Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2008

# Examen III

(40 puntos)

Nombre:

Carnet:

1. **(20 puntos)** Considere cuidadosamente las siguientes cuestiones y seleccione exactamente una respuesta de las alternativas que se presentan. Cada cuestión contestada correctamente vale **cuatro (4) puntos** pero una cuestión contestada incorrectamente **resta dos (2) puntos**.

   (a) Considere un lenguaje con alcance estático en el que se permite anidamiento de subrutinas. Al momento de llamar la subrutina, ¿quién debe encargarse, entre la subrutina llamadora y la subrutina llamada, de la creación del enlace de la cadena estática, y por qué?

   i. **La subrutina llamadora, pues es ella quien puede determinar su nivel de anidamiento estático con relación a la subrutina llamada.**
   ii. La subrutina llamadora, pues dado que cualquiera de las dos puede determinar el nivel de anidamiento estático, en la subrutina llamadora el código estaría dentro del prólogo.
   iii. La subrutina llamada, pues es ella quien puede determinar su nivel de anidamiento estático con relación a la subrutina llamadora.
   iv. La subrutina llamada, pues dado que cualquiera de las dos puede determinar el nivel de anidamiento estático, en la subrutina llamada el código estaría dentro del prólogo.

   (b) Considere las siguientes definiciones Haskell

```
forrest run = run (forrest run)
grok = forrest (\h i f a d ->
                  if (d == 1) then [(i,f)]
                              else (h i a f (d-1)) ++ [(i,f)] ++ (h a f i (d-1)))
```

   Ahora considere las siguientes afirmaciones:

   i. El tipo de `forrest` es `a -> (a -> a)`
   ii. El tipo de `forrest` es `(a -> a) -> a`
   iii. `forrest` no es una función de orden superior.
   iv. `grok "foo" 42` es una función currificada de tipo `t -> Integer -> [(String,Integer)]`
   v. `grok "foo" "bar" "baz"` es una función currificada, de tipo `Integer -> [(String,String)]`

   ¿Cuáles afirmaciones son ciertas?

   A. Sólo i y v.
   B. Sólo ii y iv.
   C. Sólo i, iii y v.
   D. **Sólo ii y v.**
   E. Sólo ii, iii y v.

(c) Se tiene el siguiente programa escrito en un lenguaje que arma los parámetros actuales en el registro de activación según el orden de derecha a izquierda de los parámetros formales:

```
procedure foo (bar, baz, qux : int)
begin
  baz := baz + 1
  qux := qux + bar
end
begin
  var grok, bleh : int;
  grok := 2;
  bleh := 3;
  foo(grok + bleh, grok, grok);
  write(grok * bleh)
end
```

Considere *dos* corridas del mismo programa:

- En la primera, el procedimiento `foo` es llamado pasando `bar` por valor y el resto de los parámetros por valor/resultado. Los resultados se regresan en el mismo orden en que se desarma el registro de activación.
- En la segunda, el procedimiento `foo` es llamado pasando *todos* los parámetros por referencia.

¿Cuál sería la salida del programa en estos dos casos, respectivamente?

i. 24 y 21
ii. 3 y 24
iii. **21 y 24**
iv. 24 y 3

(d) El lenguaje funcional Scheme provee las primitivas `force` y `delay` porque:

i. El orden de evaluación de Scheme es prefijo.
ii. El orden de evaluación de Scheme es normal.
iii. **El orden de evaluación de Scheme es aplicativo.**
*Nota:* Cuando es necesario simular orden de evaluación normal (perezoso) se utiliza la primitiva `delay` sobre cualquier *s-expresión* para convertirla en una "promesa", la cual debe ser evaluada explícitamente después utilizando `force`.
iv. El lenguaje no tiene un orden de evaluación definido, de modo que utilizando `force` se establece el normal y utilizando `delay` se establece el aplicativo.

(e) Considere las siguientes definiciones en un lenguaje orientado a objetos con modelo de valor que dispone de operadores para tomar la dirección de un objeto (`&`) para almacenarla en un apuntador (`*`).

```
class foo { int grok() }
class bar : extends foo { int grok() }
bar b;
foo *pf = &b;
```

i. Si `pf->grok()` invoca a `foo::grok()` entonces el lenguaje tiene ligadura dinámica y se dice que el método `foo::grok()` es virtual.
ii. Si `pf->grok()` invoca a `bar::grok()` entonces el lenguaje tiene ligadura dinámica y se dice que el método `foo::grok()` ha sido redefinido (*redefined*).
iii. **Si `pf->grok()` invoca a `bar::grok()` entonces el lenguaje tiene ligadura dinámica y se dice que el método `foo::grok()` ha sido sustituido (*overrided*).**
*Nota:* La diferencia entre redefinir y sustituir estriba en que el primero ocurre solamente cuando el lenguaje tiene ligadura estática, mientras que el segundo ocurre cuando el lenguaje tiene ligadura dinámica.
iv. Si `pf->grok()` invoca a `foo::grok()` y luego a `bar::grok()` entonces el lenguaje tiene ligadura dinámica y se dice que el método `bar::grok()` ha sido redefinido (*redefined*).

2. **(6 puntos)** Demuestre que los combinadores en Lambda Cálculo definidos a continuación son Operadores de Punto Fijo.

   Para demostrar que un combinador es un Operador de Punto Fijo, basta demostrar que

$$\mathbf{Y}g \quad = \quad g(\mathbf{Y}g)$$

   para cualquier $g$ arbitrario.

   (a) $\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

$$\begin{aligned}
&\mathbf{Y}g \\
&= \quad \{\text{Expansión de } \mathbf{Y}\} \\
&(\underline{\lambda f.}(\lambda x.\underline{f}(xx))(\lambda x.\underline{f}(xx))\underline{g} \\
&= \quad \{\beta \text{ reducción}\} \\
&(\underline{\lambda x}.g(\underline{xx}))\underline{(\lambda x.g(xx))} \\
&= \quad \{\beta \text{ reducción}\} \\
&g((\lambda x.\underline{g}(xx))(\lambda x.\underline{g}(xx)) \\
&= \quad \{\beta \text{ abstracción}\} \\
&g(\underline{(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))}g) \\
&= \quad \{\text{Definición de } \mathbf{Y}\} \\
&g(\mathbf{Y}g)
\end{aligned}$$

   (a) $\hat{\mathbf{Y}} = \lambda f.(\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy))$

$$\begin{aligned}
&\hat{\mathbf{Y}}g \\
&= \quad \{\text{Expansion de } \hat{\mathbf{Y}}\} \\
&(\underline{\lambda f.}(\lambda x.\underline{f}(\lambda y.xxy))(\lambda x.\underline{f}(\lambda y.xxy)))\underline{g} \\
&= \quad \{\beta \text{ reducción}\} \\
&(\underline{\lambda x.}g(\lambda y.\underline{xx}y))\underline{(\lambda x.g(\lambda y.xxy))} \\
&= \quad \{\beta \text{ reducción}\} \\
&g(\underline{\lambda y.((\lambda x.g(\lambda y.xxy))(\lambda x.g(\lambda y.xxy))y)} \\
&= \quad \{\eta \text{ equivalencia}\} \\
&g(((\lambda x.\underline{g}(\lambda y.xxy))(\lambda x.\underline{g}(\lambda y.xxy)))) \\
&= \quad \{\beta \text{ abstracción}\} \\
&g(\underline{\lambda f.((\lambda x.f(\lambda y.xxy))(\lambda x.f(\lambda y.xxy)))}g) \\
&= \quad \{\text{Definición de } \hat{\mathbf{Y}}\} \\
&g(\hat{\mathbf{Y}}g)
\end{aligned}$$

3. Programación Funcional de Orden Superior

(a) **(4 puntos)** Considere las siguientes definiciones de tipos de datos Haskell:

```
type Vector = [Integer]
type Matriz = [Vector]
```

Escriba la función

```
transpose :: Matriz -> Matriz
```

que recibe una matriz representada en *row-major order* como una lista de listas de al menos un elemento cada una, y produce como resultado la matriz transpuesta utilizando la misma representación; la función debe emitir un `error` si la lista recibida no corresponde a una matriz válida. Por ejemplo,

```
> transpose [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
> transpose [[1,2],[3],[4,5]]
*** Exception: Matriz invalida
> transpose [[]]
*** Exception: Matriz invalida
```

*Pista*: transponer es muy fácil si se puede garantizar que la matriz está bien formada.

La matriz estará bien formada si no es vacía, la longitud de todos sus elementos es la misma, y además dicha longitud es mayor que 0; eso lo verifica la función auxiliar `isValid`. Si la matriz está bien formada, para transponer simplemente es necesario tomar el primer elemento de cada lista para armar una nueva lista y luego transponer la matriz constituida por los restos de las listas; eso lo hace la función auxiliar `doIt`.

```
transpose   :: Matriz -> Matriz
transpose m =
  if (isValid m) then doIt m
                 else error "Matriz invalida"
    where doIt ([]:_) = []
          doIt rows   = map head rows : (doIt $ map tail rows)
          isValid []  = False
          isValid m   = allEqual (head l) (tail l)
             where l             = map length m
                   allEqual e es = (e > 0) && (and $ map (e==) es)
```

(b) **(4 puntos)** Considere las funciones de orden superior sobre listas disponibles en Haskell con el funcionamiento discutido en clase

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Implemente ambas funciones en Scheme.

```
(define foldr
  (lambda (f b l)
    (if (null? (cdr l))
        (f (car l) b)
        (f (car l) (foldr f b (cdr l))))))
(define foldl
  (lambda (f b l)
    (if (null? (cdr l))
        (f b (car l))
        (foldl f (f b (car l)) (cdr l)))))
```

4. **(6 puntos)** Muchos interpretadores de Prolog proveen el predicado `setof/3` definido como:

```
setof(Cosas,Objetivo,Conjunto)
```

tal que obtiene **todos** los resultados posibles de `Objetivo`, conservando las `Cosas` indicadas, para instanciar en `Conjunto` la lista de `Cosas` *sin repeticiones*. Es particularmente útil si uno quiere obtener todas las soluciones para el `Objetivo` en forma de lista, por ejemplo, si se tiene la base de datos

```
p(a,b).
p(b,c).
p(a,c).
p(b,c).
p(c,a).
```

entonces puede utilizarse

```
?- setof([X,Y],p(X,Y),Todas).
Todas = [[a,b],[a,c],[b,c],[c,a]]
```

Note que la solución `[b,c]` está repetida, pero `setof/3` **elimina** las repeticiones. Suponga que Ud. tiene un interpretador de Prolog que **no** dispone de este predicado. Provea una implantación en Prolog que simule el predicado `setof/3`. Asuma que dispone del predicado `member/2` discutido en clase y en el libro de texto.

*Pista*: genere resultados, "recuérdelos" y luego "olvídelos" a medida que los recopila en la lista.

Para generar los resultados y "recordarlos" se usa un predicado generador típico de Prolog, que obtiene un resultado, lo agrega a la base de datos como un predicado temporal y fuerza una falla para que el backtracking genere más resultados. Cuando no quedan resultados, se hace triunfar incondicionalmente el predicado generador.

Luego, se utilizar un predicado para construir la lista de resultados. Se consulta la base de datos para encontrar todos los resultados temporales y eliminarlos. Si el resultado ya está en la lista, se ignora, en caso contrario se agrega a la lista. Se utiliza un cut para evitar considerar cada resultado más de una vez. Una vez eliminados todos los resultados, se cuenta con la lista final.

```
setOf(Cosas,Objetivo,Conjunto) :- genera(Cosas,Objetivo),
                                  recopila([],Conjunto).
genera(Cosa,Objetivo) :- call(Objetivo),
                         asserta(mesirve(Cosa)),
                         fail.
genera(_,_).
recopila(Vienen,Van) :- mesirve(Cosa),
                        retract(mesirve(Cosa)),
                        (
                           member(Cosa,Vienen),
                           recopila(Vienen,Van)
                           ;
                           recopila([Cosa|Vienen],Van)
                        ), !.
recopila(Todas,Todas).
```

5. **(5 puntos extra)** Considere la siguiente declaración de una clase abstracta en C++

```
class Grok {
  virtual void bleh () = 0;
}
```

Ahora considere los siguientes fragmentos de un programa C++

```
class Zen : Grok {
  virtual void bleh () { /* ... */ }
}
class Master : Grok {
  virtual void hack () { /* ... */ }
}
Grok  funcion1()        { /* ... */ }  // (1)
void  funcion2(Grok  p) { /* ... */ }  // (2)
Grok& funcion3(Grok &p) { /* ... */ }  // (3)
int main () {
  Grok    foo;                         // (4)
  Grok   *bar;                         // (5)
  Zen     baz;                         // (6)
  Master  qux;                         // (7)
  static_cast<Grok>(baz);              // (8)
}
```

Indique y justifique cuáles de las instrucciones numeradas son válidas o inválidas.

**Nota:** esta pregunta solamente será tomada en cuenta si Ud. aprueba el examen con los puntos obtenidos en el resto de las preguntas.

C++ utiliza modelo de *valor* para las variables, por tanto cada variable es un objeto concreto elaborable, siendo posible tener apuntadores a objetos concretos y tomar referencias de objetos concretos.

**Línea 1** Es **inválida**. Como `Grok` es una clase abstracta es incorrecto definir una función que retorne una variable de tipo `Grok`, pues no es posible elaborar un objeto de una clase abstracta.

**Línea 2** Es **inválida**. Como `Grok` es una clase abstracta es incorrecto definir una función que reciba como parámetro formal un objeto de tipo `Grok`, pues no es posible elaborar un objeto de una clase abstracta.

**Línea 3** Es **válida**. Tanto el parámetro como el valor de retorno de la función son *referencias* a un objeto de la clase `Grok`. Si bien `Grok` es una clase abstracta, las referencias a la clase `Grok` son compatibles con objetos de *subclases* de `Grok` que potencialmente serían clases concretas.

**Línea 4** Es **inválida**. Como `Grok` es una clase abstracta es incorrecto definir una variable de tipo `Grok` pues es imposible elaborar un objeto de una clase abstracta.

**Línea 5** Es **válida**. La variable definida es un *apuntador* a objetos de la clase `Grok`. Si bien `Grok` es una clase abstracta, el apuntador podría hacer referencia a un objeto de *subclases* de `Grok` que potencialmente serían objetos concretos.

**Línea 6** Es **válida**. La clase `Zen` implementa el método `bleh()` de modo que es una clase concreta y es posible elaborar objetos concretos.

**Línea 7** Es **inválida**. La clase `Master` sigue siendo abstracta porque no provee implementación para el método `bleh()` de modo que es imposible elaborar un objeto de una clase abstracta.

**Línea 8** Es **inválida**. Si bien `baz` es un objeto concreto, al forzar su tipo *éstatico* a `Grok`, estaría perdiendo sus atributos concretos para pertenecer a la clase abstracta, lo cual contradiría su naturaleza concreta.