Examen 3: Lenguajes de Programación

Oscar Andrés Rosas Hernández [SoyOscarRH@ciencias.unam.mx | 417024956]
Facultad de Ciencias, UNAM, CDMX, México

Primer Problema

(2 pts.) Evalua las siguientes expresiones usando cada uno de los pasos de parametros que se solicitan, debes de poner la última expresión a evaluar antes de dar el resultado final de la misma, usando:

```
{with*
  {
    {i -1}
    {j -1}
    {swap {fun {x y}
      {seqn
        {set tmp x}
        {set x y}
        {set y tmp} }}}}

  {seqn
    {swap i j}
    {- j i}}}
```

- Paso de parámetros por valor
  Esta interesante, siendo simplistas:
  - Declaramos una variable $i = -1$
  - Declaramos una variable $j = -1$
  - declaramos una funcion swap
  - ejecutamos una fn swap (no nos importa que haga porque porque al ser paso por valor y una funcion sin side effects [bueno, para ser exactos, si tiene pues afecta a tmp, esperemos que tmp este definido en otra parte del programa], nada se verá afectado fuera de la funcion).
  - hacemos $j - i = -1 - -1 = -1 + 1 = 0$ y ese ese el resultado de la evaluación
  - Ahora, si nos ponemos a mirar que hace swap:
    - creamos una variable llamada x con el valor de i, una llamada y con el valor de b.
    - hacemos que tmp valga x (osea tmp = -1)
    - hacemos que x valga y (osea x = -1)
    - hacemos que y valga tmp (osea y = -1)
- Paso de parámetros por referencia
  Ya que aunque estamos usando paso de parametros por refencia y el swap se comporta como esperamos tenemos el gran problema de que como ambas variables valen lo mismo el swap no hara nada tampoco.
  - Declaramos una variable $i = -1$
  - Declaramos una variable $j = -1$
  - declaramos una funcion swap
  - ejecutamos una fn swap.
  - creamos uno nombre para la variable i (es decir pasamos su direccion de memoria) llamada x, y equivalentemente otro nombre para j, y. con el valor de b.

  - hacemos que tmp valga x (osea tmp = -1)
  - hacemos que x valga y (osea x = -1)
  - hacemos que y valga tmp (osea y = -1)
- hacemos $j - i = -1 - -1 = -1 + 1 = 0$ y ese ese el resultado de la evaluación

Importante mencionar que sin importar el tipo de paso de parametros es valor es el mismo:

ID	Valor
swap	`{fun {x y} ...}`
j	-1
i	-1

SEGUNDO PROBLEMA

(2 pts.) ¿Cuales son las diferencias principales entre el paso de parametros por necesidad y por nombre?

Son cosas bastante diferentes, el paso por parametros por necesidad esta muy asociado a la idea de la memoización, la idea es que si tienes la aplicacion de una funcion en un mismo contexto y con los mismos argumentos varias veces entonces no ejecutemos el codigo de la funcion varias veces sino solo 1 vez.

Pasar por nombre es muy parecido a pasar algo por referencia, pero la diferencia con este es que al pasar algo por referencia los parametros son evaluados antes de empezar el cuerpo de la funcion, mientras que al pasar por nombre es mas como uan substitucion textual en que el valor del parametro es ejecutado solo cuando y si lo necesitemos.

Por lo tanto pocas cosas en comun conceptualmente tienen que ver ambos pasos de parametros, una idea sencilla seria que si usamos paso de parametros por necesidad y tenemos varias aplicaciones a funciones en un mismo contexto y con los mismos argumentos entonces solo se ejecutara el cuerpo de la funcion una 1, mientras que usando paso por nombre se hara varias veces.

TERCER PROBLEMA

(1 pts.) Convierte el codigo anterior a CPS

Listo n.n

```
#lang plai
(define identity (lambda (x) x))
(define (filter-neg l) (filter-neg/k l identity))

(define (filter-neg/k numbers k)
    (cond
      [(empty? numbers) (k empty)]
      [(negative? (first numbers))
        (filter-neg/k (rest numbers) (lambda (others) (k (cons (first numbers) others))))]
      [else (filter-neg/k (rest numbers) k)]))
```

(1 pt.) ¿Qué regresa la función que convertiste a CPS cuando recibe la lista '(0 1 -1 0 -4 1 -2)?

Sencillo, regresa una lista con solo numeros negativos, es decir $'(-1 - 4 - 2)$

CUARTO PROBLEMA

(2 pts.) Da la expresion asociada a la continuacion y el resultado de dicha expresion, para cada uno de los siguientes codigos:

-
```
(define c #f)

(+ 1 (+ 2 (+ 3 (+
  (let/cc here (set! c here) 4) 5))))
```
Sencilla, primero que nada el resultado de la expresion es 15. Y la expresion asociada a la continuacion, que en ese caso esta guardada en c es equivalente a la funcion; `(lambda (v) (+ 1 (+ 2 (+ 3 (+ v 5)))))`
-
```
(define c empty)
(+ 1 (+ 2 (+ 3 (+ (let/cc here (set! c here) 4) 5))))
(c 20)
```
Sencilla, primero que nada el resultado de la expresion es 15. Y la expresion asociada a la continuacion, que en ese caso esta guardada en c es equivalente a la funcion; `(lambda (v) (+ 1 (+ 2 (+ 3 (+ v 5)))))`
Por eso es que `(c 20) = 31`

SEXTO PROBLEMA

(2 pts.) Realiza la inferencia de tipos de la siguiente expresion, mencionando al termino de la inferencia, los tipos de cada una de las variables de la funcion

```
(define foo
 1(lambda (lst item)
    2(cond
      3((nempty? nlst) nempty)
      4((nequal? item (nfirst lst)) (nrest lst))
      5(else (ncons (nfirst lst) (foo (nrest lst) item))))))
```

No me dio tiempo de poner todos los pasos pero la idea es:

- para 3 nempty recibe una lista de numeros y en este caso regresa una lista de numeros vacia
- para 4 nequal recibe dos numeros, por lo tanto item y el primer elemento de lst son numeros y esta expresion regresa nrest que recibe una lista y regresa una lista
- para 5 regresamos una nueva lista, donde el primer elemento es el primer elemento de lst y la concatenamos con el resultado de la llamada recursiva
- por lo tanto en 3, 4, 5, regresamos una lista de numeros, por lo tanto 2 regresa una lista de numeros, por lo tanto podemos ya dar los tipos de la funcion, lst es una lista de numeros, item un numero y regresa una lista de numeros

OCTAVO PROBLEMA

(2 pts.) Da las sentencias de variables de tipo para las siguientes funciones de Racket:

- $list - length : \forall t. \mid list(t) -> number$
- $fibonacci : number -> number$

NOVENO PROBLEMA

(2 pts.) Selecciona 2 caracteristicas de la siguiente lista del Paradigma Orientado a Objetos:

- **Encapsulamiento:** Recordemos que las clases son como una caja cerrada, todos los métodos y las variables están dentro y toda la información solo puede ser accedido por medio de sus sistemas propios, esto hace que sea de verdad como una caja cerrada, su información solo entra y sale por las formas y de la forma en que la clase quiera. Puede incluso no importante como funciona por dentro, todo lo que necesitas saber es que funciona.
  Por ejemplo si tenemos una clase que se llama Socket, tendremos metodos como enviar o recibe, pero no nos importara como es que esta implementado, la clase es responsable de manejar su estado interno y nos permite acceder a el solo por los metodos que la misma considera apropiado.
- **Herencia o Jerarquía:** Esto es clave tambien, la herencia permite que una clase pueda servir como plantilla para la creación de futuras clases.
  Es el mecanismo por el cual una clase permite heredar las características (atributos y métodos) de otra clase. Nos ayuda con la reutilizacion de codigo, pues permite que se puedan definir nuevas clases basadas de unas ya existentes, generando así una jerarquía de clases dentro de una aplicación.
  Si una clase deriva de otra, esta hereda sus atributos y métodos y puede añadir nuevos atributos, métodos o redefinir los heredados.
  Importante mencionar por ejemplo el gran Principio de Sustitución de Liskov:
  Nos dice que si en alguna parte de nuestro código estamos usando una clase, y esta clase es extendida, tenemos que poder utilizar cualquiera de las clases hijas y que el programa siga siendo válido.

DECIMO PROBLEMA

(2 pts.) Se tiene el siguiente código que intenta dar una versión memoizada de la función que calcula el n-esimo numero de la sucesión de Tribonacci:

```
(define (tribonacci n)
  (if (< n 3)
    1
    (+ (tribonacci (- n 1)) (tribonacci (- n 2)) (tribonacci (- n 3)))))

(define (tribonacci-memo n tabla)
  (let ([res (hash-ref tabla n â ninguno)])
    (cond
      [(equal? res â ninguno)
      (hash-set! tabla n (tribonacci n))
      (has-ref tabla n)]
      [else res])))
```

Pues funciona muy bien si es que tenemos una tabla hash global que siempre pasamos como parametro a nuestra funcion, pero no es lo mas comun al usar memoización, pues queremos conservar la firma de la funcion a los usuarios de la misma, una implementacion que no cambiara la firma de la funcion seria algo como:

```
(define (tribonacci n)
  (if (< n 3)
    1
    (+ (tribonacci (- n 1)) (tribonacci (- n 2)) (tribonacci (- n 3)))))

(define tribonacci-vals (make-hash))
(define (tribonacci-memo n)
  (let ([val (hash-ref tribonacci-vals n empty)])
    (cond
      [(empty? val)
```

```
        (hash-set! tribonacci-vals n (tribonacci n))
        (has-ref tribonacci-vals n)]
      [else val])))
```