Universidad Nacional Autónoma de México

Facultad de Ciencias



# Lenguajes de Programación

Tarea 7: Continuation Passing Style (CPS)

Johann Ramón Gordillo Guzmán
418046090

José Jhovan Gallardo Valdéz
310192815

Tarea presentada como parte del curso de **Lenguajes de Programación** impartido por la profesora **M.I. Karla Ramírez Pulido**.

13 de Mayo del 2020

**Link al código fuente:** https://github.com/JohannGordillo/

# 1. Preguntas

1. Define con tus propias palabras y en no más de cuatro renglones el concepto de *continuación*.

   **Respuesta.**

   Las continuaciones son funciones que guardan el contexto de toda la ejecución de un programa. Son usadas para optimizar la complejidad en espacio de los algoritmos recursivos, y son diferentes de los acumuladores usados en recursión de cola, en el sentido de que un acumulador solo guarda resultados parciales de haber evaluado una función, y una continuación guarda el contexto de la función.

2. Explica el funcionamiento de las primitivas `call/cc` y `let/cc` de Racket y da un ejemplo de uso de cada una.

   **Respuesta.**

   - call/cc.

     *call/cc* es una abreviatura para *call-with-current-continuation*.

     Toma como parámetro la continuación actual $k$ y puede usar esta función en su cuerpo para obtener algún valor. En otras palabras, toma una captura del contexto actual de un programa como un objeto y le aplica la función que toma como parámetro. Cuando una continuación es aplicada a un argumento, la continuación existente es eliminada y la continuación aplicada es reemplazada en su lugar, así el flujo de ejecución del programa continuará hasta el punto en el que el argumento de la continuación se vuelve el valor de regreso de la llamada a función de *call/cc*.

     Por ejemplo:

     (+ 1729 (* 4 (+ 1 (call/cc (λ (k) (+ 2 (k 1)))))))

     que se reducirá a (+ 1729 (* 4 (+ 1 1))) = (+ 1729 (* 4 2)) = (+ 1729 8) = 1737.

     Obsérvese que no se toma en cuenta el +2, pues se evalúa la aplicación de función (k 1) donde k es una función (λ (·) (+1729 (* 4 (+ 1 ·))). Es decir, se evalúa:

     ((λ (·) (+1729 (* 4 (+ 1 ·))) 1)
     (+1729 (* 4 (+ 1 1)))
     (+ 1729 (* 4 2))
     (+ 1729 8)
     1737

     ```
     > (+ 1729 (* 4 (+ 1 (call/cc (λ (k) (+ 2 (k 1)))))))
     1737
     >
     ```

- let/cc.

  Esta primitiva funciona como una asignación local cuyo identificador es la continuación actual y en cuyo cuerpo se describe qué hacer con ésta.

  Analizando más a fondo el comportamiento de esta función, tenemos que una expresión de la forma (let/cc k e) capturará la continuación actual, representada como una función, y ligará la variable k a la continuación capturada para finalmente evaluar la expresión e.

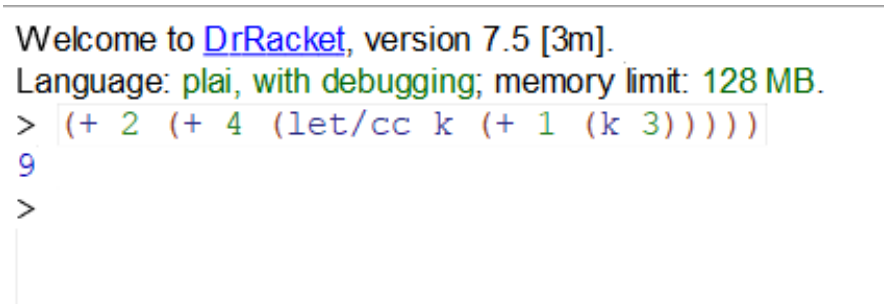
  Por ejemplo:

  (+ 2 (+ 4 (let/cc k (+ 1 (k 3)))))

  La continuación asociada a evaluar la expresión anterior es: (λ ↑ (v) (+ 2 (+ 4 v))), y cuando hacemos (k 3) lo que estamos haciendo es aplicar la función (λ ↑ (v) (+ 2 (+ 4 v))) con parámetro 3, de la siguiente manera:

  ((λ ↑ (v) (+ 2 (+ 4 v))) 3)
  (+ 2 (+ 4 3))
  (+ 2 7)
  9

  Podemos comprobar el resultado en Racket:



3. Evalúa el siguiente código en Racket, explica su resultado, y da la continuación asociada a evaluar, usando la notación $\lambda\uparrow$.

```
> (define c #f)

> (+ 1 (+ 2  (+ 3 (+ (let/cc k (set! c k) 4) 5))))

> (c 10)
```

**Respuesta.**

Al ejecutar el código anterior en Racket, obtenemos:

```
;;=================================================================
;; Ejercicio de continuación asociada.
;;=================================================================

(define c #f)
(+ 1 (+ 2  (+ 3 (+ (let/cc k (set! c k) 4) 5))))
(c 10)
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
15
21
>
```

Este resultado es sencillo de explicar. En la primera línea de código lo que hacemos es asociar la variable global $c$ con el valor booleano false. Esto con el fin únicamente de inicializarla.

Posteriormente, ejecutamos:

$$(+\ 1\ (+\ 2\ (+\ 3\ (+\ (\text{let/cc k (set! c k)}\ 4)\ 5))))$$

Donde la continuación asociada es:

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

Esta continuación será guardada en la variable global $c$, de manera que podamos usarla posteriormente.

Continuando con la ejecución del programa, vemos que la expresión es de la forma (let/cc k e), donde $e$ es 4, y al aplicar la función ($\lambda \uparrow$ (v) (+ 1 (+ 2 (+ 3 (+ v 5))))) con este parámetro obtenemos:

```
((λ↑ (v) (+ 1 (+ 2 (+ 3 (+ v 5))))) 4)
(+ 1 (+ 2 (+ 3 (+ 4 5))))
(+ 1 (+ 2 (+ 3 9)))
(+ 1 (+ 2 12))
(+ 1 14)
15
```

Resultado que coincide con el obtenido usando Racket.

La última línea de código a evaluar es:

```
(c 10)
```

Como en $c$ se guardó la continuación asociada ($\lambda \uparrow$ (v) (+ 1 (+ 2 (+ 3 (+ v 5))))), obtenemos:

```
((λ↑ (v) (+ 1 (+ 2 (+ 3 (+ v 5))))) 10)
(+ 1 (+ 2 (+ 3 (+ 10 5))))
(+ 1 (+ 2 (+ 3 15)))
(+ 1 (+ 2 18))
(+ 1 20)
21
```

Resultado que coincide con el obtenido usando Racket.

4. Modifica cada una de las siguientes funciones de forma que usen el estilo de paso de continuaciones (*Continuation Passing Style*).

**Respuesta.**

*a*) (define (potencia n m)
(if (zero? m)
1
(* n (potencia n (sub1 m)))))

**Respuesta.**

Mi implementación en Racket es:

```
#|-----------------------------------------------------------------
 | potencia: number number -> number
 |-----------------------------------------------------------------
 | Devuelve 'n' a la potencia 'm'.
 |-----------------------------------------------------------------
 |#
(define (potencia n m)
  (potencia-cps n m (λ (x) x)))

#|-----------------------------------------------------------------
 | potencia-cps: number number procedure -> number
 |-----------------------------------------------------------------
 | Función auxiliar para la función 'potencia' usando el
 | estilo de paso de continuaciones.
 |-----------------------------------------------------------------
 |#
(define (potencia-cps n m k)
  (if (zero? m)
      (k 1)
      (potencia-cps n (sub1 m) (λ (v) (* n (k v))))))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
> (potencia 2 3)
8
> (potencia 5 43)
113686837721616029739379882812 5
> |
```

*b*) (define (suma-digitos n)
(if (< n 10)
n
(+ (modulo n 10) (suma-digitos (quotient n 10)))))

**Respuesta.**

Mi implementación en RACKET es:

```
#|-----------------------------------------------------------------
 | suma-digitos: number -> number
 |-----------------------------------------------------------------
 | Devuelve la suma de los dígitos que componen al número
 | de entrada 'n'.
 |-----------------------------------------------------------------
 |#
(define (suma-digitos n)
  (suma-digitos-cps n (λ (x) x)))

#|-----------------------------------------------------------------
 | suma-digitos-cps: number procedure -> number
 |-----------------------------------------------------------------
 | Función auxiliar para la función 'suma-digitos' usando el
 | estilo de paso de continuaciones.
 |-----------------------------------------------------------------
 |#
(define (suma-digitos-cps n k)
  (if (< n 10)
      (k n)
      (suma-digitos-cps (quotient n 10)
                        (λ (v) (+ (modulo n 10) (k v))))))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
> (suma-digitos 29)
11
> (suma-digitos 11111)
5
>
```

*c*) (define (cuadrados lst)
(if (empty? lst)
empty
(cons (expt (car lst) 2) (cuadrados xs))))

**Respuesta.**

Mi implementación en RACKET es:

```
#|-----------------------------------------------------------------
 | cuadrados: (listof number) -> (listof number)
 |-----------------------------------------------------------------
 | Devuelve una lista con los elementos de la lista de
 | entrada, pero elevados al cuadrado.
 |-----------------------------------------------------------------
 |#
(define (cuadrados lst)
  (cuadrados-cps lst (λ (x) x)))

#|-----------------------------------------------------------------
 | cuadrados-cps: (listof number) procedure -> number
 |-----------------------------------------------------------------
 | Función auxiliar para la función 'cuadrados' usando el
 | estilo de paso de continuaciones.
 |-----------------------------------------------------------------
 |#
(define (cuadrados-cps lst k)
  (if (empty? lst)
      (k empty)
      (cuadrados-cps (cdr lst)
                     (λ (v) (append (k v)
                                    (list (expt (car lst) 2)))))))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
> (cuadrados '(1 2 3 4 5))
'(1 4 9 16 25)
> (cuadrados '(6 7 8 9 10))
'(36 49 64 81 100)
>
```

*d*) (define (reversa lst)
(if (empty? lst)
empty
(append (reversa (cdr lst)) (list x))))

**Respuesta.**

Mi implementación en RACKET es:

```
#|-------------------------------------------------------------------
 | reversa: (listof any) -> (listof any)
 |-------------------------------------------------------------------
 | Devuelve una lista con los elementos de la lista de
 | entrada, pero en reversa.
 |-------------------------------------------------------------------
 |#
(define (reversa lst)
  (reversa-cps lst (λ (x) x)))

#|-------------------------------------------------------------------
 | reversa-cps: (listof any) procedure -> (listof any)
 |-------------------------------------------------------------------
 | Función auxiliar para la función 'reversa' usando el
 | estilo de paso de continuaciones.
 |-------------------------------------------------------------------
 |#
(define (reversa-cps lst k)
  (if (empty? lst)
      (k empty)
      (reversa-cps (cdr lst)
                   (λ (v) (append (list (car lst)) (k v))))))
```

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
> (reversa '(1 2 3 4 5))
'(5 4 3 2 1)
> (reversa '(a b c d e))
'(e d c b a)
>
```

5. Dada la siguiente definición del predicado `primo?` que decide si un número positivo mayor a 2 es primo. Modifícala usando memoización.

```
(define (primo? n)
    (if (equal? n 2)
        #t
        (aux n 2 (sub1 n))))


(define (aux n i j)
    (cond
        [(> i j) #t]
        [(zero? (modulo n i)) #f]
        [else (aux n (add1 i) j)]))
```

**Respuesta.**

Mi implementación en Racket es la siguiente:

```
#|-----------------------------------------------------------------
 | primo?: number -> boolean
 |-----------------------------------------------------------------
 | Decide si un número positivo mayor a 2 es primo.
 |-----------------------------------------------------------------
 |#
(define (primo? n)
  (primo-memo n (make-hash (list (cons 2 #t)))))

(define (primo-memo n tbl)
  (let ([busqueda (hash-ref! tbl n 'vacia)])
    (cond
      [(equal? busqueda 'vacia)
       (define nuevo (aux n 2 (sub1 n)))
       (hash-set! tbl n nuevo)
       nuevo]
      [else busqueda])))

(define (aux n i j)
  (cond
    [(> i j) #t]
    [(zero? (modulo n i)) #f]
    [else (aux n (add1 i) j)]))

(test (primo? 1) #t)
(test (primo? 2) #t)
(test (primo? 25) #f)
(test (primo? 26) #f)
(test (primo? 37) #t)
```

Podemos comprobar que el algoritmo funciona con base en las pruebas implementadas:

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
good (primo? 1) at line 42
  expected: #t
  given: #t

good (primo? 2) at line 43
  expected: #t
  given: #t

good (primo? 25) at line 44
  expected: #f
  given: #f

good (primo? 26) at line 45
  expected: #f
  given: #f

good (primo? 37) at line 46
  expected: #t
  given: #t

>
```

Si tuvieramos que implementar memoización para la función auxiliar, podríamos hacerlo de la siguiente manera:

```
(define (primo? n)
  (primo-memo n (make-hash (list (cons 2 #t)))))

(define (primo-memo n tbl)
  (let ([busqueda (hash-ref! tbl n 'vacia)])
    (cond
      [(equal? busqueda 'vacia)
       (define nuevo (aux n 2 (sub1 n)))
       (hash-set! tbl n nuevo)
       nuevo]
      [else busqueda])))

(define (aux n i j)
  (aux-memo n i j (make-hash empty) (make-hash empty)))

(define (aux-memo n i j tbl1 tbl2)
  (let ([busqueda (hash-ref! tbl1 i 'vacia)])
    (cond
      [(equal? busqueda 'vacia)
       (define nuevo
         (cond
           [(> i j) #t]
           [(zero? (modulo n i)) #f]
           [else (aux-memo n (add1 i) j tbl1 tbl2)]))
       (hash-set! tbl2 j nuevo)
       (hash-set! tbl1 i tbl2)
       nuevo]
      [else (let ([busqueda2 (hash-ref! busqueda j 'vacia)]) busqueda2)])))
```

Podemos comprobar que de nuevo se pasan las pruebas unitarias:

```
Welcome to DrRacket, version 7.5 [3m].
Language: plai, with debugging; memory limit: 128 MB.
good (primo? 1) at line 62
  expected: #t
  given: #t

good (primo? 2) at line 63
  expected: #t
  given: #t

good (primo? 25) at line 64
  expected: #f
  given: #f

good (primo? 26) at line 65
  expected: #f
  given: #f

good (primo? 37) at line 66
  expected: #t
  given: #t

>
```

Si decidieramos no usar las funciones `aux-memo` y `primo-memo`, podemos implementarlo como sigue:

```
(define tbl (make-hash (list (cons 2 #t))))
(define tbl1 (make-hash empty))

(define (primo? n)
  (let ([busqueda (hash-ref! tbl n 'vacia)])
    (cond
      [(equal? busqueda 'vacia)
       (define nuevo (aux n 2 (sub1 n)))
       (hash-set! tbl n nuevo)
       nuevo]
      [else busqueda])))

(define (aux n i j)
  (let ([busqueda (hash-ref! tbl1 i 'vacia)])
    (cond
      [(equal? busqueda 'vacia)
       (define nuevo
         (cond
           [(> i j) #t]
           [(zero? (modulo n i)) #f]
           [else (aux n (add1 i) j)]))
       (hash-set! tbl1 i (make-hash (list (cons j nuevo))))
       nuevo]
      [else (let ([busqueda2 (hash-ref! busqueda j 'vacia)])
              (cond
                [(equal? busqueda2 'vacia)
                 (define nuevo
                   (cond
                     [(> i j) #t]
                     [(zero? (modulo n i)) #f]
                     [else (aux n (add1 i) j)]))
                 (hash-set! tbl1 i (make-hash (list (cons j nuevo))))
                 nuevo]
                [else busqueda2]))])))
```

Que igual pasa las pruebas unitarias.

# 2. Bibliografía


- Ramírez, K. (2020).
  *Notas del curso de Lenguajes de Programación.*
  Facultad de Ciencias - UNAM
  Ciudad de México, México.

- Krishnamurthi, S. (2017).
  *Programming Languages: Application and Interpretation.*
  Estados Unidos de América.