Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Abril-Julio 2015

Carnet:_____

Nombre:_____

Examen III

(40 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	Pregunta 4	**Total**
18 puntos	6 puntos	8 puntos	8 puntos	**40 puntos**

Pregunta 1 - 18 puntos

Esta pregunta consta de seis (6) subpreguntas de selección, numeradas de 1.1 a 1.6. Cada una de las subpreguntas viene acompañada de cuatro posibles respuestas (a, b, c, d), entre las cuales sólo **una** es correcta. Ud. deberá marcar completamente y sin posibilidad de confusión la opción que considere correcta. Cada una de las subpreguntas tiene un valor de tres (3) puntos. Tres subpreguntas incorrectas eliminan una correcta.

1.1. La cadena estática es manejada por el llamador, con código generado a tiempo de compilación ubicado en la secuencia de llamada. La cadena dinámica es manejada por el llamado, con código generado a tiempo de compilación ubicado en el prólogo, ¿Es posible que para un registro de activación particular la cadena dinámica sea idéntica a la cadena estática?

a) No, pues la cadena estática depende del anidamiento, y la dinámica del orden de invocación.

b) Si, y sólo si el lenguaje no tiene anidamiento de subrutinas.

c) No, pues se almacenan en partes diferentes del registro de activación.

d) Siempre es posible.

1.2. Considere las características mínimas de los lenguajes funcionales Scheme y Haskell comparados en clase. ¿Cuál de las siguientes afirmaciones es **falsa**?

a) Solamente uno tiene verificación fuerte de tipos.

b) Solamente uno tiene metaprogramación.

c) Solamente uno tiene I/O separado explícitamente.

d) Solamente uno permite definir operadores arbitarios.

1.3. En clase discutimos dos mecanismos para manejo de excepciones. ¿Cuáles de las siguientes afirmaciones son ciertas?

I. Ambos determinan dinámicamente el manejador adecuado para cada instrucción.

II. Ambos determinan estáticamente el manejador adecuado para cada instrucción.

III. Los dos son igual de eficientes al despachar una excepción, pero uno es más lento que el otro cuando no hay excepciones.

IV. Ambos requieren manejadores implícitos adicionales.

a) Sólo I, III y IV.

b) Sólo II, III y IV.

c) Sólo III.

d) Sólo IV.

1.4. En relación a las co-rutinas y los iteradores, se puede decir de manera general que

a) Un iterador es una co-rutina que sólo puede ceder control a su llamador.

b) Si un lenguaje dispone de continuaciones, entonces tiene ambos automáticamente.

c) Si un lenguaje compilado los provee, almacena los registros de activación en área estática.

d) Si un lenguaje tiene iteradores recursivos, entonces tiene co-rutinas recursivas.

1.5. En la mayoría de los lenguajes orientados a objetos con verificación dinámica de tipos (Perl, Python, Ruby) todas las llamadas a métodos se hacen con asociación dinámica. ¿Por qué?

a) Para poder permitir técnicas como despacho doble o despacho múltiple.

b) Porque es menos complicado que usar `self` como argumento implícito en cada llamada.

c) Porque es más práctico construir *v-tables* en todas las clases, en lugar de selectivamente.

d) Porque la introspección permite que cualquier clase o instancia cambie dinámicamente.

1.6. Considere las siguientes declaraciones en un lenguaje orientado a objetos que maneja las variables con modelo de valor y en el que la herencia corresponde a la noción de subtipo. Suponga que en ese lenguaje se tiene una jerarquía de dos clases `Baz` subclase de `Bar`, en la que `Bar` es abstracta y `Baz` es concreta. Considere las siguientes declaraciones

I. `Bar *m1(Baz p)`

II. `Baz Bar::m2(Bar *p)`

III. `Bar *Baz::m3(Baz *q,Bar r)`

IV. `Baz::Baz(Bar *b)`

¿Cuales de las declaraciones son **válidas**?

a) Sólo I, II y IV.

b) Sólo I, II y III.

c) Sólo II, III y IV.

d) Sólo III.

Pregunta 2 - 6 puntos

Considere el siguiente programa escrito en pseudocódigo

```
qux : int;

proc grok( foo, bar, baz : int )
  foo := foo + bar + baz + qux;
  print(qux);
  bar := foo + bar + baz + qux;
  print(qux);
  baz := foo + bar + baz + qux;
end

begin
  qux := 2;
  grok(qux,qux,qux)
  print(qux)
end
```

Ejecute el programa aplicando las reglas de pasaje de parámetros para cada uno de los casos especificados en la siguiente tabla. Suponga que el lenguaje pasa los parámetros en orden de derecha a izquierda.

Escriba en la columna **Salida del Programa** los resultados emitidos durante cada corrida.

Parámetro	**Pasaje por**	**Salida del Programa**
`foo`	Referencia	`8`
`bar`	Copia	`8`
`baz`	Referencia	`26`
`foo`	Copia	`2`
`bar`	Referencia	`14`
`baz`	Valor	`8`

Pregunta 3 - 8 puntos

Escriba la función

```
partition :: (a -> Bool) -> [a] -> ([a],[a])
```

tal que aplique un predicado a cada elemento de una lista, separando aquellos que cumplen el predicado, de aquellos que no lo cumplen, *preservando* el orden relativo de los elementos. Por ejemplo,

```
> partition even [1,4,2,2,3,6,7,4,2,6]
([4,2,2,6,4,2,6],[1,3,7])
```

Nota: para obtener los ocho (8) puntos *debe* escribir la función usando funciones de orden superior. Si la expresa con recursión de cola sólo recibirá cuatro (4) puntos. Si la expresa con recursión directa sólo recibira dos (2) puntos. Su solución debe recorrer la lista *una sola vez*.

```
partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = foldr (select p) ([],[]) xs
  where select p x (ts,fs) | p x       = (x:ts,fs)
                           | otherwise = (ts, x:fs)
```

Pregunta 4 - 8 puntos

- **(3 puntos)** Implante el predicado `divisores(N,Divisores)` que triunfe *exactamente una vez* si `Divisores` corresponde a la lista de divisores de `N`, ordenados de menor a mayor. El predicado debe fallar si `N` está unificado con cualquier cosa que no sea un número entero positivo. Por ejemplo,

```
?- divisores(6,Divisores).
Divisores = [1,2,3,6]
yes
?- divisores(6,[1,2,3,6]).
yes
?- divisores(6,[1,2,3]).
no
?- divisores(0,Divisores).
no
```

  Nota: el predicado estándar Prolog `mod/2` representa el cálculo del resto de la división entera, de su primer argumento como dividendo y su segundo argumento como divisor.

- **(5 puntos)** Un número entero es perfecto cuando es igual a la suma de sus divisores propios. Implante el predicado `perfecto(N)` que triunfe si `N` es un número perfecto. Si `N` está instanciado con un número entero, el predicado debe triunfar *exactamente una vez* si el número es perfecto y fallar si no lo es. Si `N` es una variable libre, entonces el predicado debe producir los infinitos números perfectos aprovechando *backtracking*. Por ejemplo,

```
?- perfecto(6).
yes
?- perfecto(7).
no
?- perfecto(N).
N = 6 ? ;
N = 28 ? ;
N = 496 ?
```

```
divisores(N,Divisores) :- integer(N), N > 0,
                          go(N,N,[],Divisores).

go(1,_,A,[1|A]) :- !.
go(M,N,A,D) :- R  is mod(N,M), R = 0, !,
               M1 is M - 1, go(M1,N,[M|A],D).
go(M,N,A,D) :- M1 is M - 1, go(M1,N,A,D).
```

```
perfecto(N) :- integer(N), !,
               divisores(N,D),
               sum(D,T),
               N =:= T - N.
perfecto(N) :- var(N), !,
               natural(N1),
               perfecto(N1),
               N = N1.

natural(1).
natural(N) :- natural(N1), N is N1 + 1.

sum([],0) :- !.
sum([D|R],S) :- sum(R,T), S is D + T.
```

Pregunta 5 - 5 puntos extra

Considere la siguiente clase definida en Ruby

```
class Foo
  attr_accessor :value
  def m1
    @value = 4
  end
  def m2
    m1
    @value > 4
  end
  def m3
    @value = 4
    @value > 4
  end
  def m4
    self.value = 4
    @value > 4
  end
end
```

Defina una clase `Bar` como subclase de `Foo` tal que:

- **(2 puntos)** Evaluar `Bar.new.m2` retorna `true`, pero el método `m2` **no** fue redefinido en `Bar`.
- **(3 puntos)** Evaluar `Bar.new.m4` retorna `true`, pero el método `m4` **no** fue redefinido en `Bar`.

```
class Bar < Foo
  def m1
    @value = 5
  end
  def value=(v)
    @value = 42
  end
end
```