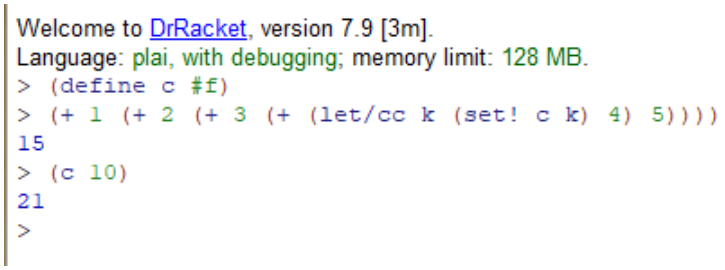
Facultad de Ciencias, UNAM
Lenguajes de Programación
Tarea 6

Rodríguez Campos Erick Eduardo
Rubí Rojas Tania Michelle

23 de enero de 2021

1. Evalúa el siguiente código en Racket, explica su resultsado, 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)
```



Cuando invocamos una continuación, esta nos envía de regreso al punto en el programa donde se evaluó la expresión marcada, la continuación toma un argumento, que remplaza el valor de la expresión marcada y la evaluación se reanuda a partir de ahí. Tenemos que *let/cc* hace tres cosas:

a) Sirve como la expresión marcada por la continuación.

b) Captura la continuación (*cc* significa "continuación actual") y la asigna a una variable que designamos, en este caso k.

c) Como el *let*, evalúa cualquier expresión dentro y la última expresión se convierte en el valor de retorno. En este caso, *let/cc* guarda nuestra continuación en la variable c, y luego devuelve 4 (ya que c tiene como valor #f). Por lo tanto el valor de la expresión es 15.

Notación $\lambda\uparrow$:

```
(+ 1 (+ 2 (+ 3 (+ (let/cc k (set! c k) 4) 5))))
(+ 1 (+ 2 (+ 3 (+ (lambda ↑ (k) (k (+ 1 (+ 2 (+ 3 (+ 4 5)))))))))))
(k (+ 1 (+ 2 (+3 (+ 4 5)))))
(k 15) = 15
```

Cuando invocamos nuestra continuación c con un argumento, es equivalente a reemplazar el valor de 4 por el valor de c, y así el valor de la expresión es 21.

Notación $\lambda\uparrow$:

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

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

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

   Modificar la función usando la técnica de recursión de cola:

   ```
   (define (potencia n m)
       (potencia-tail n m 1))

   (define (potencia-tail n m acc)
       (if (zero? m)
           acc
           (potencia-tail n (sub1 m) (* n acc))))
   ```

   Transformar la función usando CPS:

   ```
   (define (potencia n m)
       (potencia/k n m ((λ (x) x))))

   (define (potencia/k n m k)
       (if (zero? m)
           (k 1)
           (potencia/k n (sub1 m) (λ(v) (k (* n v))))))
   ```

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

   Modificar la función usando la técnica de recursión de cola:

   ```
   (define (suma-digitos n)
       (suma-digitos-tail n 0))

   (define (suma-digitos-tail n acc)
       (if (= n 0)
           acc
           (sumar-digitos-tail (quotient n 10) (+ acc (modulo n 10)))))
   ```

   Transformar la función usando CPS:

```
(define (suma-digitos n)
    (suma-digitos/k n (λ (x) x)))

(define (suma-digitos/k n k)
    (if (= n 0)
        (k 0)
        (suma-digitos/k (quotient n 10) (+ λ(v) (k (+ v (modulo n 10)))))))
```

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

Modificar la función usando la técnica de recursión de cola:

```
(define (cuadrados lst)
    (cuadrados-tail lst '()))

(define (cuadrados-tail lst acc)
    (if (empty? lst)
        acc
        (cuadrados-tail (cdr lst) (cons acc (expt (car lst) 2)))))
```

Transformar la función usando CPS:

```
(define (cuadrados lst)
    (cuadrados/k lst (λ (x) x)))

(define (cuadrados/k lst k)
    (if (empty? lst)
        (k empty)
        (cuadrados/k (cdr lst) (λ (v) (k (cons (expt (car lst) 2) v))))))
```

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

Modificar la función usando la técnica de recursión de cola:

```
(define (reversa lst)
    (reversa-tail lst '()))

(define (reversa-tail lst acc)
    (if (empty? lst)
        acc
        (reversa-tail (cdr lst) (cons acc (car lst)))))
Transformar la función usando CPS:


(define (reversa lst)
    (reversa/k lst (λ (x) x)))
```

```
(define (reversa/k lst k)
    (if (empty? lst)
        (k empty)
        (reversa/k (cdr lst) (λ (v) (k (append v (list (car lst))))))))
```

3. Evalúa la siguiente expresión usando el paso de parámetros que se indica.

```
{with {{a 8}
       {b -8}
       {swap {fun {x y}
                  {with {{tmp x}}
                     {seqn {set x y}
                           {set y tmp}}}}}}
   {seqn {swap a b}
         {- a {+ b a}}}}
```

(*a*) Paso de parámetros por valor.
Ambiente:

swap	{fun {x y} {with {{tmp x}} {seqn {set x y} {set y tmp}}}}	0x12
b	-8	0x11
a	8	0x10

Evaluación de {swap a b}:

```
{{fun {x y} {with {{temp x}} seqn {set x y} {set y temp}}} 8 -8}
{with {{temp 8}} {seqn {set x -8} {set y tmp}}}
{seqn {set x -8} {set y 8}}
{set x -8}      entonces x = -8
{set y 8}       entonces y = 8
```

Evaluación de {-a {+ b a}}:

```
{- 8 {+ b 8}}
{- 8 {+ -8 8}}
{- 8 {0}} = 8
```

(*b*) Paso de parámetros por referencia.
Ambiente:

swap	{fun {x y} {with {{tmp x}} {seqn {set x y} {set y tmp}}}}	0x12
b	-8	0x11
a	8	0x10

Evaluación de {swap a b}:

```
{{fun {x y} {with {{temp x}} seqn {set x y} {set y temp}}} 8 -8}
{with {{temp 8}} {seqn {set x -8} {set y tmp}}}
{seqn {set x -8} {set y 8}}
{set x -8}      entonces x = -8
{set y 8}       entonces y = 8
```

Ambiente modificado:

`swap`	`{fun {x y} {with {{tmp x}} {seqn {set x y} {set y tmp}}}}`	0x12
`b`	`8`	0x11
`a`	`-8`	0x10

Evaluación de `{-a {+ b a}}`:

```
{- -8 {+ b -8}}
{- -8 {+ 8 -8}}
{- -8 {0}} = -8
```

4. Evalúa la siguiente expresión usando el paso de parámetros que se indica.

```
{with {{a 1}
       {foo {fun {x}
                 {seqn {set a {+ {* x -2} a}}
                 a}}}}
   {{fun {y} {+ y {- y y}} {foo 3}}}}
```

(a) Paso de parámetros por nombre.
Ambiente:

`foo`	`{fun {x} {seqn {set a {+ {* x -2} a}} a}}`	0x11
`a`	`1`	0x10

Evaluación de `{{fun {y} {+ y {- y y}}} {foo 3}}`:

```
{+ {foo 3} {- {foo 3} {foo 3}}}
```

Evaluación de la primera ocurrencia de `{foo 3}`:

```
{{fun {x} {seqn {set a {+ {* x -2} a}} a}} 3}
{seqn {set a {+ {* 3 -2} a}} a}
{set a {+ -6 1}}     entonces a = -5
```

Ambiente modificado:

`foo`	`{fun {x} {seqn {set a {+ {* x -2} a}} a}}`	0x11
`a`	`-5`	0x10

Evaluación de la segunda ocurrencia de `{foo 3}`:

```
{{fun {x} {seqn {set a {+ {* x -2} a}} a}} 3}
{seqn {set a {+ {* 3 -2} a}} a}
{set a {+ -6 -5}}     entonces a = -11
```

Ambiente modificado:

`foo`	`{fun {x} {seqn {set a {+ {* x -2} a}} a}}`	0x11
`a`	`-11`	0x10

Evaluación de la tercera ocurrencia de `{foo 3}`:

```
{{fun {x} {seqn {set a {+ {* x -2} a}} a}} 3}
{seqn {set a {+ {* 3 -2} a}} a}
{set a {+ -6 -11}}     entonces a = -17
```

Ambiente modificado:

`foo`	`{fun {x} {seqn {set a {+ {* x -2} a}} a}}`	0x11
`a`	`-17`	0x10

Evaluamos `{+ {foo 3} {- {foo 3} {foo 3}}}` con los resultados de las evaluaciones de cada ocurrencia de `{foo 3}`:

```
{+ -5 {- -11 -17}}
{+ -5 6} = 1
```

(*b*) Paso de parámetros por necesidad.
Ambiente:

`foo`	`{fun {x} {seqn {set a {+ {* x -2} a}} a}}`	0x11
`a`	`1`	0x10

Evaluación de `{{fun {y} {+ y {- y y}}} {foo 3}}`:

```
{+ {foo 3} {- {foo 3} {foo 3}}}
```

Evaluación de la primera ocurrencia de `{foo 3}`:

```
{{fun {x} {seqn {set a {+ {* x -2} a}} a}} 3}
{seqn {set a {+ {* 3 -2} a}} a}
{set a {+ -6 1}}     entonces a = -5
```

Evaluamos `{+ {foo 3} {- {foo 3} {foo 3}}}` con el resultado de la evaluación de `{foo 3}`:

```
{+ -5 {- -5 -5}}
{+ -5 0} = -5
```

5. Evalúa la siguiente expresión usando el paso de parámetros por referencia-regreso

```
{with {{b 2}
       {f {fun {x} {seqn {set x 4}
                         {+ x b}}}}}
   {+ {f b} b}}
```

Ambiente:

`f`	`{fun {x} {seqn {set x 4} {+ x b}}}`	0x11
`b`	2	0x10

Evaluación de `{f b}`

```
{{fun {x} {seqn {set x 4} {+ x b}}} b}
{seqn {set x 4} {+ x b}}    entonces x = 4
{+ 4 b}
{+ 4 2} = 6
```

Ambiente modificado:

f	{fun {x} {seqn {set x 4} {+ x b}}}	0x11
b	4	0x10

Evaluación de `{+ {f b} b}` con el resultado de la evaluación de `{ f b}`:

```
{+ 6 b}
{+ 6 4} = 10
```

6. Dada la siguiente del 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)]))
```

Modificación usando memoización:

```
(define tbl (make-hash (list '(2 #t))))

(define (primo? n)
    (primo-nemo? n 2 (sub1 n) tbl/h))

(define (primo-nemo? n i j tbl/h)
    (let ([busqueda (hash-ref! tbl/h n 'vacia)])
        (match busqueda
            [(list z) z]
            ['vacia #:when (> i j) #t]
            ['vacia #:when (zero? (modulo n i)) #f]
            [else (let ([nuevo (primo-nemo? n (addl i) j tbl/h)])
                    (hash-set! tbl/h n nuevo) nuevo)])))
```