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

Carnet:_____

Nombre:_____

Examen I

(25 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	**Total**
12 puntos	6 puntos	7 puntos	**25 puntos**

Pregunta 1 - 12 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 dos (2) puntos. Tres subpreguntas incorrectas eliminan una correcta.

1.1. El diseño de un lenguaje de programación establece que en todas sus implantaciones los números enteros deben tener la máxima precisión nativa de la plataforma anfitriona y su aritmética debe efectuarse utilizando complemento a dos. Tal decisión de diseño puede favorecer o no las siguientes características del lenguaje:

I. La eficiencia de ejecución de programas escritos en el lenguaje.
II. La portabilidad de la implantación del lenguaje.
III. La portabilidad de los programas escritos en el lenguaje.

¿Cuáles **perjudica**?

a) I y II.
b) I y III.
c) II y III.
d) Todas.

1.2. Un lenguaje compilado, con alcance estático y funciones anidadas necesita:

a) Que el ejecutable maneje una tabla de símbolos del estilo Tabla Central de Referencias.
b) Que su compilador maneje una tabla de símbolos del estilo Tabla Central de Referencias.
c) Que el ejecutable siga la cadena estática para acceder a símbolos no locales.
d) Que el compilador siga la cadena estática para acceder a símbolos no locales.

1.3. C es un lenguaje compilado, con alcance estático, funciones de segunda clase, sin clausuras. Por lo tanto para una variable local a un procedimiento **siempre** es cierto:

a) Se elabora en la pila de ejecución.
b) Está en una dirección fija de memoria.
c) Su nombre sólo es visible en el alcance del procedimiento.
d) Ninguna de las anteriores.

1.4. Considere un lenguaje imperativo, de propósito general, con alcance estático y sin clausuras. ¿Cuál de las siguientes verificaciones **no** puede completarse enteramente de forma estática?

a) Verificar que toda variable recibe valores cónsonos con su tipo.
b) Verificar que toda excepción pueda ser atrapada.
c) Verificar que el valor de una variable nunca es cambiado.
d) Todas pueden verificarse de forma estática.

1.5. En relación a módulos como administrador y módulos como tipo podemos afirmar:

a) Los primeros son similares a la orientación a objetos, y los segundos son similares a las librerías tradicionales.

b) Los primeros necesitan tener espacio de nombre abierto, pero los segundos pueden operar con espacio abierto o cerrado indistintamente.

c) Se diferencian sintácticamente, pues el código ejecutable final es prácticamente idéntico.

d) Se diferencian operacionalmente, pues el código ejecutable final de los módulos como administrador es más simple.

1.6. El ambiente de ejecución de un lenguaje con almacenamiento en *heap* requiere un manejador de memoria para administrar el espacio adicional. Considere las siguientes afirmaciones en relación a cualquier manejador de memoria:

I. *Best Fit* siempre es mejor que *First Fit* para controlar la fragmentación interna.

II. La fragmentación externa depende de la selección de tamaños para los bloques.

III. La variante Fibonacci para el "Buddy System" mejora la fragmentación interna.

¿Cuáles afirmaciones son *ciertas*?

a) I y II solamente.

b) II y III solamente.

c) I y III solamente.

d) Todas son ciertas.

Pregunta 2 - 6 puntos

Considere el siguiente programa escrito en pseudocódigo:

```
int foo = 3
int bar = 2

proc B(proc wee, int bar)
  if bar > 1
    B(wee,bar-1)
  wee(bar)
  print(bar)

proc A(int baz, int foo)
  proc fudge(int qux)
    bar := foo + bar + qux

  int bar = baz
  B(fudge,foo)
  print(bar)

main
  A(foo,2)
  print(bar)
end
```

Ejecute el programa aplicando las reglas de alcance y construcción de clausuras para cada uno de los casos especificados en la siguiente tabla. Escriba en la columna **Salida del Programa** los resultados emitidos durante cada corrida.

Alcance	**Asociación**	**Salida del Programa**
Estático	Profunda (*Deep*)	1 2 3 9
Dinámico	Profunda (*Deep*)	1 5 10 2
Dinámico	Superficial (*Shallow*)	4 6 3 2

Pregunta 3 - 7 puntos

Considere el siguiente fragmento en lenguaje C, correspondiente al cuerpo de alguna subrutina:

```
{ int a, b, c;
  ...
  { int d, e;
    ...
    { int f, g;
      ...
    }
    ...
  }
  ...
  { int h, i, j;
    ...
  }
  ...
}
```

Suponga que cada variable entera ocupa 4 bytes y que se desea reservar espacio en el registro de activación para las variables locales. En este caso, basta planificar espacio sólo para 7 enteros, 28 bytes, ubicando las variables en los *offsets* a–0, b–4, c–8, d–12, e–16, f–20, g–24, h–12, i–16 y j–20.

Considere el tipo de datos Haskell

```
data Bloque = B [String] [Bloque]
```

para modelar la estructura de bloques anidados. La primera lista incluye los nombres de variables definidas al inicio del bloque, y la segunda lista contiene los bloques anidados. Puede suponer que nunca se repiten los nombres de variable en toda la estructura de bloques. Así, escriba la función

```
espacio :: Bloque -> (Integer,[(String, Integer)])
```

que recibe una estructura de bloques anidados y determina la cantidad *mínima* de espacio requerido, además de asignar los desplazamientos para cada símbolo usando solapamiento. Para el fragmento de ejemplo, podríamos definir

```
test = B ["a","b","c"] [B ["d","e"] [B ["f","g"] []], B ["h","i","j"] []]
```

y luego evaluar

```
> espacio test
(28, [("a",0), ("b",4), ("c",8), ("d",12), ("e",16), ..., ("j",20)]
```

Nota: puede usar cualquier función del *Preludio* y escribir funciones auxiliares si le conviene.

Una solución compacta algo idiomática, aprovechando las funciones del Preludio para ahorrar tiempo podría ser

```
espacio block = size (offset 0 block)
  where offset d (B []     bs) = (concat . map (offset d)) bs
        offset d (B (v:vs) bs) = (v,d) : offset (d+4) (B vs bs)
        size xs                = (4 + maximum (map snd xs), xs)
```

Una solución con recursión directa simple y sincera, sin mayor preocupación por la eficiencia en espacio y tiempo podría ser

```
espacio (B vars nested) = doit vars nested 0 []
  where
    doit vs ns t d = combine doit ns t' t' d'
      where
        (t',d') = offset vs t d

        offset (v:vs) t d = offset vs (t+4) (d ++ [(v,t)])
        offset []     t d = (t,d)

        combine f []           st mt dd = (mt,dd)
        combine f ((B v n):bs) st mt dd = combine f bs st mt' (dd ++ db)
           where (tb,db) = doit v n st []
                 mt'     = max tb mt
```

Esa misma solución, escrita usando el tipo `Lista a`, requiere reescribir el tipo `Bloque` como mostré antes del examen, y también el valor de prueba `test` (la indentación es mía para facilitar su comprensión).

```
data Lista a = Vacia | Cons a (Lista a)
             deriving (Show,Eq)

data Bloque = B (Lista String) (Lista Bloque)
            deriving (Show)

test :: Bloque
test = B (Cons "a" (Cons "b" (Cons "c" Vacia)))
         (Cons (B (Cons "d" (Cons "e" Vacia))
                  (Cons (B (Cons "f" (Cons "g" Vacia)) Vacia)
                    Vacia)
               )
           (Cons (B (Cons "h" (Cons "i" (Cons "j" Vacia))) Vacia)
             Vacia)
         )
```

Hechos esos cambios, la solución que Ud. debía escribir quedaría como

```
espacio (B vars nested) = doit vars nested 0 Vacia
  where
    doit vs ns t d = combine doit ns t' t' d'
      where
        append Vacia ys       = ys
        append (Cons x xs) ys = Cons x (append xs ys)

        (t',d') = offset vs t d

        offset (Cons v vs) t d = offset vs (t+4) (append d (Cons (v,t) Vacia))
        offset Vacia       t d = (t,d)

        combine f Vacia             st mt dd = (mt,dd)
        combine f (Cons (B v n) bs) st mt dd = combine f bs st mt' (append dd db)
          where (tb,db) = doit v n st Vacia
                mt'     = max tb mt
```

Note que en la solución anterior **todo** está construido exclusivamente con recursión explícita y *pattern matching* equivalente a escribir funciones por caso. Todas las funciones fueron definidas como locales (`where`) por costumbre y estilo adecuado, pero no hay problema en escribirlas como funciones en el alcance principal.