Lenguajes de Programación *Juan Carlos Faz Leal*
Tarea 04: Ejercicios de PROC
11 de octubre del 2022

---

1. Ejercicio 3.20 [*] En PROC, los procedimientos tienen solo un argumento, pero se puede obtener el efecto de procedimientos de múltiples argumentos utilizando procedimientos que devuelven otros procedimientos. Por ejemplo, uno podría escribir código como

```
let f = proc (x) proc (y) ...
in ((f 3) 4)
```

   Este truco se llama Currificar , y se dice que el procedimiento está Currificado. Escriba un procedimiento Currificado que tome dos argumentos y devuelva su suma. Puedes escribir x + y en nuestro idioma escribiendo -(x , -(0, y)).

   Sabemos que el lenguaje PROC se compone de dos cosas; una variable y un cuerpo. Dado que el enunciado nos dice que el procedimiento tiene dos argumentos, significa que habrá dos variables, con esto nos podemos imaginar como si tuviéramos que usar proc dos veces. proc(x) y proc(y), y estos deben retornar la suma.

```
proc (x)
    proc (y)
        -(x, -(0, y))
```

   Dados los componentes de proc mencionados anteriormente asignamos la variable a “x” y otro procedimiento a su vez en el cuerpo del primer proc, asignamos la variable “y”, donde el cuerpo del segundo proc retorna la suma de “x” y “y” según el ejercicio.

2. Ejercicio 3.27 [*] Agregue un nuevo tipo de procedimiento llamado traceproc a al idioma. A traceproc funciona exactamente como proc, excepto que imprime un mensaje de rastreo al entrar y al salir.

   El problema nos dice que debemos añadir un nuevo tipo de procedimiento llamado traceproc al lenguaje, este funciona exactamente igual que proc, tiene una variable y un cuerpo, excepto que este imprime un mensaje de rastreo al aplicar y al salir del procedimiento.

   Como proc y traceproc tienen los mismos terminales, entonces nuestra sintaxis concreta nos queda:

```
Expression := traceproc (Identifier) Expression
```

   En nuestra sintaxis abstracta tenemos algo como:

```
(traceproc-exp var body)
```

Pareciera que nada cambia, pero el problema nos dice que debemos imprimir un mensaje de rastreo al aplicar y al salir.

Entonces debemos modificar nuestro apply-procedure, para cuando lea un procedimiento entra al procedimiento, y depues de obtener el valor del cuerpo en el entorno, salimos.

```
(define (apply-procedure proc val)
  (unless (procedure? proc)
    (error 'value-of "no es un procedimiento: e" proc))
  (let ([var (procedure-var proc)]
        [body (procedure-body proc)]
        [saved-env (procedure-saved-env proc)])
    (display "Entering procedure.")
    (value-of body (extend-env var val saved-env))
    (display "Exiting procedure.")))
```

3. Ejercicio 3.23 [**] ¿Cuál es el valor del siguiente programa PROC?

```
let makemult = proc (maker)
                proc (x)
                 if zero?(x)
                 then 0
                 else -(((maker maker) -(x,1)), -4)
in let times4 = proc (x) ((makemult makemult) x)
   in (times4 3)
```

Usa los trucos de este programa para escribir un procedimiento para factorial en PROC. Como sugerencia, recuerde que puede usar Currificación (ejercicio 3.20) para definir un procedimiento de dos argumentos times.

Por el nombre que se le da al identificador del let me suena a que estan haciendo una multiplicacion. El segundo let se llama: times4, como si fuera un operador que dijera: cuatro veces, pero cuatro veces que? Un numero x.

En el ejemplo el x que utilizamos es 3, entonces diremos que es 4 veces el 3, lo cual da como resultado 12. Esto se logra llamando recursivamente al maker con maker, donde cada vez que se manda a llamar, antes se pregunta si el x es 0, si es asi se detiene y devuelve 0, sino, se ejecuta el else, donde se resta uno a la x y se suma un cuatro al resultado, por lo tanto para la siguiente iteracion la x es 2 y la multiplicacion va en 4, vuelve a iterar hasta que la x sea igual a 0.

El valor del programa es: 12

Y el procedimiento para el factorial:

```
let factorial = proc (func)
                    proc (x)
                        if zero?(x)
                        then 1
                        else *(((func func) -(x,1)), x)
in let fac = proc (x) ((factorial factorial) x)
    in (fac 3)
```

4. Ejercicio 3.24 [**] Usa los trucos del programa anterior para escribir el par de procedimientos mutuamente recursivos, odd y even, como en el ejercicio 3.32.

   El problema me dice que debo usar los trucos de llamada recursiva, por lo tanto se debe parecer algo a los problemas anteriores, algo como: (even even) o como (odd odd).

   Voy a tener dos banderas para saber si un numero es par o impar.

```
False = zero?(1).
True = zero?(0).
```

   Estas banderas son las que retornaremos para saber si un numero es par o impar.

   El problema nos dice que son procesos, por lo tanto tendremos estos dos procesos representativos del problema:

```
Proc (makeeven)
Proc (makeodd)
```

   Y obvio no puede faltar el numero evaluado x.

```
Proc (x)
```

   Entonces con estas herramientas podemos resolver el problema.

   odd:

```
let false = zero?(1)
in let true = zero?(0)
   in let makeeven = proc (makeeven)
                       proc (makeodd)
                         proc (x)
                           if zero?(x)
                           then true
                           else (((makeodd makeeven) makeodd) -(x, 1))
      in let makeodd = proc (makeeven)
                         proc (makeodd)
                           proc (x)
                             if zero?(x)
                             then false
                             else (((makeeven makeeven) makeodd) -(x, 1))
         in ((makeodd makeeven) makeodd)
```

Como utilizamos la recursividad, la usamos para hacer saltos entre los procesos.

Ejemplo: Nos dan el número 2, conocemos este numero y sabemos que no es impar, por lo tanto deberia ser falso que sea impar.

El programa va a restar la x hasta que el numero se haga 0 y este debe retornar su respectiva bandera, en este caso debe retornar falso.

Como queremos saber si un numero es impar, entonces se llama recursivamente a makeodd en el cuerpo del let, y se pregunta si el numero x es igual a 0, como el 2 no es igual a 0, entonces se le resta 1 y se llama recursivamente a makeeven. Entra al proceso makeeven y se pregunta si el x es igual a 0, como nuestro x es igual a 1 entonces se le resta otra vez 1 a la x y se manda a llamar a makeodd, como ahora la x es igual a 0 entonces el programa entra al if y devuelve false.

Por tanto el 2 no es impar.

Para hacer el Even solo cambiamos la llamada en el cuerpo del let

even:

```
let false = zero?(1)
in let true = zero?(0)
   in let makeeven = proc (makeeven)
                       proc (makeodd)
                         proc (x)
                           if zero?(x)
                           then true
                           else (((makeodd makeeven) makeodd) -(x, 1))
      in let makeodd = proc (makeeven)
                         proc (makeodd)
                           proc (x)
                             if zero?(x)
                             then false
                             else (((makeeven makeeven) makeodd) -(x, 1))
         in ((makeeven makeeven) makeodd)
```

5. Ejercicio 3.25 [*] Los trucos de los ejercicios anteriores se pueden generalizar para mostrar que podemos definir cualquier procedimiento recursivo en PROC. Considere el siguiente fragmento de código:

```
let makerec = proc (f)
                let d = proc (x)
                          proc (z) ((f (x x)) z)
                in proc (n) ((f (d d)) n)
in let maketimes4 = proc (f)
                      proc (x)
                       if zero?(x)
                       then 0
                       else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

Demuestre que devuelve 12.

maketimes4 es un procedimiento que toma un procedimiento times4 y devuelve un procedimiento times4. Primero convertimos maketimes4 en un procedimiento maker que toma un maker y devuelve un procedimiento times4 (supongamos que usamos f para representar maketimes4):

```
proc (f)
  let maker = proc (maker)
                let recursive-proc = (maker maker)
                in (f recursive-proc)
  in ...
```

Pero el código no funcionaría porque una vez que llamamos (maker maker), primero llamará a (maker maker), lo que provocará una recursividad infinita. Arreglaremos esto envolviendo (maker maker) dentro de otro procedimiento:

```
proc (f)
  let maker = proc (maker)
                proc (x)
                  let recursive-proc = (maker maker)
                  in ((f recursive-proc) x)
  in ..
```

Ahora obtenemos un maker, llamamos al maker con maker, obtendremos una versión recursiva de f:

```
proc (f)
  let maker = proc (maker)
                proc (x)
                  let recursive-proc = (maker maker)
                  in ((f recursive-proc) x)
  in (maker maker)
```

Quedando el programa asi:

```
let makerec = proc (f)
                let maker = proc (maker)
                              proc (x)
                                let recursive-proc = (maker maker)
                                in ((f recursive-proc) x)
                in (maker maker)
in let maketimes4 = proc (f)
                      proc (x)
                        if zero?(x)
                        then 0
                        else -((f -(x, 1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

El resultado es: 12.