



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 6

INTEGRANTES

Torres Valencia Kevin Jair - 318331818
Aguilera Moreno Adrián - 421005200
Rivera Silva Marco Antonio - 318183583

PROFESORA

Karla Ramírez Pulido

AYUDANTES

Alan Alexis Martínez López
Manuel Ignacio Castillo López
Alejandra Cervera Taboada

ASIGNATURA

Lenguajes de Programación

25 de noviembre de 2022

1. Explica con tus propias palabras y el concepto de continuación.

Un claro ejemplo para explicar el concepto, es como lo que sucede en la película "Feliz día de tu muerte", en donde la protagonista de la película muere exactamente el día de su cumpleaños, pero esta al morir, revive el mismo día desde que despertó (día que muere), pero esta recuerda exactamente lo que vivió en el día por decirlo así el día anterior, por lo que cada que revive esta hace diferentes actividades/acciones para evitar que la maten, así hasta descubrir a su asesino.

Entonces nuestra protagonista por decirlo así en cada uno de sus días es una continuación, en donde, cada continuación conoce/sabe el contexto de las anteriores, por lo que en cada una sabe "que hacer o no hacer", en cada uno de estos días.

2. Explica con tus propias palabras el funcionamiento de las primitivas `call/cc` y `let/cc` del lenguaje de programación Racket y da un ejemplo de uso de cada una.

▷ **Solución:** El uso de `call/cc` y `let/cc` se reducen al mismo fin. No existe diferencia en el resultado al implementar estas funciones, sin embargo existen ciertas diferencias sutiles, como lo son

1. *Sintaxis.* Al usar `call/cc` tenemos que hacer uso de una lambda explícita, mientras que con `let/cc` no es necesario.
2. *Semántica.* En cuanto a semántica la diferencia es que con `let/cc` la continuación esta ligada a nuestra variable `k` (ya sea explícita o implícita). Al usar `call/cc` una `k` que puede ser implícita o explícita, pero funge como la continuación (a diferencia de `let/cc`).

A continuación se da una función escrita con `call/cc` y `let/cc`:

- Función con base en la primitiva `let/cc`:

```
(define (f n)
  (* 10
    (/ 5
      (let/cc k
        (k (- 2
              (+ 1 n))))))))
```

-
- Función con base en la primitiva `call/cc`:

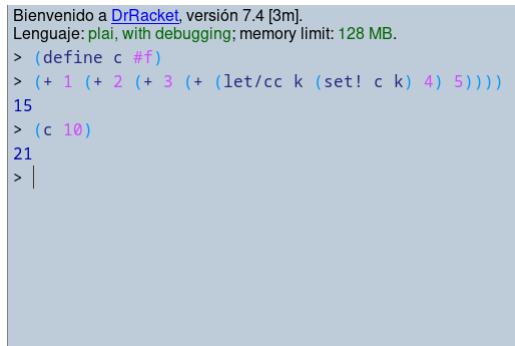
```
(define (f n)
  (* 10
    (/ 5
      (call/cc
        (lambda (k)
          (k (- 2
                (+ 1 n))))))))
```

En ambos casos se hace uso del “contexto” de la función, por lo que sus resultados al ser usadas serán el mismo. ◁

3. Evalúa el siguiente código en el lenguaje de programación 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)
```

Se anexa captura de pantalla, sobre el resultado arrojado en Racket.



```

Bienvenido a DrRacket, versión 7.4 [3m].
Lenguaje: plai, with debugging; memory limit: 128 MB.
> (define c #f)
> (+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
15
> (c 10)
21
> |

```

Se evalúa de la siguiente forma:

■ Notación $\lambda \uparrow$ (Racket).

```
(+ 1 (+ 2 (+ 3 (+ (let/cc k ( set! c k) 4) 5))))
(+ 1 (+ 2 (+ 3 (+ ( $\lambda \uparrow$  (v) (v (+ 1 (+ 2 (+ 3 (+ 4 v )) 5))))))))
(+ 1 (+ 2 (+ 3 (+ ( $\lambda \uparrow$  (v) (v (+ 1 (+ 2 (+ 3 (+ 4 5 ))))))))))
(v (+ 1 (+ 2 (+ 3 (+ 4 5))))))
(v 15) = 15
```

Se obtiene 15 por que no se le pasa a la continuación el valor de $(c\ 10)$ (con un argumento), por lo que este hace que Racket no lo considere para la evaluación, haciendo que Racket solo tome la "continuación actual", entonces la continuación de c (en un inicio almacenando $\#f$) ahora contendrá a 4.

■ Notación $\lambda \uparrow$ (Se pasa $(c\ 10)$).

```
(+ 1 (+ 2 (+ 3 (+ (let/cc k ( set! c k) 4) 5))))
(+ 1 (+ 2 (+ 3 (+ ( $\lambda \uparrow$  (v) (v (+ 1 (+ 2 (+ 3 (+ 10 v )) 5))(c 10))))))
(+ 1 (+ 2 (+ 3 (+ ( $\lambda \uparrow$  (v) (v (+ 1 (+ 2 (+ 3 (+ 10 5 ))))))))))
(v (+ 1 (+ 2 (+ 3 (+ 10 5))))))
(v 21) = 21
```

Se obtiene 21 por que se le pasa a la continuación el valor de $(c10)$ (con un argumento), por lo que este hace que Racket lo considere para la evaluación de la continuación.

4. Observa la siguiente función del lenguaje de programación Racket.

```
(let ([fib (lambda (n) (if (or (zero? n) (= n 1)) 1 (+ (fib (- n 1)) (fib 3))
```

- a. Prueba la expresión en el intérprete de Racket y con base en la respuesta obtenida, explica el proceso que siguió el intérprete para llegar a ésta. Anexa una captura de pantalla del intérprete de Racket al probar la expresión.

Figura 1: Ejecución de la función dada.

```
> (let ([fib (lambda (n) (if (or (zero? n) (= n 1)) 1 (+ (fib (- n 1)) (fib (- n 2))))) (fib 3))
= (+ (fib (- 3 1)) (fib (- 3 2)))
= (+ (+ (fib (- 2 1)) (fib (- 2 2))) (fib 1))
= (+ (+ (1) (1)) (1))
= (+ 2 1)
= 3
```

- b. Modifica la función usando el Combinador de Punto Fijo Y. Prueba la expresión en el intérprete de Racket y con base en la respuesta obtenida, explica el proceso que siguió el intérprete para llegar a ésta. Anexa una captura de pantalla del intérprete de Racket al probar la expresión.

Figura 2: Fib con combinador de punto fijo Y.

La anterior es una función *mutuamente recursiva* que implementa el combinador de punto fijo Y. La segunda parte de la función (la que define a Y)

realiza la sustitución de punto fijo Y hasta llegar al respectivo valor y lo utiliza para no generar tantos registros en memoria, la ejecución interna se basa en realizar las reducciones a la 2da expresión y aplicarlas a la primera.

- c. Modifica la función usando el Combinador de Punto Fijo Z. Prueba la expresión en el intérprete de Racket y con base en la respuesta obtenida, explica el proceso que siguió el intérprete para llegar a ésta. Anexa una captura de pantalla del intérprete de Racket al probar la expresión.

```

18
19 (let ((Z (λ (fib)
20         ((λ (x) (fib (λ (v) ((x x) v))))
21         (λ (x) (fib (λ (v) ((x x) v))))))))
22     (let ((fib-Z (λ (fib)
23                   (λ (n)
24                     (if (or (zero? n) (= n 1))
25                         1
26                         (+ (fib (- n 1)) (fib (- n 2)))))))
27         (let ((fib (Z fib-Z)))
28             (fib 3))))
29
Welcome to DrRacket, version 8.5 [cs].
Language: plai, with debugging; memory limit: 128 MB.
3
> |

```

Figura 3: Fib con combinador de punto fijo Z.

La ejecución de esta función es parecida a la anterior, pero obedece al regimen $Z(f) = f(Z(f))$ que lo que esta escrito en la línea 27 del código. La ejecución consta de las reducciones de las lambdas que realizan recursión mutua con la función fib.

5. *Punto extra*

Observa la siguiente función del lenguaje de programación Racket

```
(let ([sum (lambda (n) (if (zero? n) 0 (+ n (sum (sub1 n))))))])
  (sum 5))
```

- a. Prueba la expresión en el intérprete de Racket y con base en la respuesta obtenida, explica el proceso que siguió el intérprete para llegar a ésta. Anexa una captura de pantalla del intérprete de Racket al probar la expresión.

```

> (letrec ([sum (λ (n) (if (zero? n) 0 (+ n (sum (sub1 n)))))])
  (sum 5))
15
>

```

Figura 4: Ejecución de la función dada.

```

> (let ([sum (lambda (n) (if (zero? n) 0 (+ n (sum (sub1 n))))))
  (sum 5))
= (+ 5 (sum 4))

```

$$\begin{aligned}
&= (+ \ 5 \ (+ \ 4 \ (\text{sum } 3))) \\
&= (+ \ 5 \ (+ \ 4 \ (+ \ 3 \ (\text{sum } 2)))) \\
&= (+ \ 5 \ (+ \ 4 \ (+ \ 3 \ (+ \ 2 \ (\text{sum } 1))))) \\
&= (+ \ 5 \ (+ \ 4 \ (+ \ 3 \ (+ \ 2 \ (+ \ 1 \ 0))))) \\
&= (+ \ 5 \ (+ \ 4 \ (+ \ 3 \ (+ \ 2 \ (1))))) \\
&= (+ \ 5 \ (+ \ 4 \ (+ \ 3 \ (3)))) \\
&= (+ \ 5 \ (+ \ 4 \ (+ \ 6))) \\
&= (+ \ 5 \ (10)) \\
&= 15
\end{aligned}$$

- b. Modifica la función usando el Combinador de Punto Fijo Y .Prueba la expresión en el intérprete de Racket y con base en la respuesta obtenida, explica el proceso que siguió el intérprete para llegar a ésta. Anexa una captura de pantalla del intérprete de Racket al probar la expresión.

```

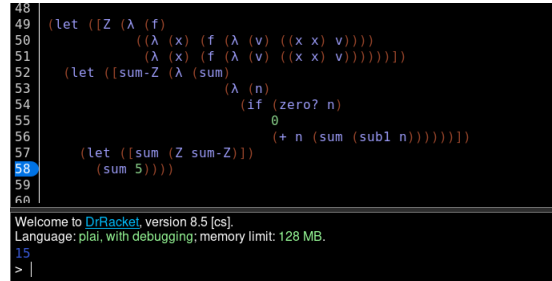
34 (let ([Y (λ (sum)
35           (λ (n)
36             (if (zero? n)
37                 0
38                 (+ n (sum (sub1 n))))))])
39   (let ([sum ((λ (Y)
40                 (λ (v)
41                   ((λ (sum) (Y (λ (v) ((sum sum) v))))
42                    (λ (sum) (Y (λ (v) ((sum sum) v)))) v)))
43         Y)])
44     (sum 5)))
45
46
47
Welcome to DrRacket, version 8.5 [cs].
Language: plai, with debugging; memory limit: 128 MB.
15
>

```

Figura 5: Ejecución de la función dada.

Para esta función, tenemos una definición interna que realiza una recursión mutua, la ejecución tiene que reducir las lambdas a su forma normal (en este caso siempre es un valor numérico) y luego pasarselas a la función anterior o siguiente en el orden.

-
- c. Modifica la función usando el Combinador de Punto Fijo Z. Prueba la expresión en el intérprete de Racket y con base en la respuesta obtenida, explica el proceso que siguió el intérprete para llegar a ésta. Anexa una captura de pantalla del intérprete de Racket al probar la expresión.



```
48 (let ([Z (λ (f)
49           ((λ (x) (f (λ (v) ((x x) v))))
50           (λ (x) (f (λ (v) ((x x) v)))))))]
51   (let ([sum-Z (λ (sum)
52                   (λ (n)
53                     (if (zero? n)
54                         0
55                         (+ n (sum (sub1 n))))))]
56         (let ([sum (Z sum-Z)])
57           (sum 5))))
58   sum)
59 RA
```

Welcome to DrRacket, version 8.5 [cs].
Language: plai, with debugging; memory limit: 128 MB.
15
> |

Figura 6: Ejecución de la función dada.

Como en el ejercicio anterior, en la línea 27 se encuentra la modificación que vuelve a esta función en un combinador de punto fijo Z. Inicialmente reduce las lambdas a sus formas normales y ejecuta ambas funciones de manera *mutuamente recursiva* hasta llegar al valor deseado.