

# Control 2 – Lenguajes de Programación

Departamento de Ciencias de la Computación

Universidad de Chile

Profesor: Éric Tanter

6 de Octubre 2008

2 horas / 2 puntos por pregunta
---------------------------------

## 1. Scope

Considere el siguiente programa:

```
{with {x 4}
  {with {f {fun {y} {+ x y}}}
    {with {x 5} {f 10}}}}
```

1. (0.5 pts) Cúal es el valor de este programa, con scope estático? y con scope dinámico?

**Estático** , 14.

**Dinámico** , 15.

2. (0.7 pts) Benancio, un alumno del curso le sugiere a ud. que podemos seguir usando scope dinámico, tomando la consideración de que cuando evaluamos una variable, tenemos que ir a buscar el valor más antiguo de esta variable en el ambiente, en vez de buscar el más reciente. Para este ejemplo esta claro que Benancio tiene la razón!!

Es cierto que Benancio tiene la razón en general?, Si es así justifique, sino de un contra-ejemplo donde la aseveración de Benancio falla, además explique de forma concisa cual es su error conceptual.

**Resp**, Cambiar el orden o el acceso al ambiente no soluciona el uso de scope dinámico, por ejemplo en este caso:

```
{with {a 5}
  {with {a {+ a a}}
    {with {f {fun {x} {+ x a}}}
      {f 1}}}}
```

Según la técnica de *Benancio* el resultado da 6, pero en realidad debe dar 11!!.

El error conceptual que tiene Benancio es que no logra comprender bien la diferencia entre scope estático y dinámico, donde el primero crea una clausura al momento de definir una función encerrando el ambiente actual, en cambio el segundo no encierra ningún ambiente al momento de crear una función.

3. (0.8 pts) Benancio propone la siguiente implementación de la función **interp**:

```
(define (interp expr ds)
  (type-case FAE expr
    ...
    [app (fun-expr arg-expr)
      (local ([define fun-val (interp fun-expr ds)])
```

```
(interp (closureV-body fun-val)
        (aSub (closureV-param fun-val)
              (interp arg-expr ds)
              ds))))))
```

Explique si la anterior implementación cumple con la definición de scope dinámico o estático o ninguno de los dos anteriores. Provea ejemplos para justificar su respuesta.

**Resp,** A pesar de que al momento de crear la clausura guarda el ambiente, este no es utilizado al momento de aplicar la función, usando el ambiente actual y no aquel que le corresponde por la clausura, por tanto no cumple con la definición de scope estático. En el caso del scope dinámico vemos que siempre se usa el ambiente actual que se va pasando de evaluación en evaluación, por tanto la implementación propuesta se comporta similar a un scope dinámico. Ej:

```
{with {b 1}
  {with {f {fun {x} {+ x b}}}}
  {with {b 2} {f 1}}}}
```

En este ejemplo el resultado de la expresión es 3 con la implementación dada.

## 2. Fibonacci en Haskell

1. **(0.5 pts)** Defina la función *zipOp*, vista en clase, que dado un operador y dos listas, retorna una lista donde los elementos de ambas listas han sido combinados usando el operador. No se olvide especificar el tipo de esta función.

```
zipOp :: (a -> b -> c) -> [a] -> [b] -> [c]
zipOp f [] _ = []
zipOp f _ [] = []
zipOp f (a:as) (b:bs) = (f a b) : zipOp f as bs

-- Soluci'on alternativa, mas compacta
zipOp :: (a -> b -> c) -> [a] -> [b] -> [c]
zipOp op as bs = map (\(a,b)-> op a b) (zip as bs)
```

2. **(0.5 pts)** Cual es el resultado de `zipOp (+) [1..] [1..]`?  
[2,4,6,...]
3. **(0.5 pts)** Defina una expresión *fibs* :: [Int] que genera la secuencia infinita de los números de Fibonacci:  
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...  
`fibs = 0 : 1 : zipOp (+) fibs (tail fibs)`
4. **(0.5 pts)** Usando *fibs*, defina una función *fib* :: Int → [Int] que retorna los *n* primeros números de Fibonacci (partiendo de 0).  
(Puede usar funciones estandar de Haskell.)  
`fib n = take n fibs`

## 3. Google's MapReduce

Para procesar grandes volúmenes de información, Google adoptó y hizo popular un modelo de programación llamado MapReduce. Ese nombre viene de las famosas funciones *map* y *reduce* (esa última se llama *fold* en Scheme). Este modelo funcional tiene la ventaja de ser fácilmente paralelizable, y esta al centro de la escalabilidad del sistema de procesamiento de Google. Vamos a ver una implementación de ese modelo usando Scheme.

Primero, explicamos a continuación como funciona MapReduce, a través de un ejemplo simple:

- La entrada del proceso **map-reduce** es una lista de pares *llave/valor*. Por ejemplo, consideremos un caso de documentos que contienen texto y para los cuales queremos sacar el numero de occurrencias de cada palabra. La lista:

```
(('doc1 ("a" "b" "c")) ('doc2 ("c" "a" "a")))
```

representa dos documentos de entrada, con su respectivo identificador como llave (**doc1** y **doc2**) y como valores, listas de palabras.

- El usuario tiene que proveer dos funciones *m* y *r*. *m* es una función que toma un par de entrada y produce un conjunto de pares llave/valor *intermedios*. Por ejemplo, en nuestro ejemplo una llave intermedia es una palabra, y un valor intermedio es simplemente un 1, para denotar su occurrencia. Es decir, para el **doc1** *m* retorna:

```
((("a" 1) ("b" 1) ("c" 1))
```

- El procedimiento *m* se aplica a todos los pares de entrada y se concatenan los pares intermedios. Así, obtenemos la siguiente lista de pares intermedios:

```
((("a" 1) ("b" 1) ("c" 1) ("c" 1) ("a" 1) ("a" 1))
```

- Luego, **map-reduce** agrupa todos los valores intermedios asociados con la misma llave intermedia. Por ejemplo, considerando los dos documentos de entradas, tenemos:

```
((("a" (1 1 1)) ("b" (1)) ("c" (1 1)))
```

- Finalmente, se aplica el procedimiento *r* a todos los pares intermedios consolidados. En el caso de contar occurrencias de palabras, *r* suma los numeros asociados, obteniendo:

```
((("a" 3) ("b" 1) ("c" 2))
```

(Para las preguntas, pueden usar Scheme o Haskell.)

1. (0.5 pts) De la definición de los procedimientos *m* y *r* que se usarón en el ejemplo considerado.

En Haskell:

```
m :: String -> String -> [(String, Int)]
m = (\_ xs -> map (\x -> (x,1)) xs)
r :: String -> [Int] -> Int
r = (\_ ys -> sum ys)
```

En Scheme:

```
;; m :: pair-of(Symbol, list-of(String)) -> list-of-pair(String,number)
(define m (lambda (kv) (map (lambda (x) (list x 1)) (second kv))))
;; r :: pair-of(String, list-of(number)) -> number
(define r (lambda (pair) (foldl + 0 (second pair))))
```

2. (0.5 pts) Describa el tipo de *m* y *r* en el caso general (es decir, no para el caso particular del ejemplo). Las dependencias entre los tipos de *m* y *r* deben aparecer, usando variable de tipos tal como visto en Haskell <sup>1</sup>.

En Haskell:

```
-- m :: (Eq v) => (k -> [v] -> [(v,i)])
-- r   :: (Eq v) => (v -> [i] -> u)
```

En Scheme:

```
;; m :: pair-of(k,list-of(v)) -> list-of-pair(v,i)
;; r :: pair-of(v,list-of(i)) -> u
```

donde:

k = llave

---

<sup>1</sup>En el caso de (Eq a) => a, indica que a pertenece a las clase Eq, por tanto puede ser cualquier tipo que sea comparable, dado que la clase Eq engloba a todos los tipos comparables, por ej. **Int**, **Char**, **String**, ...

```

v = valor
i = valor intermedio, producido por la funci'on m
u = valor de reduci'on de valores intermedios, producido por la funci'on r

```

3. (0.5 pts) Implemente la funci3n map-reduce y usela para resolver el ejemplo. (*Piense en usar funciones auxiliares para descomponer el trabajo en funciones entendibles, reusables, y gen3ricas.*)

En Haskell:

```

-- Cada Variable de tipo que usemos en las sgtes firmas
-- viene explicado en esta secci'on de comentarios.
-- k = llave
-- v = valor
-- i = valor intermedio, producido por la funci'on m
-- u = valor de reduci'on de valores intermedios, producido por la funci'on r

mapReduce :: (Eq v) => (v -> [i] -> u)
              -> (k -> [v] -> [(v,i)])
              -> [(k, [v])] -> [(v, u)]

-- La funci'on mapReduce, compuesta por tres etapas: "mapear", agrupar y reducir
mapReduce r m = (reduce_ r) . g . (map_ m)

map_ :: (k -> [v] -> [(v,i)]) -> [(k, [v])] -> [(v,i)]
-- Recibe la funci'on m (del usuario) para aplicar el map value cada (key, values)
map_ m = foldl (\ls (key,values)-> ls ++ (m key values)) []

g :: (Eq v) => [(v, i)] -> [(v, [i])]
-- Agrupa todas las llaves intermedias, funci'on usada por mapReduce
-- inter,interr, representan cada uno valores intermedios
g [] = []
g ((value,inter):rest) = (:) (value,(groupby value rest))
                        (g (filter \(other,interr) -> (value /= other)) rest))
  where groupby value [] = [inter]
        groupby value ((other,interr):rest) = if (value == other)
        then (interr:groupby value rest) else (groupby value rest)

reduce_ :: (v -> [i] -> u) -> [(v, [i])] -> [(v, u)]
-- Recibe la funci'on r (del usuario) para aplicar el reduce a cada grupo
reduce_ r = map \(value,inters) -> (value,(r value inters)))

```

En Scheme:

```

;; Cada Variable de tipo que usemos en los sgtes contratos
;; viene explicado en esta secci'on de comentarios.
;; k = llave
;; v = valor
;; i = valor intermedio, producido por la funci'on m
;; u = valor de reduci'on de valores intermedios, producido por la funci'on r

;; map-reduce :: (pair-of(v,list-of(i)) -> u)
;;              x (pair-of(k,list-of(v)) -> list-of-pair(v,i))
;;              x list-of-pair(k,list-of(v)) -> list-of-pair(v,u)
;; La funci'on map-reduce, compuesta por tres etapas: "mapear", agrupar y reducir
(define map-reduce (lambda (r m pairs)
  (reduce- r (group (map- m pairs))))))

```

```

;;map- :: (pair-of(k,list-of(v)) -> list-of-pair(v,i))
;;      x list-of-pair(k,list-of(v)) -> list-of-pair(v,i)
;; Recibe la funci'on m (del usuario) para aplicar el map a cada (key, values)
;; kv, representa el par (llave, valores)
(define map-
  (lambda (m pairs)
    (foldl (lambda (kv ls) (append ls (m kv))) empty pairs)))

;; group :: list-of-pair(v,i) -> list-of-pair(v,list-of(i))
;; Agrupa todas las llaves intermedias, funci'on usada por mapReduce
;; inter, representa un valor intermedio
(define group
  (lambda (pairs)
    (if (empty? pairs) empty
        (let ((value (first (first pairs)))
              (inter (second (first pairs))))
          (cons (list value (groupby value inter (rest pairs)))
                (group (filter (lambda (pair) (not (string=? value (first pair))))
                           (rest pairs))))))))))

;; reduce- :: (pair-of(v,list-of(i)) -> u)
;;          x list-of-pair(v, list-of(i)) -> list-of-pair(v,u)
;; Recibe la funci'on r (del usuario) para aplicar el reduce a cada grupo
(define reduce-
  (lambda (r pairs)
    (map (lambda (pair) (list (first pair) (r pair))) pairs)))

;; Funci'on Anexa
;; groupby :: v x i x list-of-pair(v,i) -> list-of(i)
;; Funci'on auxiliar de apoyo a group
(define groupby
  (lambda (value inter ls)
    (if (empty? ls) (list inter)
        (let ((other (first (first ls)))
              (interr (second (first ls))))
          (if (string=? value other) (cons interr (groupby value inter (rest ls)))
              (groupby value inter (rest ls)))))))

```

4. (0.5 pts) Use map-reduce para obtener, dado una lista de documentos y sus contenidos (como en el ejemplo), un índice invertido que señala para cada palabra, la lista de documentos en que aparece.

En Haskell:

```

-- inverted_index :: [(k, [v])] -> [(v, [k])]
-- Funci'on que retorna el 'indice invertido de la lista (key, values)
-- esta parametrizada por funciones r y m aplicadas al mapReduce
inverted_index = mapReduce (\_ docs -> foldl (\ls doc -> (if (elem doc ls) then ls else (doc:ls)))
                                          [] docs)
                        (\doc keys -> map (\key -> (key,doc)) keys)

```

En Scheme:

```

;; inverted-index :: list-of-pair(k,list-of(v))
;;                -> list-of-pair(v,list-of(k))
;; Funci'on que retorna el 'indice invertido de la lista (llave, lista de
;; valores) esta parametrizada por r y m aplicadas al map-reduce.
;; dk, representa el par (doc, llaves)

```

```

(define (inverted-index pairs)
  (map-reduce (lambda (xds)
    (foldl (lambda (doc ls) (if (elem doc ls) ls (cons doc ls)))
      empty (second xds)))
    (lambda (dk)
      (map (lambda (key) (list key (first dk))) (second dk)))
    pairs))

```

Para mayor detalle:

**Scheme** , ver archivo mapReduce.scm

**Haskell** , ver archivo mapReduce.hs