

Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación



Karla Ramírez Pulido
Continuaciones

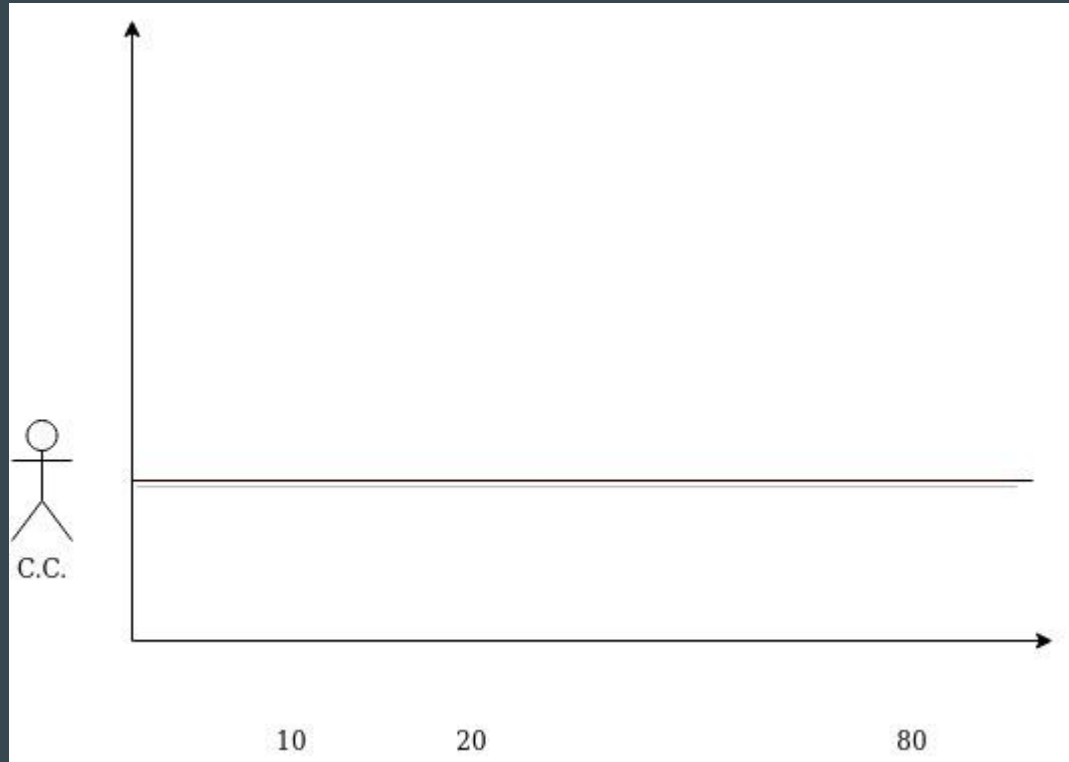
Idea introductoria

Supongamos...



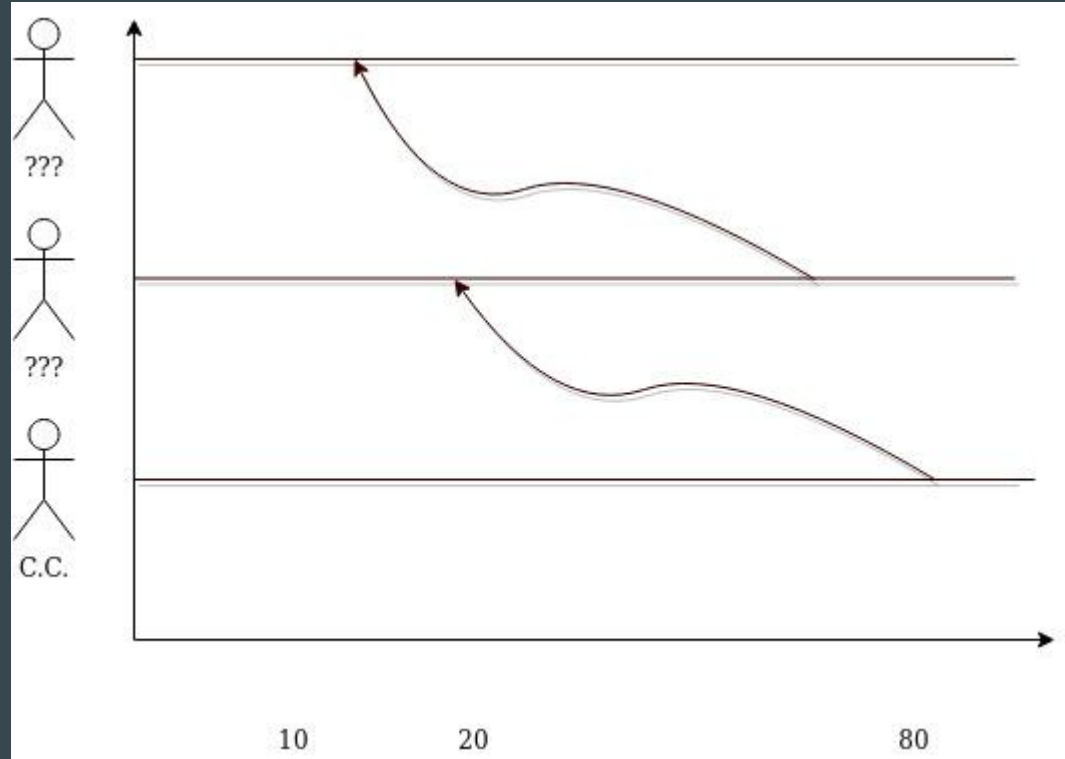
Idea introductoria

Ahora....



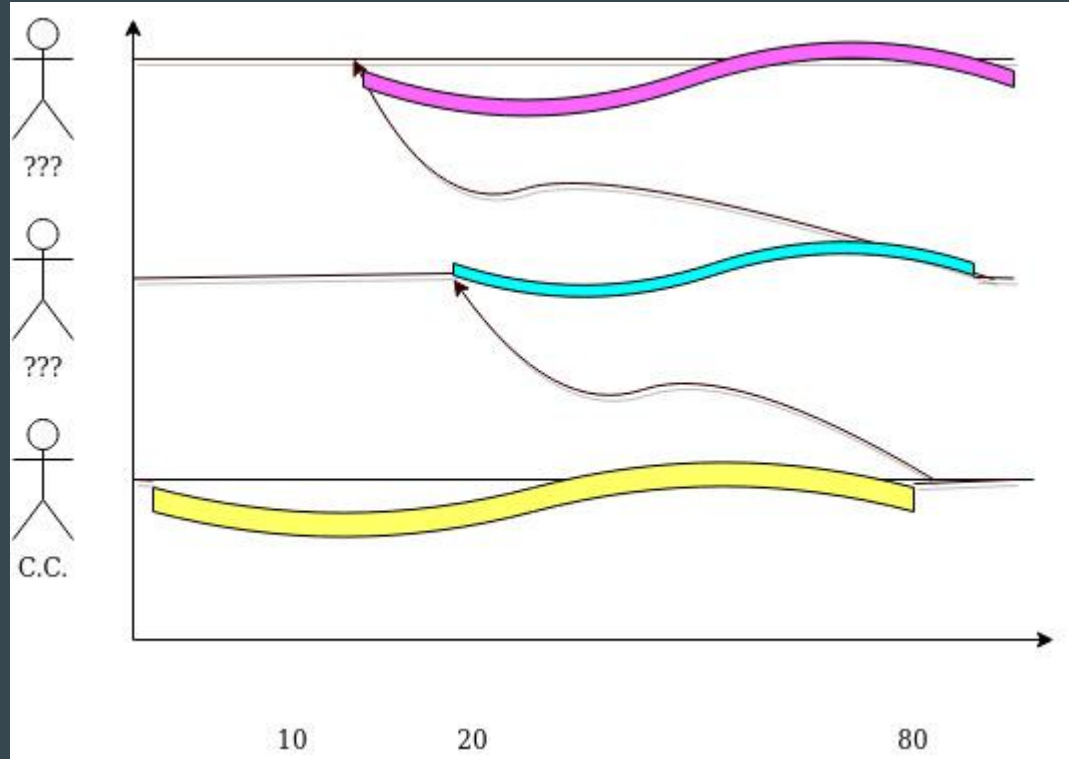
Idea introductoria

Ahora....



Idea introductoria

Ahora....



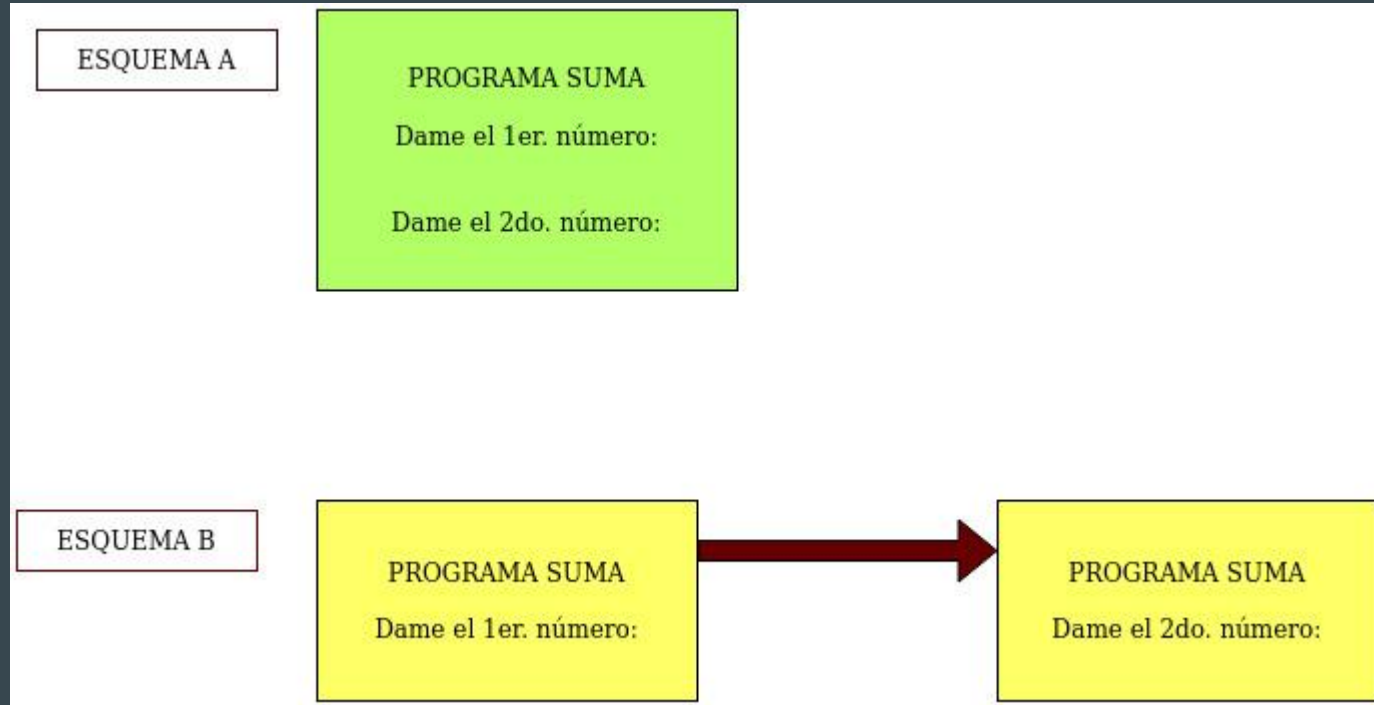
Continuación

Que lleva el contexto de todo lo que debe de hacer la función.

¿Qué significa llevar el contexto?

Forma de terminar: escapar de la ejecución (sape)

Ejemplo Programa SUMA en Web



Estructura de programas en WEB

```
(web-display  
  (+ (web-read "First number: ")  
     (web-read "Second number: ")))
```

```
(web-display  
  (+ •  
     (web-read "Second number: ")))
```

```
(lambda (•)  
  (web-display  
    (+ •  
       (web-read "Second number: "))))
```


Estructura de programas en WEB

```
(web-display  
  (+  
    (web-read/k "First number: "  
      (lambda(•) • ))  
    (web-read "Second number: ")))
```

```
(web-read/k "First number: "  
  (lambda(•)  
    (web-display  
      (+ •  
        (web-read "Second number: "))))))
```

Estructura de programas en WEB

```
(web-read/k
  "First number: "
  (lambda (•)
    (web-read/k "Second number: "
      (lambda (•)
        (web-display (+ • •)))))))
```

```
(web-read/k
  "First number: "
  (lambda (•1)
    (web-read/k "Second number:"
      (lambda (•2)
        (web-display(+ •1 •2))))))
```

Estructura de programas en WEB

```
(web-read/k "First number:"  
  (lambda (•1)  
    (web-read/k "Second number: "  
      (lambda (•2)  
        (web-display (+ •1 •2)))))))
```

El programa principal es:

```
(web-read/r "First number: " f1)  
  
(define (f1 •1)  
  (web-read/r "Second number: " f2))  
  
(define (f2 •1 •2)  
  (web-display (+ •1 •2)))
```

El esquema original es:

(f v1 ... vm)

la transformación como en CPS

(f/k v1 ... vm k)

Estructura de programas en WEB usando CPS

```
(+ (web-read"First number: ")  
   (web-read"Second number: "))
```

```
(web-read/k "First number: "  
  (lambda (l-val)  
    (web-read/k "Second number: "  
      (lambda (r-val)  
        (+ l-val r-val ))))))
```

```
(lambda(k)  
  (web-read/k "First number: "  
    (lambda (l-val)  
      (web-read/k "Second number: "  
        (lambda (r-val)  
          (k (+ l-val r-val ))))))))
```

CONTINUACIONES

Dos formas de usarlas:

call/cc = call with current
continuation

(call/cc (lambda (k)

... k ...)) ;;k es la continuación

(let/cc k

... k ...)) ;;k está ligada a la

;;continuation

Más ejemplos:

```
(call/cc (lambda (k)
           (k "foo")))
```

= "foo"

```
(call/cc (lambda (k)
           "foo"))
```

= "foo"

;; Las siguientes formas de expresar
;; continuaciones con call/cc son ;; equivalentes

(call/cc (lambda (k) expr)) <=>

(call/cc (lambda (k) (k expr)))

Más ejemplos:

```
(call/cc (lambda (k)
```

```
  (k "foo")
```

```
  (error "ignored"))))
```

```
= "foo"
```

; todo lo que está después de la "k" es ignorado

Uso de continuaciones

```
(let/cc k  
  (k 3))
```

```
(lambda (•) •)
```

;;La función identidad

Uso de continuaciones

```
(let/cc k  
  (k 3))
```

Si ya sabemos que esto regresa 3

Ahora evaluando una nueva
expresión:

```
(+ 1  
  (let/cc k  
    (k 3)))
```

```
(lambda (•) (+ 1 •))
```

Uso de continuaciones

= (+ 1
 ((lambda (•) (+ 1 •)) 3))

= (+ 1
 (+ 1 3))

= 4

(+ 1
 ((lambda↑ (•) (+ 1 •))
 3))

Se evalúa a 4

Otro ejemplo de continuación

```
(define (f n)
```

```
  (+ 10
```

```
    (* 5 (let/cc k
```

```
          (/ 1 n )))))
```

¿Cuál es el valor de la evaluación del siguiente código?

```
(define (f n)
```

```
  (+ 10
```

```
    (* 5
```

```
      (let/cc k
```

```
        (/ 1 n ))))))
```



```
(define (f n)
```

```
  (+ 10
```

```
    (* 5
```

```
      (λ↑ (•)
```

```
        (+ 10
```

```
          (* 5
```

```
            (/ 1 n ))))) )
```

Otro ejemplo de continuación

```
(define (f n)
```

```
  (+ 10
```

```
    (* 5 (let/cc k
```

```
      (/ 1 n )))))
```

```
> (+ 3 (f 0))
```

```
(define (f n)
```

```
  (+ 10
```

```
    (* 5
```

```
      (λ↑ (•)
```

```
        (+ 10
```

```
          (* 5
```

```
            (/ 1 n )))) )
```

Otro ejemplo de continuación

```
> (+ 3 (f 0))
```

```
(+ 3 (f 0))
```

```
(+ 10
```

```
(* 5
```

```
(λ↑ (•)
```

```
(+ 3
```

```
(+ 10
```

```
(* 5
```

```
(/ 1 0 )))))))
```

```
(define (f n)      ;;n=0
```

```
(+ 10
```

```
(* 5
```

```
(λ↑ (•)
```

```
(+ 3
```

```
(+ 10
```

```
(* 5
```

```
(/ 1 0 )))))))
```



La continuación ligada a k es:

(lambda↑(•)

(+3

(+10

(* 5 •))))

donde el punto es $(/ 1 n) = (/ 1 0)$

entonces el código completo es:

(+ 3 (f 0)

(+ 10

(* 5

(lambda↑(•)

(+ 3

(+ 10

(* 5

(/ 1 0))))))))

Otro ejemplo de continuaciones (ver orden de evaluación)

Con k explícita:

```
(+ 4 (let/cc k  
      (k (+ 1 2)) ))
```


```
(+ 4 (lambda↑(•)
```

```
(k (+ •  
      ;; donde el punto es 4  
      (+ 1 2))))))
```

Con k implícita:

```
(+ 4 (let/cc k  
       (+ 1 2) ))
```

```
(+ 4 (lambda↑(•)
```

```
 (+ 4  
      (+ 1 2))))
```


Otro ejemplo de continuaciones (ver orden de evaluación)


Con k explícita:

```
(+ 4 (let/cc k  
      (k (+ 4 (+ 1 2))))))
```

```
(+ 4 (lambda↑(•)  
      (k (+ 4 (+ 1 2))  
          )))
```

Con k implícita (no la escribió el programador)
debería de estar en el cuadro verde:

```
(+ 4 (let/cc k  
       (+ 4 (+ 1 2)) ))
```

```
(+ 4 (lambda↑(•)  
       (+ 4 (+ 1 2))  
      ))
```

Otro ejemplo de continuaciones: call/cc

```
(+ 4 (call/cc
      (lambda (k)
        (k (+ 1 2)))) )
```

1. ¿Cual es el resultado del código?
2. ¿Cuál es la continuación asociada es decir $\lambda \uparrow$?

Respondiendo las preguntas anteriores:

(+ 4 (call/cc

(lambda (k)

(k (+ 1 2)))))

(+ 4

(lambda↑(k)

(k (+ contexto-ant

(+ 1 2))))

= (k (+ 4 3)) = (k 7) = 7

Otro ejemplo de continuaciones

(string-append

"foo "

(call/cc (lambda (k) "bar "))

"boo")

= "foo bar boo"

¿Qué función principal es:?

string-append

i.e. concatenar cadenas

y los argumentos de
string-append (son 3)

1. "foo"
2. **continuación: call/cc ...**
3. "boo"

Otro ejemplo de continuaciones

(string-append

"foo "

(call/cc (lambda (k) "bar "))

"boo")

= "foo bar boo"

Recuerden que la función principal es `string-append` y éstos son sus argumentos

`"foo "` + `(call/cc (lambda(k) "bar "))` + `"boo"`

= `"foo "` +
 `(lambda↑(k) "foo " "bar " loquefaltaporhacerdeString-append)`
+ `"boo"`

(porque la función principal es `string-append` y éstos son sus argumentos)

= `(k ("foo " + "bar " + "boo"))`

(la `k` está implícita)

= `"foo bar boo"`

Cuando solo tenemos la continuación en un programa:

```
1      (display  
2        (call/cc (lambda (k)  
3          (display "I got here.\n")  
4          (k "This string was passed to the continuation.\n")  
5          (display "But not here.\n")))))
```


Salida del programa:

I got here.

This string was passed to the continuation.

Otro ejemplo:

```
> (+ 10 (let/cc k 32))
```

42

¿Cuál es la función **lambda↑** asociada?

```
(+ 10 (lambda↑ (k)
```

```
  (k (+ contexto-ant 32)))
```

```
)
```

```
(+ 10 (lambda↑(k)
```

```
  (k (+ 10 32 ))) )
```

```
= (k (+ 10 32))
```

```
= (k 42)
```

```
= 42
```

Otro ejemplo

> (+ 10 (let/cc k (+ 1 (k 2))))

¿Cuál es el resultado de la expresión anterior?

Otro ejemplo

```
> (+ 10 (let/cc k (+ 1 (k 2))))
```

12

¿Por qué?

Es una suma

Tiene 2 lados: izquierdo = 10 y derecho = continuación i.e. let/cc

Otro ejemplo

```
> (+ 10 (let/cc k (+ 1 (k 2))))
```

12

Del lado derecho ¿dónde está la k?

(k 2)

Si hay una suma pero la k no está inmediatamente en todo el cuerpo de la continuación eso sería (let/cc k

(k (+ 1 ...)))

Otro ejemplo

```
> (+ 10 (let/cc k (+ 1 (k 2))))
```

12

Evaluación:

```
(+ 10 (let/cc k (+ 1 (k 2))))
```

```
(+ 10 (lambda↑(•) (+ 1 (k (+ • 2))) ) )
```

```
(+ 10 (lambda↑(10) (+ 1 (k (+ 10 2))) ) ) = (k 12) = 12
```

Ejercicio 1:

(+ 1
(let/cc k
 (k 5)))

= 6

(+ 1
(lambda↑ (k)
 (k (+ 1 5))))

Solo evaluamos la continuación

(k (+ 1 5)) = (k 6) = 6

Ejercicio 2:

```
(+ 1  
  (let/cc k  
    (- 5 (k 5))))
```

= 6

```
(+ 1  
  (lambda↑ (k)  
    (- 5 (k (+ 1 5))) ) )
```

Solo evaluamos la continuación

$(k (+ 1 5)) = (k 6) = 6$

Ejercicio 3:

> (+ 1 (+ 2 (let/cc k

(+ 10 1))))

= 14

(+ 1 (+ 2 (let/cc k

(+ 10 1))))

(+ 1 (+ 2

(lambda↑ (k)

(+ 1 (+ 2 (+ 10 1)))))

Solo evaluamos la continuación

(k (+ 1 (+ 2 (+ 10 1)))) = (k 14) = 14

Ejercicio 4:

```
> (+ 1 (+ 2 (let/cc k (k 5) (+ 10 1))))
```

8

```
(+ 1 (+ 2 (let/cc k (k 5) (+ 10 1))))
```

Tenemos la *k* explícita y tenemos las sumas como operaciones que contienen una continuación

```
= (+ 1 (+ 2
```

```
(lambda↑ (k)
```

```
(k contexto-ant 5 )
```

```
(+10 1) )))
```

```
= (+ 1 (+ 2
```

```
(lambda↑ (k)
```

```
(k (+ 1 (+ 2 5 )))
```

```
(+ 10 1) )))
```

Ejercicio 4:

= (+ 1 (+ 2

(lambda↑ (k)

(k (+ 1 (+ 2 5)))

(+ 10 1))))

= (k (+ 1 (+ 2 5))))

= (k (+ 1 7))

= (k 8) = 8

Una continuación más: ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
  "boo" )
```



```
(string-append
```

```
  "foo "
```

```
  continuación: call/cc
```

```
  "bar ")
```

Una continuación más: ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
  "boo" )
```

Esta es una variable llamada **saved** que está inicializada con el valor de **#f**

¿Cuál es el código asociado con la notación de lambda↑ ?

Siguiendo el ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
  "boo" )
```



La k está implícita:

```
(string-append
```

```
  "foo "
```

```
  (lambda↑ (k)
```

```
    (k (set! saved k )
```

```
      "bar ")))
```

```
  "boo")
```

Siguiendo el ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
  "boo" )
```



```
saved = #f
```

```
(string-append
```

```
  "foo "
```

```
  (lambda↑ (k)
```

```
    (k (string-append
```

```
      "foo "
```

```
      (set! saved k )
```

```
      "bar "
```

```
      "boo"))))
```

```
    "boo")
```

Siguiendo el ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
  "boo" )
```

```
(string-append
```

```
  "foo"
```

```
  (lambda↑ (k)
```

```
    (k (string-append
```

```
      "foo "
```

```
      (set! saved k )
```

```
      "bar "
```

```
      "boo" ) ))
```

```
    "boo")
```


Siguiendo el ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
"boo")
```

```
(string-append
```

```
  "foo"
```

```
  (lambda↑ (k)
```

```
    (k (string-append
```

```
      "foo "
```

```
      (set! saved k )
```

```
      "bar "
```

```
      "boo" ) ))
```

```
"boo")
```

Siguiendo el ejercicio 5

```
(define saved #f)
```

```
(string-append
```

```
  "foo "
```

```
  (call/cc (lambda (k)
```

```
    (set! saved k)
```

```
    "bar " ))
```

```
"boo")
```

```
(string-append  
  "foo"
```

```
(continuación: call/cc "bar"  
y asignación a var. saved=k)
```

```
  "boo")
```

```
= "foo bar boo"
```

Siguiendo el ejercicio 5

```
(saved "BAR ")
```

```
( (lambda↑ (k)
```

```
  (k (string-append
```

```
    "foo "
```

```
    (set! saved k )
```

```
    " " )
```

```
    "boo" ) ) "BAR ")
```

```
(saved "BAR ")
```

```
(saved es k)
```

¿Cuál k?

= String-append de

1. "foo"
2. continuación ahora recibe BAR
3. "boo"

= "foo BAR boo"

Gracias

¿Dudas?