Control 2 – Lenguajes de Programación (CC41A)

Departamento de Ciencias de la Computación – Universidad de Chile

Profesor: Éric Tanter

1 de Octubre 2007

1. **(1pt)** Implemente en Scheme la función `map` (sin usar el `map` de Scheme, obviamente ;))
2. **(1pt)** Explique por qué soportar funciones recursivas en los siguientes lenguajes *no* implica introducir ambientes recursivos:
   a) lenguaje con funciones de primer orden
   b) lenguaje con scope dinámico
3. **(2pt)** Jugando con régimen de evaluación...
   a) Implemente en Scheme la función `seq3`, que es similar al `begin`: permite evaluar en secuencia 3 expresiones y retorna el valor de la última.

   ```
   (seq3 (write "hello")
         (write "world")
         (+ 4 5))         --> "hello""world"9
   ```

   Hint: acuerdese que Scheme es un lenguaje con evaluación temprana.
   b) De la traducción exacta de su función `seq3` en Haskell. ¿Por qué ya no cumple su labor?
   c) Implemente en Scheme la función `if0` que toma tres parámetros, `e1`, `e2` y `e3`, tal que evalua `e2` si `e1` vale 0, `e3` sino. ¿Por qué no es posible, en Scheme, definir `if0` para que se use de la siguiente manera?:

   ```
   (if0 (- 4 5)
        (write "zero")
        (write "not zero")))  --> "not zero"
   ```

   Cambie el programa anterior para que funcione con su definición de `if0`.
   d) Defina `if0` en Haskell, y explique por qué el programa anterior ahora sí funciona tal cual.

4. **(1pt)** En la implementación de un lenguaje lazy, usted vio que para disminuir las veces que una expresión se forza a un valor, éste se guarda. Esta técnica tiene el nombre de caching.

   a) ¿Tiene sentido hacer esto en un lenguaje como Java o C?

   b) Dé un ejemplo que justifique su respuesta.

5. **(1pt)** Considere el lenguaje BCFAE, que soporta estructuras de datos mutables (boxes), y su función de evaluación para una expresión `if`:

```
(define (interp exp env)
  (type-case BCFAE exp
     ...
     ((if (test then else))
       (if (interp test env)
           (interp then env)
           (interp else env))) ...))
```

   ¿Por qué no es correcto interpretar una expresión `if` de esa manera? De un programa que muestra que el interprete esta errado.

Pauta

1. **(1pt)**

```
(define (-map f l)
   (cond
      ((null? l) '())
      (else (cons (f (car l)) (-map f (cdr l))))))
```

2. **(1pt)**

   a) **(0.5pt)** Como la lista de funciones para el interprete se define a priori (es decir no podemos definir nuevas funciones dentro del código de F1WAE) entonces una función puede llamarse a si misma (ie, recursivamente) sin problemas, ya que al hacer el lookup dentro de la lista de funciones encontrará la definición.

   b) **(0.5pt)** Como la aplicación de una función se interpreta en el ambiente actual, éste contiene una asociación de la función recursiva que se está aplicando. Luego no es necesario crear ambientes recursivos en este caso.

3. **(2.0pt)**

   a) **(0.5pt)**

```
(define (seq3 e f g) g)
```

   Esto tiene sentido, ya que por ser Scheme un lenguaje con evaluación *eager* los argumentos en una aplicación de función son evaluados antes de ser usados (incluso aunque no sean usados)

   b) **(0.5pt)**

```
seq3 e f g = g
```

o alternativamente:

```
seq3 _ _ g = g
```

La función así definida en Haskell no tiene sentido, ya que al no ser usados los argumentos en el cuerpo de ésta, no serán forzados a un valor (sólo quedan guardados como expresiones). Si hicieramos un llamado similar al del punto anterior, los valores *hello* y *world* no serían impresos por consola.

c) **(0.5pt)**

```
(define (if0 test truth falsity)
    (if (zero? test)
        truth
        falsity))
```

Como Scheme es un lenguaje con evaluación *eager*, los argumentos (truth, falsity) serán interpretados (y no pasados como expresiones) aunque no sean utilizados. Necesitamos crear un mecanismo que permita recibir expresiones, pero aplicarlas cuando nosotros queremos, ie. funciones de primera clase!.

```
(define (if0 test truth falsity)
   (if (zero? (test))
      (truth)
      (falsity)))
(if0 (lambda () (- 4 5))
   (lambda () (write "zero"))
   (lambda () (write "not zero")))
```

d) **(0.5pt)**

```
if0 test truth falsity =
| (test == 0) = truth
| otherwise = falsity

(if0 (4 - 5) (print "zero") (print "not zero"))
```

En este caso no es necesario *encapsular* las expresiones dentro de lambdas, ya que las ramificacione del if0 serán sólo evaluadas si son necesitadas (estamos en un lenguaje con evaluación *lazy*!!!).

4. **(1pt)**

a) **(0.5pt)** Cuando se implementa *caching* en un lenguaje lazy para disminuir el número de veces que una expresiones de forza a un valor, se está asumiento que la aplicación de una función retorna siempre el mismo valor cada vez que es invocada (ie. no tiene efectos de borde). Esto no tiene sentido en un lenguaje como Java o C, puesto que el valor que retorna una función dependerá del contexto donde éste se llame (Estado de un objeto por ej.)

b) **(0.5pt)**

```
class Person {
   ...
   public String _name;
   public String getName() { return _name; }
   public String setName(String name) { _name = name; }
}

{
   ...
   me.getName(); -> "fulano" (caching!)
   me.setName("sutano");
   me.getName(); -> "fulano" (se usó el valor guardado!)
}
```

5. (1.0pt) Como ahora podemos inducir cambios de estado en nuestro lenguaje, éstos son expresiones válidas y por lo tanto ser usado en cualquier parte de nuestro programa. En particular dentro del test, pueden cambiar cambios de estado; luego tal cual está el intérpete no se *traspasa* esa información a las ramificaciones del if.

```
{with {b {newbox 0}}
      {if0 {seqn {setbox b 5} {openbox b}}
           1
           {openbox b}}} -> 0! y queremos 5!.
```

6. (+1.0pt)

```
(define (interp expr env store)
   (type-case BCFAE expr
      ...
      [if0 (test truth falsity)
         (type-case ValuexStore (interp test env store)
            [vxs (test-value test-store)
               (if (num-zero? test-value)
                   (interp truth env test-store)
                   (interp falsity env test-store))])]))
```

Introducimos dos nuevas estructuras de datos en nuestro intérprete: *Store* y *ValuexStore*. De esta manera, el guardar el store resultante de interpretar la condición (*test*), nos permite notificar de los cambios de estado que hayan sucedido a las ramificaciones del if0.