Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

•••

Karla Ramírez Pulido

Estilo de Paso de Continuaciones
Continuation Passing Style

Continuation Passing Style, CPS

Estilo de Paso de Continuaciones

Transformaciones a procedimientos de Orden Superior

¡¡WOW!!

Llevamos el estado del PROCEDIMIENTO, porque tenemos TODO EL PROCEDIMIENTO.

Ejemplo de la vida real

¿Qué pasa cuando vamos a una fiesta y tomamos fotos?

¿Qué pasa cuando solo muestran fotos?

¿Qué pasa cuando recuerdan la fiesta?

Si tuviéramos una memoria excelente (no se olvida ningún detalle) podríamos relatar completa la fiesta... es más a veces hasta podríamos “revivir el momento”

Ahora pensemos en un partido de futbol:

¿Qué pasó? ¿quiénes podrían decirnos algo? ¿podrán decirnos todo?

¿Que es una continuación?

(f v1 ... vm)

Si evaluamos la función “f” tendríamos como respuesta un valor “a”,

entonces ejecutamos mejor (f/k v1 ... vm k)

donde a “k” le aplicamos el valor de “a”

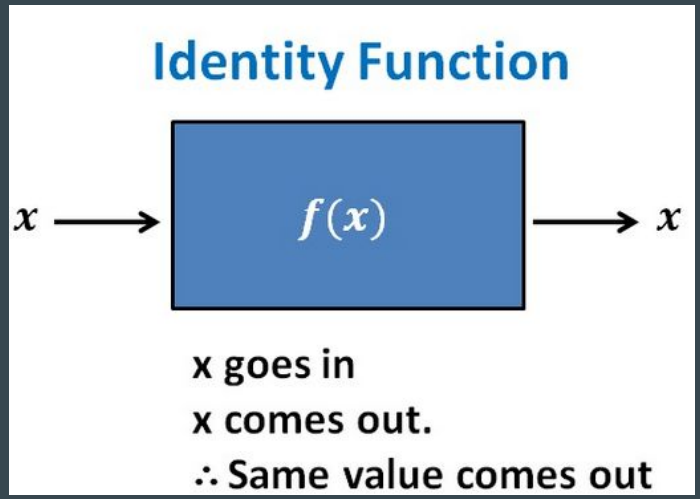
Entonces “k” debe de ser una función

Idiom de continuaciones: usando “k”

¿Qué función es k?

k es la función identidad

i.e. (lambda (x) x) ó (λ(x)x)



recibe y regresa lo mismo

¡¡OOHH!! es como nuestro acumulador en recursión de cola...

Factorial con CPS

```
;;Factorial llamemos usual

(define (fact n)

   (if (= n 0)

      1

      (* n (fact (- n 1)))))
```

```
;;Factorial con recursión de cola

(define (fact n)

      (fact/cola   n   1))


(define (fact/cola  n   acc)

       (if  (= n 0)

             acc

             (fact/cola (-  n 1)  (*  n acc))))
```

Factorial con CPS

```
;;Factorial con recursión de cola

(define (fact n)

        (fact/cola   n   1))


(define (fact/cola  n   acc)

       (if  (= n 0)

            acc

            (fact/cola (-  n 1)  (*  n acc))))
```

```
;;Factorial con CPS

(define (fact n)

        (fact/k  n   (λ(x) x)))


(define (fact/k  n   k)

       (if  (=  n  0)

            (k   1)

            (fact/k (-  n 1)  (λ(v) (k  (* n v))))))
```

Ejecutemos factorial con CPS con cero

(fact 0)

(fact/k 0 (λ(x) x))

((λ(x) x) 1)

x = 1

evalúo el cuerpo de la función identidad i.e. “x” y regresa 1

```
;;Factorial con CPS

(define (fact n)

        (fact/k  n  (λ(x) x)))

(define (fact/k  n  k)

        (if  (=  n  0)

             (k  1)

             (fact/k (-  n 1)  (λ(v) (k  (* n v))))))
```

Ejecutemos factorial con CPS con uno

(fact 1)

(fact/k 1 (λ(x) x))

(fact/k (- 1 1) (λ(v) ((λ(x) x) (* 1 v)))

i.e.

(fact/k 0 (λ(v) ((λ(x) x) (* 1 v)))

(k 1)

((λ(v) ((λ(x) x) (* 1 v))) 1)

v = 1

```
;;Factorial con CPS

(define (fact n)

   (fact/k n (λ(x) x)))

(define (fact/k n k)

   (if (= n 0)

       (k 1)

       (fact/k (- n 1) (λ(v) (k (* n v)))))))
```

Ejecutemos factorial con CPS con uno

((λ(v) ((λ(x) x) (* 1 v))) 1)

v = 1 sustituimos en la “v” ligada dentro del cuerpo de la función

(λ(v = 1) ((λ(x) x) (* 1 1)))

((λ(x) x) (* 1 1)) = ((λ(x) x) 1) = (λ(x = 1) x) evaluamos el cuerpo de esa función = 1

Definición de factorial usando CPS

```
(define (fact n)
        (fact/k  n   (λ(x) x)))


(define (fact/k  n   k)
        (if  (=  n  0)
             (k   1)
             (fact/k (-  n 1)  (λ(v) (k  (* n v))))))
```

Ejecutemos factorial con CPS con 5

(fact 5)

(fact/k 5 (λ(x) x))

n = 5 y k = (λ(x) x)

(fact/k (- n 1) (λ(x) x)) donde n = (- 5 1) = 4 y k = (λ(x) x)

(fact/k 4 (λ(v) ((λ (x) x) (* 5 v))))

donde n = 4 y k = (λ(v) ((λ (x) x) (* 5 v)))

(fact/k 3 (λ(v) ((λ(v) ((λ(x) x) (* 5 v)))

(* 4 v))))

donde n = 3 y k= (λ(v) ((λ(v) ((λ(x) x)(* 5 v))) (* 4 v)))

(fact/k 2 (λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))

(* 3 v))))

donde n = 2 y k = (λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) (* 3 v)))

(fact/k 1 (λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) (* 3 v)))

(* 2 v))))

donde n = 1 y

k = (λ(v) ((λ(v) (* (λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) (* 3 v))) (* 2 v)))

(fact/k 0 (λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))

(* 3 v))) (* 2 v))) (* 1 v)))

(fact/k 0 (λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))

(* 3 v))) (* 2 v))) (* 1 v)))

donde n = 0 y

k = (λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))

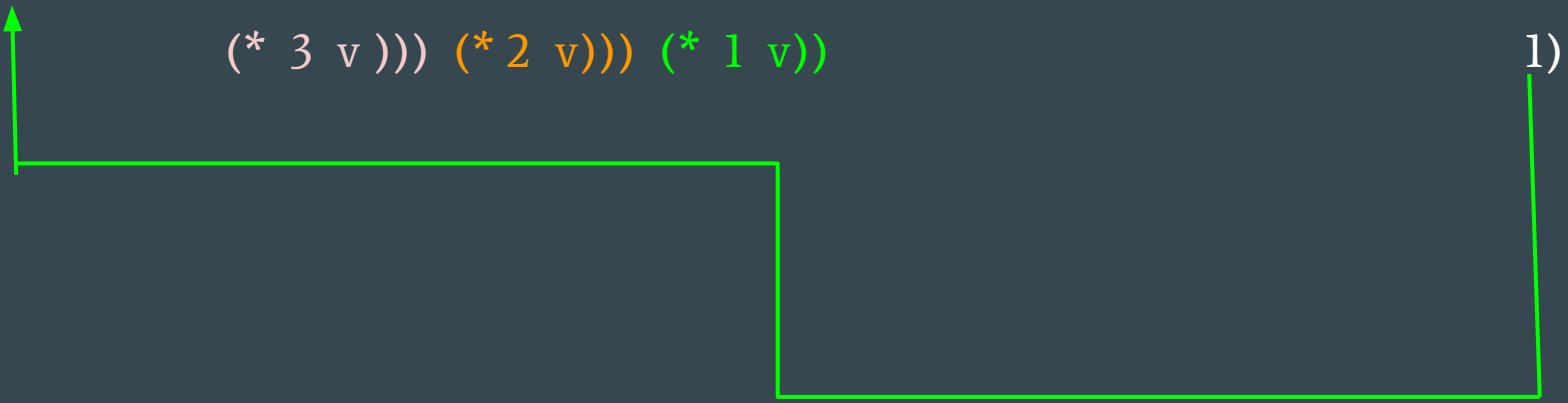
(* 3 v))) (* 2 v))) (* 1 v))

y como (zero? n) se cumple, entonces

(k 1)

```
(k 1)
```

```
( (λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))
```



Asignamos al parámetro formal “v” su valor que es 1, i.e. v = 1

Al asignar v = 1, sustituir en el alcance esa (λ(v) ... (* 1 1)) con 1 tenemos:

((λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))

(* 3 v))) (* 2 v))) (* 1 1))

((λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v)))

(* 3 v))) (* 2 v))) 1)

Asignamos al parámetro formal "v" su valor que es 1, i.e. v = 1

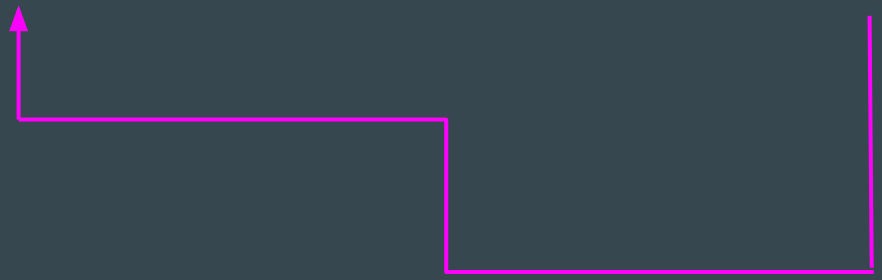
(λ(v) ((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) (* 3 v)))(* 2 1))

((λ(v) ((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) (* 3 v))) 2)

Asignamos al parámetro formal “v” su valor que es 1, i.e. v = 2

((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) (* 3 2)))

((λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 v))) 6))



Asignamos al parámetro formal “v” su valor que es 1, i.e. v = 6

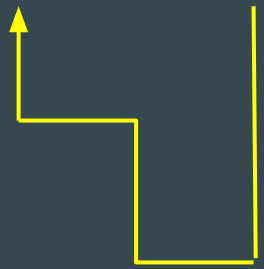
(λ(v) ((λ(v) ((λ(x) x) (* 5 v))) (* 4 6)))

((λ(v) ((λ(x) x) (* 5 v))) 24)

Asignamos al parámetro formal “v” su valor que es 1, i.e. v = 24

(λ(v) ((λ(x) x) (* 5 v))) 24)

((λ(x) x) (* 5 24)))

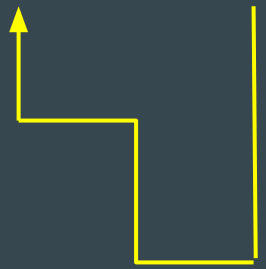


Asignamos al parámetro formal “x” su valor que es 1, i.e. x = 120

Regresamos 120 (cuerpo de la función identidad)

((λ(x) x) (* 5 24)))

((λ(x) x) 120))



Asignamos al parámetro formal “x” su valor que es 1, i.e. x = 120

```
(define (sum n)
    (if (zero? n)
        0
        (+ n (sum (sub1 n)))))
```

> (sum 1000)

> (sum 2)

(+ 2 (+ 1 (sum 0)))

(+ 2 (+ 1 0))

(+ 2 1)

3

¿Cuál es el comportamiento de la función sum?

```
(define (sum n)
  (if (zero? n)
      0
      (+ n (sum (sub1 n)))))
```

Modificarla usando la técnica de recursión de cola

```
(define (sum n)
  (sum-recC n 0))

(define (sum-recC n acc)
  (if (zero? n)
      acc
      (sum-recC (sub1 n) (+ n acc))))
```

> (sum 2)

genera una nueva llamada:

(sum-recC 2 0)

¿Qué regresa sum-recC?

```
(define (sum n)
      (sum-recC   n  0))


(define (sum-recC  n  acc)
      (if (zero?  n)
            acc
            (sum-recC (sub1  n) (+ n acc))))
```

```
(sum-recC   2    0)
             n   acc

(sum-recC (sub1 2) (+ 2 0))
→ (sum-recC  1   2)
              n    acc

(sum-recC  (sub1 1)  (+ 1 2))

→ (sum-recC   0    3)
              n    acc

Resultado:   3
```

```
(define (sum n)
      (sum-recC   n  0))


(define (sum-recC  n  acc)
      (if (zero?  n)
            acc
            (sum-recC (sub1  n) (+ n acc))))
```

Transforma la función sum usando CPS

```
(define (sum n)
      (sum/k  n (λ(x) x)))


(define (sum/k  n   k)
  (if (zero? n)
      (k 0)
      (sum/k (sub1 n) (λ(v) (+ n v)))))
```

> (sum 3)

→ (sum/k 3 (λ(x) x))
n k
(zero? 3)

(sum/k 2 (λ(v) ((λ(x) x) (+ 3 v))))
(zero? 2)
n=2 k' = (λ(v) (k (+ 3 v)))

→ (sum/k 1 (λ(v) (k' (+ 2 v))))
(zero? 1)
n=1 k" = (λ(v) (k' (+ 2 v)))

```
(define (sum n)
      (sum/k  n (λ(x) x)))


(define (sum/k  n   k)
  (if (zero? n)
      (k 0)
      (sum/k (subl n) (λ(v) (k (+ n v))))))
```

→ (sum/k 1 (λ(v) (k' (+ 2 v))))
(zero? 1)
n k" = (λ(v) (k' (+ 2 v)))

(sum/k 0 (λ(v) (k" (+ 1 v))))
(zero? 0)
(k 0)
((λ(v) (k" (+ 1 v))) 0)

Asignación: v = 0
Sustituir en el cuerpo de la función:

(k" (+ 1 0))
(k" 1)

```
(define (sum n)
    (sum/k n (λ(x) x)))

(define (sum/k n k)
  (if (zero? n)
      (k 0)
      (sum/k (sub1 n) (λ(v) (k (+ n v))))))
```

→ (sum/k 1 (λ(v) (k' (+ 2 v))))
(zero? 1)
n=1 k" = (λ(v) (k' (+ 2 v)))

(sum/k 0 (λ(v) (k" (+ 1 v))))
(zero? 0)
(k 0)
((λ(v) (k" (+ 1 v))) 0)

Asignación: v = 0
Sustituir en el cuerpo de la función:

(k" (+ 1 0))
(k" 1)

k" = (λ(v) (k' (+ 2 v)))

Sustituyendo k":

(k" 1)
((λ(v) (k' (+ 2 v))) 1)

Asignación v = 1
Sustituir en el cuerpo de la función:

(k' (+ 2 1))
k' = (λ(v) (k (+ 3 v)))

(k' (+ 2 1))
k' = (λ(v) (k (+ 3 v)))

((λ(v) (k (+ 3 v))) (+ 2 1))

((λ(v) (k (+ 3 v))) 3)

Asignamos a v = 3

Sustituir en el cuerpo de la función

v = 3

(k (+ 3 3))

k = (λ(x) x)

((λ(x) x) 6)

Asignamos x = 6
sustituimos en el cuerpo de la función

→ 6

Transforma la función make-list usando:

- recursión de cola
- CPS

```
(define (make-list  n)
        (if (zero?  n)
                '()
                (cons  n  (make-list (sub1 n)))))
```

make-list: con recursión de cola

```
(define (make-list   n)
     (if (zero? n)
          '()
     (cons n (make-list (sub1 n)))))
```

```
> (make-list   2 )
(cons  2 (make-list  1))
(cons  2 (cons  1 (make-list  0)))
(cons  2 (cons  1  '() ))
= '( 2  1)
```

¿Qué hace make-list?

make-list2: con recursión de cola

```
(define (make-list n)
    (if (zero? n)
        '()
        (cons n (make-list (sub1 n)))))
```

```
(define (make-list  n)
    (make-list2   n   '() ))


(define (make-list2   n   acc)
    (if (zero? n)
        acc
        (make-list2  (sub1 n) (cons n  acc))))
```

make-list2: con recursión de cola

> (make-list 2)

(make-list2 2 ‘())

(make-list2 1 (cons 2 ‘()))

(make-list2 0 (cons 1 (cons 2 ‘())))

(cons 1 (cons 2 ‘()))

→ ‘(1 2)

Nos da una lista con los mismos elementos pero en distinto orden, esto es porque cons no es conmutativo.

```
(define (make-list  n)
        (make-list2  n  ‘() ))


(define (make-list2  n  acc)
        (if (zero? n)
                acc
                (make-list2  (sub1 n) (cons n  acc))))
```

Problema de la conmutatividad de cons:

Se corrige usando snoc: en vez de añadir al inicio de la lista, snoc añade al final de la lista.

```
(define (snoc  i  lst)
        (if  (null?  lst)
                (cons  i  '())
                (cons  (car  lst) (snoc  i  (cdr lst)))))
```

```
;; Con recursión de cola

(define (make-list  n)

        (make-list2   n   '() ))


(define (make-list2   n   acc)

        (if (zero? n)

          acc

         (make-list2  (sub1 n) (cons n  acc))))
```

¿Qué tengo que hacer para pasar a CPS?

- Ya tengo 2 funciones
- El acc = función identidad la primera ves

Caso Base

- Regreso k con la función identidad

Caso recursivo

- Regreso una llamada a función nueva donde:
  - n= sub1 n
  - acc se convierte en

    k = (lambda (v) (k (cons n v))

make-list: con CPS

```
(define (make-list  n)
      (make-list/k   n   (λ(x) x) ))


(define (make-list/k  n   k )
      (if (zero? n)
        (k  '() )
        (make-list/k   (sub1 n)  (λ(v) (k (cons n  v))) )))
```

> (make-list 1)

```
(define (make-list n)
        (make-list/k n (λ (x) x) ))
(define (make-list/k n k )
        (if (zero? n)
          (k '() )
          (make-list/k (sub1 n)
                       (λ(v) (k (cons n v))))))
```

> (make-list 1)

= (make-list/k 1 (λ (x) x))
n = 1
k = (λ (x) x)

= (make-list/k (sub1 1) ((λ(v) ((λ (x) x)
(cons n v)))))

i.e. (make-list/k 0 ((λ(v) ((λ (x) x)
(cons 1 v)))))

```
> (make-list  1)

(define (make-list/k  n  k )
  (if (zero? n)
      (k  '() )
      (make-list/k  (subl n)
                    (λ(v) (k (cons n  v)))
  )))
```

```
(make-list/k  0  (λ(v) ((λ (x) x)
                        (cons  1  v))) )
 n = 0
k =  (λ(v) ((λ (x) x) (cons  1  v)))

( k  '() )

= ( (λ(v) ((λ (x) x)  (cons  1  v)))  '() )
v = '( ) entonces sustituimos en (cons  1 v )

= (λ(v) ((λ (x) x)  (cons  1  '( ) )))
i.e. ( (λ (x) x)  (cons  1  '( )) )
 x = '(1)
Sustituimos en el cuerpo de la función ID
= '(1)
```

map

(map func list)

(map add1 ‘(1 2 3))
Resultado: (2 3 4)

```
(define (map  f   lst)
  (if (null? lst)
      ‘()
      (cons (f (car lst)) (map f (cdr lst)))))
```

Convertir la función map a CPS

```
(define (map  f  lst)

      (map2  f   lst  (lambda(x) x) ))
```

```
;;Primero : Agregamos la k como parámetro

(define (map2  f   lst   k)

  (if (null? lst)

     ....  ))
```

```
;;Agregamos la k en el cuerpo de la
;;función donde está el caso base, y lo
;;hacemos como app de función

(define (map2  f   lst   k)

  (if (null? lst)

       (k  '())

       ((cons (f  (car lst))

                  (map2 f (cdr lst))))))
```

Convertir la función map a CPS

```
;; Agregamos la k como parámetro
(define (map2  f   lst   k)
  (if (null? lst)
      '()
      (cons (f  (car lst))
            (map2 f (cdr lst)))))
```

```
;;Agregamos la k en el cuerpo de la
;;función donde está el caso base, y lo
;;hacemos como app de función
(define (map2  f   lst   k)
  (if (null? lst)
      (k  '())
      ((cons (f  (car lst))
             (map2 f (cdr lst))))))
```

Convertir la función map a CPS

```
;;Agregamos la k en el cuerpo de la función
;;donde está la llamada recursiva

(define (map2  f   lst   k)
  (if (null? lst)
      (k  '())
      (k (cons (f  (car lst))
               (map2 f (cdr lst))))))
```

```
;;Levantamos la llamada    (f (car lst))

(define (map2  f   lst   k)
  (if (null? lst)
      (k  '())
      (f  (car lst))
          (k (cons (map2  f  (cdr lst))))))
```

Convertir la función map a CPS

```
;;Creamos una nueva función-continuación

;;(lambda (v) ...)

(define (map2  f   lst   k)

   (if (null? lst)

         (k  '())

         (f  (car lst))

             (lambda (v) (k (cons (map2  f  (cdr lst))))))
```

Convertir la función map a CPS

```
;;(lambda (v) ...)  y agregamos el argumento v ligado al parámetro de la función

;;OJO lo que quedaba en recursión como cálculos pendientes alguno debe de tener "v"

(define (map2  f   lst   k)

  (if (null? lst)

      (k  '())

      (f  (car lst))

          (lambda (v) (map2  f  (cdr lst) (k (cons v  ???))))))))
```

Convertir la función map a CPS

```
;;entonces creo una nueva función lambda(v2) que lleve el resto del cálculo

(define (map2  f   lst   k)

   (if (null? lst)

         (k  '())

         (f  (car lst))

             (lambda (v) (map2  f  (cdr lst) (lambda (v2) (k (cons v  v2))))))))
```

Tarea opcional 1: 1/2 pt. extra sobre examen 3

Haz la ejecución paso por paso (no se puede renombrar a las k's como k', k", etcétera) con la siguiente instancia:

(make-list 5)

usando la implementación de CPS

Entrega única e individual: 25 de noviembre de 2022

Tarea opcional 2: 1/2 pt. extra sobre examen 3

Haz la ejecución (no se puede renombrar a las k's como k', k", etcétera) paso por paso de:

```
> (map add1 '(1 2 3))
```

usando la implementación de CPS vista en clase.

Entrega única e individual: 3 de mayo de 2023