Soluciones Parcial 2 LPP 2011-2012

Examen mañana

Pregunta 2

a) Diseño e implementación de la barrera de abstracción

El tipo de dato `circulo` se define a partir de su radio (número real) y de las coordenadas x e y (también números reales) de su centro. Definimos su constructor y sus selectores con esos elementos. Utilizaremos el sufijo circ para todas las funciones del tipo de datos:

Diseño

`(make-circ r x y)`: devuelve un círculo creado a partir de un radio y las coordenadas x e y de su centro. Todos los parámetros deben ser números reales.
`(r-circ circulo)`: devuelve el radio (número real) del círculo
`(x-circ circulo)`: devuelve la coordenada x (número real) del centro del círculo
`(y-circ circulo)`: devuelve la coordenada y (número real) del centro del círculo

Definimos dos funciones que trabajan con círculos:

`(bbox-circ c1)`: devuelve una lista con las cuatro coordenadas del cuadrado en el que está inscrito el círculo (bounding-box): (xmin,ymin,xmax,ymax)
`(desplaza-circ circulo inc-x inc-y)`: devuelve un círculo nuevo resultante de desplazar el `original` inc-x e inc-y (números reales)

Implementación

```
;; constructor
(define (make-circ r x y)
  (cons r (cons x y)))

;; selectores
(define (r-circ circ)
  (car circ))
(define (x-circ circ)
  (car (cdr circ)))
(define (y-circ circ)
  (cdr (cdr circ)))

;; bounding box, devuelve (x-min y-min x-max y-max)
(define (bbox-circ circ)
  (let ((r (r-circ circ))
        (x (x-circ circ))
        (y (y-circ circ)))
    (list (- x r) (- y r) (+ x r) (+ y r))))

;; devuelve un nuevo circulo desplazando el original
(define (desplaza-circ circ inc-x inc-y)
  (make-circ (r-circ circ)
```

```
            (+ (x-circ circ) inc-x)
            (+ (y-circ circ) inc-y)))
```

b) Bounding box de una lista de círculos

```
;; devuelve la suma de dos bounding-box (xmin ymin xmax ymax)
(define (suma-bounding-box bbox1 bbox2)
  (list
     (min (list-ref bbox1 0) (list-ref bbox2 0))
     (min (list-ref bbox1 1) (list-ref bbox2 1))
     (max (list-ref bbox1 2) (list-ref bbox2 2))
     (max (list-ref bbox1 3) (list-ref bbox2 3))))

;; devuelve el bounding-box de una lista de círculos
(define (bounding-box lista-circ)
  (if (null? (cdr lista-circ))
      (bbox-circ (car lista-circ))
      (suma-bounding-box (bbox-circ (car lista-circ))
                         (bounding-box (cdr lista-circ)))))
```

c) La solución es recursiva porque todas las llamadas a `suma-bounding-box` se quedan a la espera del final de la recursión.

Pregunta 3

```
(define (suma-exp-s exp-s1 exp-s2)
   (cond
      ((null-exp-s? exp-s1) '())
      ((leaf-exp-s? exp-s1) (list (+ exp-s1 exp-s2)))
      (else (append (suma-exp-s (first-exp-s exp-s1) (first-exp-s exp-s2))
                    (suma-exp-s (rest-exp-s exp-s1) (rest-exp-s exp-s2))))))
```

Pregunta 4

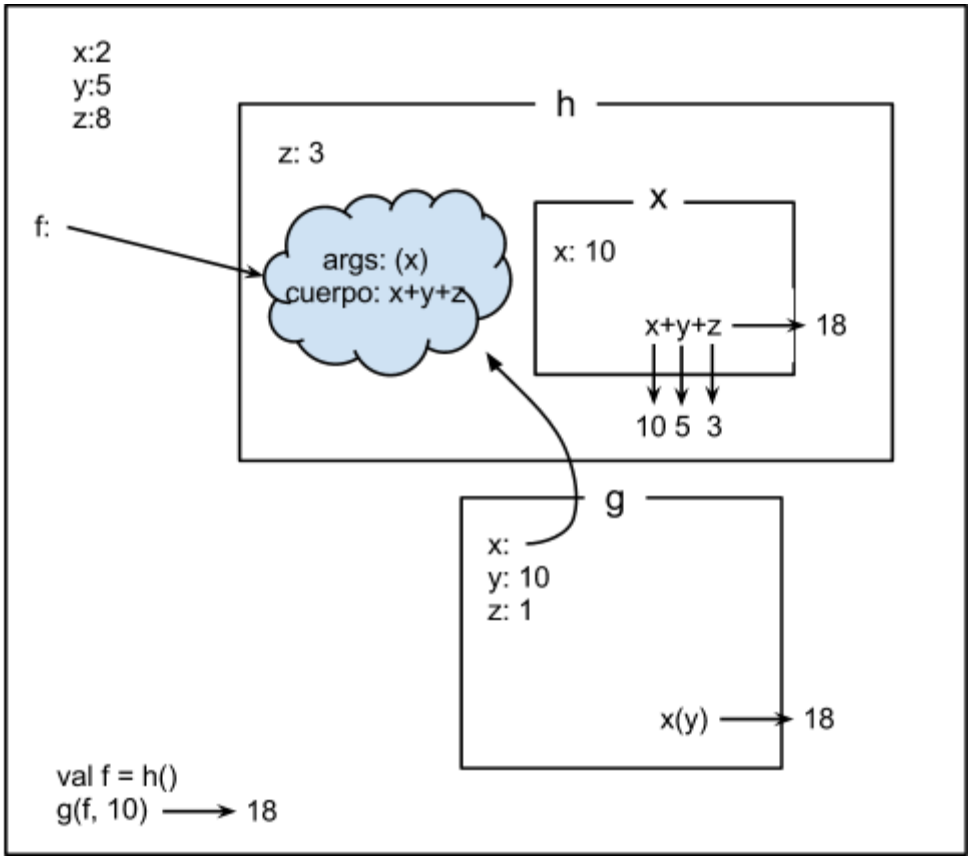
```
 (define (camino-btree btree lista)
    (cond
      ((null? lista) '())
      ((equal? (car lista) '=)
           (cons (dato-bt btree) (camino-btree btree (cdr lista))))
      ((equal? (car lista) '<)
           (camino-btree (izq-bt btree) (cdr lista) ))
      ((equal? (car lista) '>)
           (camino-btree (der-bt btree) (cdr lista) ))))
```

Pregunta 5

```
(define (palabra-tree? tree lista)
    (cond ((null? lista) #f)
              ((and (equal? (dato-tree tree) (car lista))
                        (hoja-tree? tree)
                        (null? (cdr lista))) #t))
              ((equal? (dato-tree tree) (car lista))
                        (palabra-bosque? (hijos-tree tree) (cdr lista)))
              (else #f)))
```

```
(define (palabra-bosque? bosque lista)
     (if (null? bosque) #f
         (or (palabra-tree? (car bosque) lista)
              (palabra-bosque? (cdr bosque) lista))))
```

Pregunta 6



Explicación: Las primeras sentencias dan valor a las variables *x,y,z y* definen las funciones *h* y *g*. La sentencia *val f = h()* invoca la función *h*. Esta invocación crea un ámbito en el que se define la variable *z* con el valor 3 y se crea una función anónima que se devuelve como resultado. La función queda asignada a la variable *f*. La última sentencia invoca a la función *g* pasando como parámetro la función guardada en el la variable *f* y el valor 10. Se crea un nuevo ámbito en el que se ejecuta la función *g*. Los parámetros de la función quedan ligados a los argumentos (el parámetro *x* referencia a la función anónima y el parámetro *y* toma el valor 10). En el nuevo ámbito se define la variable *z* con el valor 1 y se llama a la función *x* con el valor *y* 10. La invocación de *x* crea un nuevo ámbito dentro del ámbito en el que se definió la función (*deep binding*). En este nuevo ámbito el parámetro *x* de la función toma el valor 10 y se evalúa la expresión *x+y+z*, que devuelve 18.

Pregunta 7

```
def parejasNums(x:Int):List[(Int,Int)] = {
       for (y <- List.range(0,x+1); z <- List.range(0,x+1)) yield (y,z)
}
```

```
def aplicaFuncionDosArgsLista(f:(Int,Int)=>Int,lista:List[(Int,Int)]):List[Int] =
{
	if (lista.isEmpty) Nil
	else f(lista.head._1, lista.head._2) ::
aplicaFuncionDosArgsLista(f,lista.tail)
}
```

Examen tarde

Pregunta 2

a) Diseño e implementación de la barrera de abstracción

El tipo de dato `rectángulo` se define a partir de las coordenadas x e y (números reales) de su esquina inf. izquierda, de su ancho y de su alto (también números reales). Definimos su constructor y sus selectores con esos elementos. Utilizaremos el sufijo rect para todas las funciones del tipo de datos:

Diseño

`(make-rect x y ancho alto)`: devuelve un rectángulo creado a partir de una coordenada x e y de su esquina inferior izquierda, de su ancho y de su alto. Todos los parámetros deben ser números reales.
`(x-rect rect)`: devuelve la coord. x (número real) de la esquina inf. izq. del rectángulo.
`(y-rect rect)`: devuelve la coord. y (número real) de la esquina inf. izq. del rectángulo.
`(ancho-rect rect)`: devuelve el ancho del rectángulo (número real)
`(alto-rect rect)`: devuelve el alto del rectángulo (número real)

Definimos dos funciones que trabajan con círculos:

`(union-rect r1 r2)`: devuelve el rectángulo resultante de unir dos
`(desplaza-rect rect inc-x inc-y)`: devuelve un rectángulo nuevo resultante de desplazar el rectángulo inc-x e inc-y (números reales)

Implementación

```
;; constructor
(define (make-rect x y ancho alto)
  (list x y ancho alto))

;; selectores
(define (x-rect rect)
  (list-ref rect 0))
(define (y-rect rect)
  (list-ref rect 1))
(define (ancho-rect rect)
  (list-ref rect 2))
(define (alto-rect rect)
  (list-ref rect 3))

;; union de dos rectángulos
(define (union-rect r1 r2)
  (let ((x1 (x-rect r1))
        (x2 (x-rect r2))
        (y1 (y-rect r1))
        (y2 (y-rect r2)))
    (make-rect (min x1 x2)
               (min y1 y2)
               (max (+ x1 (ancho-rect r1))
                    (+ x2 (ancho-rect r2)))
```

```
              (max (+ y1 (alto-rect r1))
                   (+ y2 (ancho-rect r2))))))

;; devuelve un nuevo rectángulo desplazando el original
(define (desplaza-rect rect inc-x inc-y)
  (make-rect (+ (x-rect rect) inc-x)
             (+ (y-rect rect) inc-y)
             (ancho-rect rect)
             (alto-rect rect)))
```

b) Bounding box de una lista de rectángulos

```
;; devuelve el bounding-box de una lista de círculos
(define (bounding-box lista-rect)
  (if (null? (cdr lista-rect))
      (car lista-rect)
      (union-rect (car lista-rect)
                  (bounding-box (cdr lista-rect)))))
```

c) La solución es recursiva porque todas las llamadas a `union-rect` se quedan a la espera del final de la recursión.

Pregunta 3

```
(define (aplanar-exp-s f exp-s)
   (cond
      ((null-exp-s? exp-s) '())
      ((leaf-exp-s? exp-s) (list (f exp-s)))
      (else (append (aplanar-exp-s f (first-exp-s exp-s))
                    (aplanar-exp-s f (rest-exp-s exp-s))))))
```

Pregunta 4

```
 (define (palabra-btree btree palabra)
    (cond
      ((null? palabra) (vacio-bt? btree))
      ((vacio-bt? btree) #f)
      ((equal? (car palabra) (dato-bt btree))
          (or (palabra-btree (izq-bt btree) (cdr palabra))
              (palabra-btree (der-bt btree) (cdr palabra))))
      (else #f)))
```

Pregunta 5

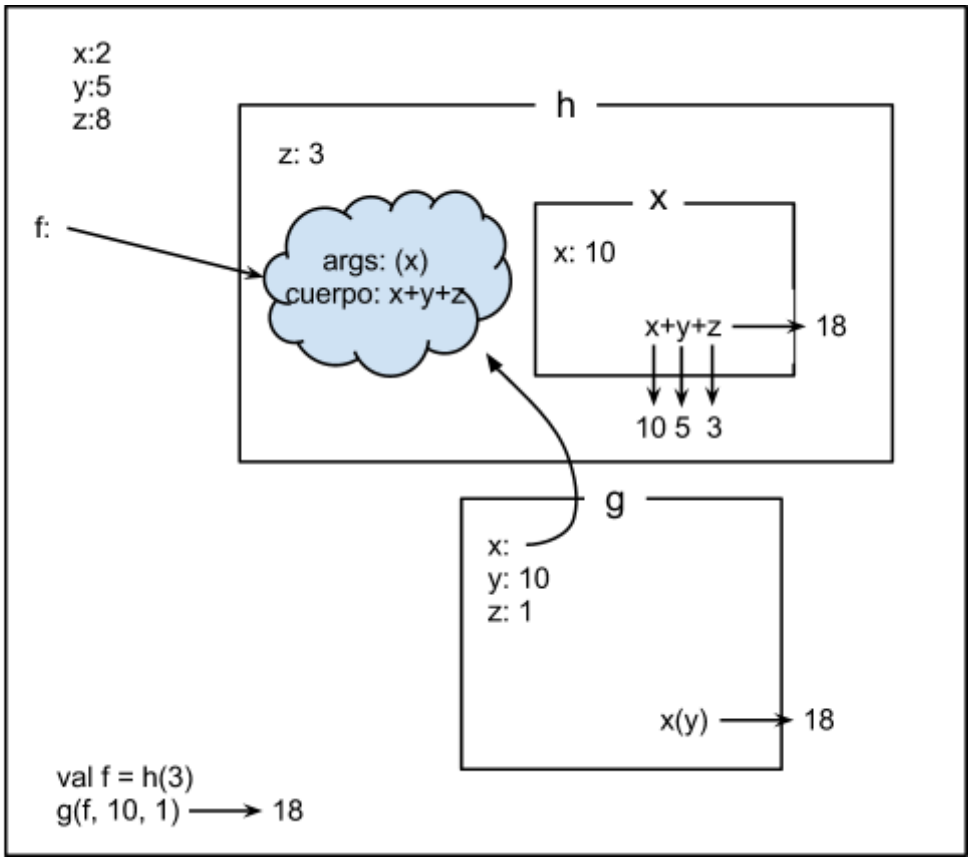
```
(define (camino-max-tree tree)
(if (null? tree) '()
    (cons (dato-tree tree)
          (camino-max-tree (camino-max-bosque (hijos-tree tree) )))))

(define (camino-max-bosque bosque)
(if (null? bosque) '()
    (buscar-max-bosque bosque (apply max (map dato-tree bosque)))))

(define (buscar-max-bosque bosque maximo)
```

```
(if (null? bosque) '()
(if (= (dato-tree (car bosque)) maximo)
   (car bosque)
   (buscar-max-bosque (cdr bosque) maximo))))
```

Pregunta 6



Explicación: Las primeras sentencias dan valor a las variables *x,y,z y* definen las funciones *h* y *g*. La sentencia *val f = h(3)* invoca la función *h* pasando como parámetro el valor 3. Esta invocación crea un ámbito en el que el parámetro *z* toma el valor 3 y se crea una función anónima que se devuelve como resultado. La función queda asignada a la variable *f*. La última sentencia invoca a la función *g* pasando como parámetro la función guardada en el la variable *f* y los valores 10 y 1. Se crea un nuevo ámbito en el que se ejecuta la función *g*. Los parámetros de la función quedan ligados a los argumentos (el parámetro *x* referencia a la función anónima, el parámetro *y* toma el valor 10 y el parámetro *z* el valor 1). En el nuevo ámbito se llama a la función *x* con el valor *y* 10. La invocación de *x* crea un nuevo ámbito dentro del ámbito en el que se definió la función (*deep binding*). En este nuevo ámbito el parámetro *x* de la función toma el valor 10 y se evalúa la expresión *x+y+z*, que devuelve 18.

Pregunta 7

```
def creaBaraja() = {
   val palos = List("Oros", "Copas", "Espadas", "Bastos")
   for (y <- palos ; z <- List.range(1,13)) yield (z,y)
}
```

```
def filtraBaraja(baraja:List[(Int,String)], f:(Int,String)
=>Boolean):List[(Int,String)] = {
   if (baraja.isEmpty) Nil else
if (f(baraja.head._1, baraja.head._2) == true)
 baraja.head::filtraBaraja(baraja.tail,f) else
   filtraBaraja(baraja.tail, f)
}
```