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

24 de noviembre de 2022

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

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

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

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.

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.

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.

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.

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.