Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2007

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 orientado a objetos en el que la herencia corresponde a la noción de subtipo y las variables son manejadas con modelo de valor. Suponga que en ese lenguaje se tiene una jerarquía de tres clases `Baz`, `Bar` y `Foo`, en la que `Baz` es subclase de `Bar` y `Bar` es subclase de `Foo`. Si en el lenguaje se declara un procedimiento `Qux` de la siguiente forma:

   ```
   procedure Qux ( in bar0 : Bar, out bar1 : Bar )
   ```

   para el que, de acuerdo con el modelo de valor del lenguaje, el parámetro de entrada `bar0` es pasado por valor y el parámetro de salida `bar1` es pasado por resultado. ¿En cuál de los siguientes casos el compilador permite que se haga una llamada de la forma `Qux(q1,q2)`?

   i. **Si `q1` es de tipo `Baz` y `q2` es de tipo `Foo`.**
   ii. Si `q1` es de tipo `Foo` y `q2` es de tipo `Baz`.
   iii. Si `q1` y `q2` son ambos de tipo `Baz`.
   iv. Si `q1` y `q2` son ambos de tipo `Foo`.

   (b) Sea la siguiente definición de función escrita en Haskell:

   ```
   foo bar baz qux = 42 + qux ( 42 * baz, bar + 42 )
   ```

   Suponga que en el lenguaje tiene solamente el tipo numerico `Int`.
   Ahora considere las siguientes afirmaciones:

   i. El tipo de `foo` es `(Int,Int,((Int,Int)->Int)) -> Int`
   ii. El tipo de `foo` es `Int -> Int -> ((Int,Int) -> Int) -> Int`
   iii. El tipo de foo es `Int -> (Int -> ((Int -> Int) -> Int) -> Int)`
   iv. La definición de función `grok = foo bar baz` sería válida.
   v. `foo` no es una función currificada, pero su último argumento `qux` si lo es.
   vi. `foo` es una función currificada, pero su último argumento `qux` no lo es.
   ¿Cuáles afirmaciones son ciertas?

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

(c) Se tiene el siguiente programa escrito en pseudo-código:

```
procedure bar (baz, qux : int)
begin
  baz := baz + qux;
  qux := baz + qux
end
begin
  var foo : int;
  foo := 21;
  bar(foo,foo);
  write(foo)
end
```

Considere *dos* corridas del mismo programa:

- En la primera, el procedimiento `bar` es llamado pasando `baz` por valor y `qux` por valor/resultado.
- En la segunda, el procedimiento `bar` es llamado pasando *ambos* parámetros por referencia.

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

i. 21 y 84
ii. 42 y 84
iii. **63 y 84**
iv. 63 y 42

(d) El primer mecanismo para implantar excepciones considerado en el libro de texto es el de mantener una pila de manejadores de excepciones durante la ejecución. Con esta alternativa, ¿cuáles de las siguientes características de la implantación son postuladas por el autor?

i. Todo `throw` debe empilar un manejador.
ii. Todo `throw` debe desempilar un manejador.
iii. Solamente los `throw` que ocurren en bloques protegidos por `try` deben desempilar un manejador.
iv. Se debe empilar un manejador cada vez que se entra a un bloque protegido `try`.
v. Se debe empilar un manejador cada vez que se entra a una subrutina no protegida.

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

(e) Se tiene una clase abstracta `Foo` en C++. El lenguaje permite declarar variables del tipo `Foo*` pero no permite declarar variables del tipo `Foo`. ¿A qué se debe esto?

i. A que C++ maneja el modelo de referencias, que cuando se trata de clases favorece el uso de tipos referencia `Foo*` sobre el tipo de valor básico `Foo`.
ii. **A que C++ maneja el modelo de valor y no podría manejar un objeto incompleto.**
iii. A que C++ utiliza asociación de nombres estática y para una clase abstracta es obligatorio utilizar asociación dinámica.
iv. A que es más eficiente utilizar una referencia para objetos en lugar de valores simples y el compilador C++ optimiza ese caso.

2. Considere las siguientes definiciones de combinadores en Lambda Cálculo

$$\begin{aligned}
\mathbf{K} &= \lambda x.\lambda y.x \\
\mathbf{S} &= \lambda f.\lambda g.\lambda x.(fx)(gx) \\
\mathbf{I} &= \lambda x.x \\
\hat{\mathbf{Y}} &= (\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))
\end{aligned}$$

(a) **(3 puntos)** Demuestre que $\mathbf{SKK} = \mathbf{I}$.

$$\begin{aligned}
&\mathbf{SKK} \\
&= \{\text{Expandiendo } \mathbf{S}\} \\
&(\underline{\lambda f}.\lambda g.\lambda x.(\underline{f}x)(gx))\underline{\mathbf{K}}\mathbf{K} \\
&= \{\beta\} \\
&(\underline{\lambda g}.\lambda x.(\mathbf{K}x)(\underline{g}x))\underline{\mathbf{K}} \\
&= \{\beta\} \\
&\lambda x.(\mathbf{K}x)(\mathbf{K}x) \\
&= \{\alpha\,[z := x]\} \\
&\lambda z.(\mathbf{K}z)(\mathbf{K}z) \\
&= \{\text{Expandiendo } \mathbf{K}\} \\
&\lambda z.((\underline{\lambda x}.\lambda y.\underline{x})\underline{z})((\lambda x.\lambda y.x)z) \\
&= \{\beta\} \\
&\lambda z.(\underline{\lambda y}.z)\underline{((\lambda x.\lambda y.x)z)} \\
&= \{\beta\} \\
&\lambda z.z \\
&= \{\text{Definición de } \mathbf{I}\} \\
&\mathbf{I}
\end{aligned}$$

(b) **(3 puntos)** Demuestre que $\hat{\mathbf{Y}}$ es un Operador de Punto Fijo.
Si $\hat{\mathbf{Y}}$ es un Operador de Punto Fijo entonces para cualquier función f debe satisfacer $\hat{\mathbf{Y}}f = f\hat{\mathbf{Y}}f$, por lo tanto verificamos

$$\begin{aligned}
&\hat{\mathbf{Y}}f \\
&= \{\text{Expandiendo } \hat{\mathbf{Y}}\} \\
&(\underline{\lambda x}.\lambda y.y(\underline{xx}y))\underline{(\lambda x.\lambda y.y(xxy))}f \\
&= \{\beta\} \\
&(\underline{\lambda y}.\underline{y}((\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))\underline{y}))f \\
&= \{\beta\} \\
&f(\underline{(\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))}f) \\
&= \{\text{Definición de } \hat{\mathbf{Y}}\} \\
&f\hat{\mathbf{Y}}f
\end{aligned}$$

3. El problema de las Torres de Hanoi consiste de una serie de discos de diámetros decrecientes con un agujero central, y tres varillas verticales en los cuales pueden ser insertados. Una de las varillas se considera "Inicial", otra de las varillas se considera "Final" y la restante se considera "Intermedia". El problema comienza colocando todos los discos en la varilla Inicial, ordenados de mayor a menor según su diámetro, con el mayor al fondo y el menor al tope. El problema se resuelve si todos los discos son llevados hasta la varilla Final respetando las siguientes reglas:

   - Solamente puede moverse **un** disco a la vez.
   - Un disco sólo puede colocarse encima de un disco que tenga diámetro **mayor**.

   Queremos resolver el problema definiendo una función `hanoi(n,i,f,m)` que produce una **lista** con los movimientos necesarios para trasladar `n` discos desde la varilla `i` hasta la `f`, usando la varilla `m` como auxiliar. Para simplificar la solución, usaremos números enteros para identificar las varillas, i.e. 1 para `i`, 2 para `f`, 3 para `m`.

   (a) **(4 puntos)** Complete la siguiente definición Haskell **sin** utilizar funciones adicionales para tener una solución al Problema de las Torres de Hanoi.

   ```
   hanoi :: ?
   hanoi 0 i f m = ...
   hanoi n i f m = ...
   ```

   Al utilizar la función debe obtenerse algo como

   ```
   > hanoi 3 1 2 3
   [(1,3), (1,2), (3,2), (1,3), (2,1), (2,3), (1,3)]
   ```

   Note que sólo debe completar la definición, no puede utilizar funciones auxiliares.

   ```
   hanoi 0 i f m = []
   hanoi n i f m = (hanoi (n-1) i m f) ++ (i,f) : (hanoi (n-1) m f i)
   ```

   (b) **(3 puntos)** Suponga que dispone de las funciones λ-cálculo vistas en clase (números, booleanos, condicionales) y adicionalmente las funciones **nil**, **cons**, **append** y **pred**, para construir listas vacías, listas, concatenar listas y calcular el predecesor de un número, respectivamente. Deduzca la función λ-cálculo recursiva Hanoi utilizando el Operador de Punto Fijo $\mathbf{Y}$ utilizando el método descrito en clase.

   Si definieramos la función **hanoi** directamente a λ-cálculo, tendríamos una función de cuatro argumentos: la cantidad de discos (n), la varilla de inicio (i), la varilla final (f) y la varilla intermedia (m), de la forma

   $$\mathbf{hanoi} = \lambda n.\lambda i.\lambda f.\lambda m.(\mathbf{ife}(\mathbf{zero}?\, n)\mathbf{nil}(\mathbf{append}(\mathbf{hanoi}(\mathbf{pred}\, n)imf)(\mathbf{cons}(\mathbf{cons}\, if)(\mathbf{hanoi}(\mathbf{pred}\, n)mfi))))$$

   Sin embargo, la recursión directa no está permitida en el λ-cálculo, por lo tanto debemos abstraer la llamada recursiva a la función **hanoi** utilizando una nueva función **g** con un argumento adicional:

   $$\mathbf{g} = (\lambda h.\lambda n.\lambda i.\lambda f.\lambda m.(\mathbf{ife}(\mathbf{zero}?\, n)\mathbf{nil}(\mathbf{append}(h(\mathbf{pred}\, n)imf)(\mathbf{cons}(\mathbf{cons}\, if)(h(\mathbf{pred}\, n)mfi))))$$

   y ahora resulta que

   $$\mathbf{hanoi} \quad = \quad \mathbf{g}\,\mathbf{hanoi}$$

   con lo que la función **hanoi** no es más que un punto fijo de la función **g**, por lo tanto basta aplicar el operador de punto fijo $\mathbf{Y}$ para poder definir

   $$\begin{aligned} \mathbf{hanoi} &= \mathbf{Yg} \\ &= \mathbf{Y}(\lambda h.\lambda n.\lambda i.\lambda f.\lambda m.(\mathbf{ife}(\mathbf{zero}?\, n)\mathbf{nil}(\mathbf{append}(h(\mathbf{pred}\, n)imf)(\mathbf{cons}(\mathbf{cons}\, if)(h(\mathbf{pred}\, n)mfi)))) \end{aligned}$$

4. Use las técnicas de metaprogramación disponibles en Prolog para implantar predicados equivalentes a las siguientes funciones de orden superior:

   (a) **(2 punto)** `map(Functor,Lista,ListaAplicada)` que tenga éxito si `ListaAplicada` es el resultado de aplicar el `Functor` a cada elemento de `Lista`, esto es, el predicado `map/3` debe poder utilizarse como

   - `map(f,[1,2,3],L)` triunfe unificando `L = [f(1),f(2),f(3)]`.
   - `map(F,[1,2,3],[f(1),f(2),f(3)])` triunfe unificando `F = f`.
   - `map(F,[1,2,3],[f(1),g(2),f(3)])` no triunfe.
   - `map(F,L,[f(1),f(2),f(3)])` triunfe unificando `F =f`, `L = [1,2,3]`.

   ```
   map(_,[],[]).
   map(F,[X|T],[A|R]) :- A =.. [F,X], map(F,T,R).
   ```

   (b) **(3 puntos)** `filter(Predicado,Lista,Filtrados)` que tenga éxito si `Filtrados` es una lista que contiene los elementos de `Lista` para los cuales el `Predicado` es cierto, esto es, el predicado `filter/3` debe poder utilizarse como

   - `filter(atom,[a,X,b,1,f(Y)],L)` triunfa con `L = [a,b]`.
   - `filter(atom,[X,Y,g(a,b)],L)` triunfa con `L = []`.
   - Cualquier aplicación sin el `Predicado` debe fallar. *Pista*: use un `cut` apropiadamente en lugar de chequear si el `Predicado` falta.

   ```
   filter(_,[],[]).
   filter(P,[X|T],[X|R]) :- C =.. [P,X], call(C), !, filter(P,T,R).
   filter(P,[_|T],L) :- filter(P,T,L).
   ```

   (c) **(2 puntos)** `foldr(Functor,Base,Lista,Resultado)` que tenga éxito si `Resultado` es un término construido aplicando el `Functor` (que se asume binario) a los elementos de la lista, dos a dos, asociando a la derecha, ie. el `Resultado` debe unificarse con `Functor(`l_1`,Functor(`l_2`,...Functor(`l_n`,Base)...))` donde l_i son los elementos de la lista de izquierda a derecha. El predicado `foldr/4` debe poder utilizarse como

   - `foldr(+,0,[1,2,3],L)` triunfa con `L = +(1,+(2,+(3,0)))`.
   - `foldr(f,B,[a,b,c],f(a,f(b,f(c,z))))` triunfa con `B = z`.
   - `foldr(f,z,L,f(a,f(b,f(c,z))))` triunfa con `L = [a,b,c]`.
   - Cualquier aplicación sin el `Functor` debe fallar.

   ```
   foldr(F,B,[X],Y) :- Y =.. [F,X,B].
   foldr(F,B,[X|T],Y) :- foldr(F,B,T,Z), Y =.. [F,X,Z].
   ```