función principal:

(fact 2) => (* 2 (fact 1)) => (* 2 (* 1 (fact 0)) => (* 2 (* 1 1)) = (* 2 1) = 2

¿Como se ve el ambiente con factorial de 5?

Veamos primero, factorial de 5 i.e. (fac 5) es:

(fact 5)

= (* 5 (fact 4))

= (* 5 (* 4 (fact 3)))

= (* 5 (* 4 (* 3 (fact 2))))

= (* 5 (* 4 (* 3 (* 2 (fact 1)))))

= (* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))

= (* 5 (* 4 (* 3 (* 2 (* 1 1)))))

= (* 5 (* 4 (* 3 (* 2 1))))

= (* 5 (* 4 (* 3 2)))

= (* 5 (* 4 6))

= (* 5 24)

= 120

Funciones con Recursión de cola (tail recursion)

Es la capacidad de tener funciones dentro de las mismas funciones.

Acumulador: resultados parciales de las llamadas recursivas.

Recursión

```
(define (fact n)
        (if (zero? n)
                1
                (* n (fact (- n 1)))))
```

Recursión de cola

```
(define (fact n)
        (fac-cola n 1)))


(define (fac-cola n acc)
        (if (zero? n)
                acc
                (fac-cola (- n 1) (* n acc))))
```

Recursión de cola

>(fact 5)

= (fac-cola 5 acc) i.e.

(fac-cola 5 1)

Inicialmente tenemos 1 registro de activación cuyo Resultado es la generación de una nueva llamada a fac-cola

```
(define (fact n)

        (fac-cola   n   1)))
```

```
;;La firma de la función fac-cola recibe un número n y un acc que
;;se inicializa en 1

(define (fac-cola   n   acc)

        (if (zero? n)

                acc

                (fac-cola (- n 1) (* n acc))))
```

Recursión de cola

(fac-cola 5 1)

genera como resultado una llamada a función:

= (fac-cola (- 5 1) (* 5 1)) i.e.

(fac-cola 4 5)

```
(define (fac-cola n acc)
    (if (zero? n)
        acc
        (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 5 1) es la expr. del else =>

(fac-cola (- 5 1) (* 5 1))

En este punto ya terminó la llamada de (fac-cola 5 1) para dar como resultado (fac-cola 4 5)

Recursión de cola

(fac-cola 4 5)

= (fac-cola (- 4 1) (* 4 5)) i.e.

(fac-cola 3 20)

```
(define (fac-cola n acc)
    (if (zero? n)
        acc
        (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 4 5) es la expr. del else =>

(fac-cola (- 4 1) (* 4 5))

En este punto ya terminó la llamada de

(fac-cola 4 5) para dar como resultado

(fac-cola 3 20)

Recursión de cola

(fac-cola 4 5)

= (fac-cola (- 4 1) (* 4 5)) i.e.

(fac-cola 3 20)

```
(define (fac-cola n acc)
    (if (zero? n)
        acc
        (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 4 5) es la expr. del else =>

(fac-cola (- 4 1) (* 4 5))

En este punto ya terminó la llamada de

(fac-cola 4 5) para dar como resultado

(fac-cola 3 20)

Recursión de cola

(fac-cola 3 20)

= (fac-cola (- 3 1) (* 3 20))

(fac-cola 2 60)

```
(define (fac-cola n acc)
    (if (zero? n)
        acc
        (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 3 20) es la expr. del else =>

(fac-cola (- 3 1) (* 3 20))

En este punto ya terminó la llamada de

(fac-cola 3 20) para dar como resultado

(fac-cola 2 60)

Recursión de cola

(fac-cola 2 60)

= (fac-cola (- 2 1) (* 2 60)) i.e.

(fac-cola 1 120)

```
(define (fac-cola n acc)
        (if (zero? n)
                acc
                (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 2 60) es la expr. del else =>

(fac-cola (- 2 1) (* 2 60))

En este punto ya terminó la llamada de

(fac-cola 2 60) para dar como resultado

(fac-cola 1 120)

Recursión de cola

(fac-cola 1 120)

= (fac-cola (- 1 1) (* 1 120)) i.e.

(fac-cola 0 120)

```
(define (fac-cola n acc)
        (if (zero? n)
                acc
                (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 1 120) es la expr. del else =>

(fac-cola (- 1 1) (* 1 120))

En este punto ya terminó la llamada de

(fac-cola 1 120) para dar como resultado

(fac-cola 0 120)

Recursión de cola

(fac-cola 0 120)

= 120

```
(define (fac-cola  n  acc)
        (if (zero? n)
                acc
                (fac-cola (- n 1) (* n acc))))
```

El resultado de (fac-cola 0 120) es la expr. del else =>

120

que es el acumulador (acc)

Recursión de cola

>(fact 5)

= (fac-cola 5 1)

= (fac-cola 4 5)

= (fac-cola 3 20)

= (fac-cola 2 60)

= (fac-cola 1 120)

= (fac-cola 0 120)

=120

(fact 5)

(fac-cola 5 1) i.e. res = (fac-cola (- 5 1) (* 5 1))

(fac-cola 4 5) i.e. (fac-cola (- 4 1) (* 4 5))

(fac-cola 3 20) i.e. (fac-cola (- 3 1) (* 3 20))

(fac-cola 2 60) i.e. (fac-cola (- 2 1) (* 2 120))

(fac-cola 1 120) i.e. (fac-cola (- 1 1) (* 1 120))

(fac-cola 0 1) i.e. res=acc

120

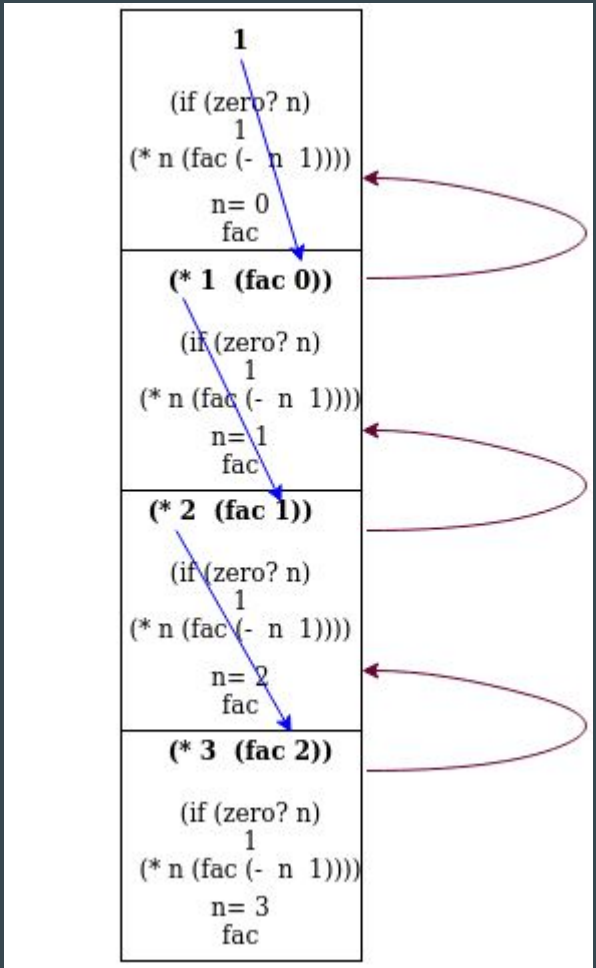
Preguntas

1. ¿Cuántas llamadas a función principal *fac-cola* se hicieron?

1. ¿Cuántos registros de activación se crearon con la llamada a *fac 5*?

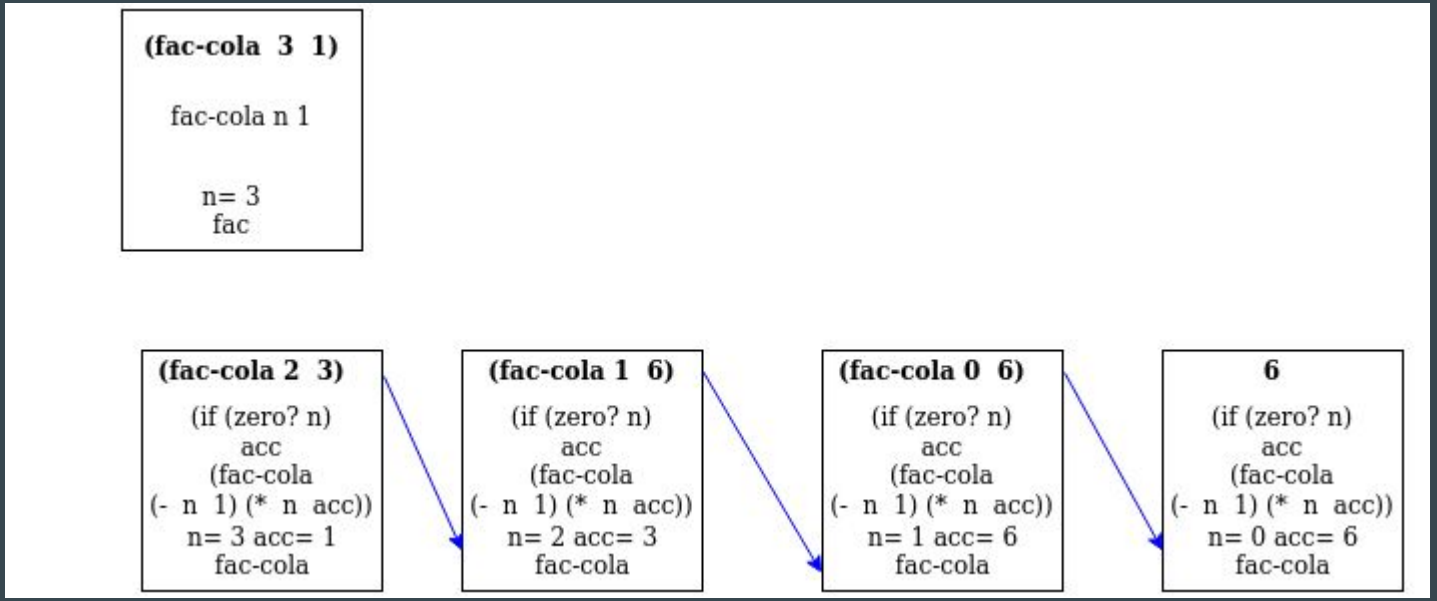
1. ¿Cuántos registros de activación están en un mismo momento en ejecución?

1. ¿Algún registro tiene que esperarse a resolver alguno de sus parámetros para mantenerse en ejecución?



Recursión

(fact 3)
= (* 3 (fact 2))
= (* 3 (* 2 (fact 1)))
= (* 3 (* 2 (* 1 (fact 0))))
= (* 3 (* 2 (* 1 1)))
= (* 3 (* 2 1))
= (* 3 2)
= 6



Recursión de Cola

(fact 3)

= (fac-cola 3 1)

= (fac-cola 2 3)
= (fac-cola 1 6)
= (fac-cola 0 6)
= 6

Función filter-pos

Implementación usando recursión y recursión de cola

Función: filter-pos

```
(define (filter-pos l)
    (cond [(empty?  l) empty]
          [else
          (if (> (first  l) 0)
              (cons (first  l) (filter-pos (rest l)))
             (filter-pos (rest l))) ] ))
```

```
> (filter-pos '())
empty

> (filter-pos '( 1   0  -1))
  (cons 1 (filter-pos '(0  -1)))
  (cons 1 (filter-pos '(-1)))
  (cons 1 (filter-pos '())
  (cons 1 empty)  i.e.
 (1)
```

- ¿Esta función es recursiva o recursiva con cola?

Paso 1: hacer una función que preserve la firma de ésta

```
(define (filter-pos l)

        (filter-pos/tail  l  ??))
```

filter-pos/tail debe recibir los mismo parámetros de la función original y agregamos uno más que será el ACUMULADOR de las llamadas recursivas con cola.

¿Qué ponemos en el acumulador “??” que hará la llamada ?

SI, es el caso base de la recursión.

Paso 2: implementamos filter-pos/tail -caso base-

```
(define (filter-pos l)
        (filter-pos/tail  l  empty))
```

```
(define (filter-pos/tail   l   acc)
        (cond [(empty?  l)  acc ]
              [else
```

CASO BASE: la definición de la función debe de llevar el caso base el cual regresa el acumulador que es instanciado la primera vez como lo que regresa el caso base en la función original.

Paso 3: implementamos filter-pos/tail -caso recursivo-

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
       (if (> (first  l) 0)
            (filter-pos/tail (rest l)  ??? )
```

Recordemos la función original:

```
(define (filter-pos l)
   (cond
     [(empty?  l) empty]
      [else
        (if (> (first  l) 0)
             (cons (first  l) (filter-pos (rest l)))
             (filter-pos (rest l))) ] ))
```

Paso 3: implementamos filter-pos/tail -caso recursivo-

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
      (if (> first  l) 0)
      (filter-pos/tail (rest  l)  (cons (first l) acc ))
```

Recordemos la función original:

```
(define (filter-pos l)
   (cond
     [(empty?  l) empty]
      [else
        (if (> (first  l) 0)
              (cons (first  l) (filter-pos (rest l)))
              (filter-pos (rest l))) ] ))
```

Paso 4: implementamos filter-pos/tail -terminar el if-

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
      (if (> (first  l) 0)
      (filter-pos/tail (rest l)  (cons (first l) acc ))
      (filter-pos/tail (rest l)  acc )) ] ))
```

Recordemos la función original:

```
(define (filter-pos l)
   (cond
     [(empty?  l) empty]
      [else
        (if (> (first  l) 0)
              (cons (first  l) (filter-pos (rest l)))
              (filter-pos (rest l))) ] ))
```

Ejecución de filter-pos/tail

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
      (if (> (first  l) 0)
      (filter-pos/tail (rest l)  (cons (first l) acc ))
      (filter-pos/tail (rest l)  acc)) ] ))
```

> (filter-pos '(3 2 -1))

= (filter-pos/tail '(3 2 -1) empty)

l = '(3 2 -1)
acc = empty

(empty? '(3 2 -1)) ;;FALSE

(if (> 3   0) ;;TRUE
    (filter-pos/tail '(2 -1) (cons 3 empty)))...

= (filter-pos/tail '(2 -1) (cons 3 empty))

Ejecución de filter-pos/tail

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
      (if (> (first  l) 0)
      (filter-pos/tail (rest l)  (cons (first l) acc ))
      (filter-pos/tail (rest l) acc)) ] ))
```

```
> (filter-pos/tail  '(2 -1)  (cons 3 empty))
```

l = '(2 -1)
acc = (cons 3 empty) = '(3)

```
(empty? '(2  -1)) ;;FALSE

(if  (>  2   0)  ;;TRUE
     (filter-pos/tail '(-1) (cons 2
                                  (cons 3 empty)) ...))
```

```
= (filter-pos/tail  '(-1)  (cons 2 (cons 3 empty)) )
```

Ejecución de filter-pos/tail

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
      (if (> (first  l) 0)
      (filter-pos/tail (rest l)  (cons (first l) acc ))
      (filter-pos/tail (rest l) acc )) ] ))
```

```
> (filter-pos/tail  '(-1)  (cons 2 (cons 3 empty)) )

          l  =  '(-1)
acc   =   (cons 2 (cons 3 empty))
='(2  3)

(empty? '(-1)) ;;FALSE

(if  (>  -1    0)  ;;FALSE
     (filter-pos/tail '() (cons 2
                                (cons 3 empty)) )


= (filter-pos/tail  '()  (cons 2 (cons 3 empty)) )
```

Ejecución de filter-pos/tail

```
(define (filter-pos/tail   l   acc)
  (cond
    [(empty?  l)  acc ]
    [else
      (if (> (first  l) 0)
      (filter-pos/tail (rest l)  (cons (first l) acc ))
      (filter-pos/tail (rest l) acc )) ] ))
```

> (filter-pos/tail '() (cons 2 (cons 3 empty)))

l = '()
acc = (cons 2 (cons 3 empty))
='(2 3)

(empty? '()) ;;TRUE

= (cons 2 (cons 3 empty)) i.e.

'(2 3)

La lista original es: '(3 2 -1)

- ¿Cómo podríamos respetar el orden de los elementos?

Manejo de espacio

¿Cuántos registros de activación se crean por cada llamada a función principal con cada una de las siguientes llamadas a función filter-pos usando una implementación:

1. Recursiva
    a. (filter-pos ‘())

    b. (filter-pos ‘(1))

    c. (filter-pos ‘(3 2 -1)

2. Recursiva con cola
    a. (filter-pos/tail ‘())
    b. (filter-pos/tail ‘(1))
    c. (filter-pos/tail ‘(3 2 -1)

Ejemplo de FOR con recursión de cola

```
(define (for init condition change body result)
    (if (condition init)
        (for (change init) condition change body (body init result))
        result))
```

- ¿Está implementada como función recursiva o como recursiva con cola?

Ejemplo: FOR

(for init condition change body result)

> (for 10 positive? sub1 + 0)

(if (positive? 10) ;TRUE

= (for (sub1 10) positive? sub1 + (+ 10 0)) i.e.

= (for 9 positive? sub1 + 10)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 9 positive? sub1 + 10)

(if (positive? 9) ;TRUE

= (for (sub1 9) positive? sub1 + (+ 9 10)) i.e.

= (for 8 positive? sub1 + 19)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 8 positive? subl + 19)

(if (positive? 8) ;TRUE

= (for (subl 8) positive? subl + (+ 8 19)) i.e.

= (for 7 positive? subl + 27)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

```
> (for 7  positive?  subl  +  27)

 (if (positive? 7 )   ;TRUE

= (for (subl 7) positive? subl + (+ 7  27))  i.e.


= (for 6 positive? subl + 34)
```

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

  (for (change init) condition change body (body init result))

     result))
```

(for init condition change body result)

> (for 6 positive? subl + 34)

(if (positive? 6) ;TRUE

= (for (subl 6) positive? subl + (+ 6 34)) i.e.

= (for 5 positive? subl + 40)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 5 positive? sub1 + 40)

(if (positive? 5) ;TRUE

= (for (sub1 5) positive? sub1 + (+ 5 40)) i.e.

= (for 4 positive? sub1 + 45)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 4 positive? subl + 45)

(if (positive? 4) ;TRUE

= (for (subl 4) positive? subl + (+ 4 45)) i.e.

= (for 3 positive? subl + 49)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

   result))
```

(for init condition change body result)

> (for 3 positive? subl + 49)

(if (positive? 3) ;TRUE

= (for (subl 3) positive? subl + (+ 3 49)) i.e.

= (for 2 positive? subl + 52)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 2 positive? sub1 + 52)

(if (positive? 2) ;TRUE

= (for (sub1 2) positive? sub1 + (+ 2 52)) i.e.

= (for 1 positive? sub1 + 54)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 1 positive? subl + 54)

(if (positive? 1) ;TRUE

= (for (subl 1) positive? subl + (+ 1 54)) i.e.

= (for 0 positive? subl + 55)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

> (for 0 positive? subl + 55)

(if (positive? 0) ;TRUE

= (for (subl 0) positive? subl + (+ 0 55)) i.e.

= (for -1 positive? subl + 55)

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

(for (change init) condition change body (body init result))

    result))
```

(for init condition change body result)

```
> (for -1  positive?  sub1  +  55)

 (if (positive? -1)   ;FALSE


= 55                  ;que es result (else-expr)
```

```
;; Código de for, usando recursión de cola

(define (for init condition change body result)

   (if (condition init)

  (for (change init) condition change body (body init result))

    result))
```

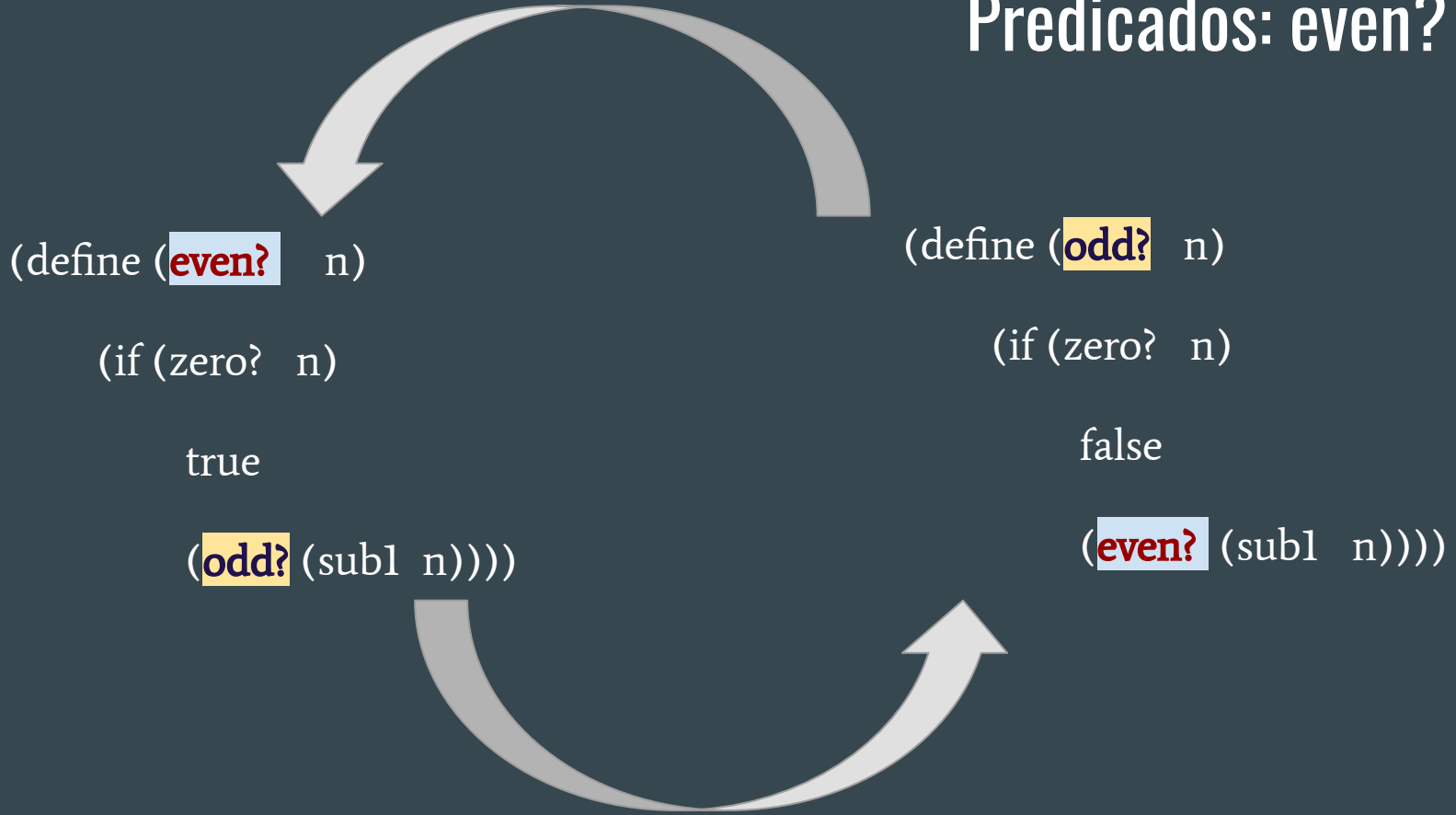
Procedimientos mutuamente recursivos

Procedimientos Mutuamente Recursivos

```
(define (even? n)
        (if (zero? n)
                true
                (odd? (sub1 n))))


(define (odd? n)
        (if (zero? n)
                false
                (even? (sub1 n))))
```

Predicados: even? y odd?



Ejecución de even? y odd? con la misma instancia

> (even? 2)

= (odd? 1)

= (even? 0)

= true

> (odd? 2)

= (even? 1)

= (odd? 0)

= false

¿Son procedimientos implementados
con recursión de cola?

GRACIAS

¿Dudas?