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

Carnet:_____

Nombre:_____

Examen II

(35 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	Pregunta 4	**Total**
21 puntos	4 puntos	5 puntos	5 puntos	**35 puntos**

Pregunta 1 - 21 puntos

Esta pregunta consta de siete (7) subpreguntas de selección, numeradas de 1.1 a 1.7. 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. Considere la siguiente declaración de dos variables apuntador en un lenguaje tipo Pascal

```
foo, bar : ^T
```

donde `T` es un tipo cualquiera. Suponga que se está implantando detección de referencias colgadas (*dangling references*) mediante llaves y cerraduras (*locks and keys*). En ese caso, cada apuntador en realidad es un registro que contiene el apuntador propiamente dicho y la llave de acceso, en ese orden; y cada objeto del *heap* es un registro con la llave de acceso, el apuntador al descriptor de tipos del objeto y el valor del objeto en sí, en ese orden.
Considerando que:

- `foo` y `bar` están almacenadas en las direcciones de memoria α y β.
- Cada llave de acceso ocupa 4 bytes y cada apuntador ocupa 8 bytes.
- `null` es una dirección inválida correspondiente a un apuntador nulo.
- `*X` se refiere al *r-value* de la dirección de memoria `X`,
- `X := Y` significa almacenar el *r-value* de `Y` en el *l-value* denotado por `X`.

¿cuáles acciones deben ser ejecutadas a bajo nivel para la instrucción `bar := foo`?

a) Si $*\alpha \neq null \wedge *(\alpha + 8) \neq **\alpha$, error de referencia colgada;
en caso contrario $\beta := *\alpha$ y $\beta + 8 := *(\alpha + 8)$

b) Si $\alpha \neq null \wedge *(\alpha + 8) \neq *\alpha$, error de referencia colgada;
en caso contrario $\beta := \alpha$ y $\beta + 8 := *(\alpha + 8)$

c) Si $*\beta \neq null \wedge *(\beta + 8) \neq **\beta$, error de referencia colgada;
en caso contrario $\alpha := *\beta$ y $\alpha + 8 := *(\beta + 8)$

d) Si $\beta \neq null \wedge *(\beta + 8) \neq *\beta$, error de referencia colgada;
en caso contrario $\alpha := \beta$ y $\alpha + 8 := *(\beta + 8)$

1.2. Continúe considerando la información de la pregunta anterior. Suponga ahora que, además de las llaves y cerraduras (*locks and keys*) para detección de referencias colgadas (*dangling references*), agregamos el uso de contadores de referencias (*reference counts*) para facilitar la recolección de basura (*garbage collection*). Ahora, cada objeto en el *heap* tendrá el contador de referencias (de 4 bytes) ubicado después del descriptor de tipos, y el valor del objeto en sí, en ese orden.
En la asignación

```
bar := foo
```

¿cuáles acciones de bajo nivel deben ser ejecutadas para mantener la consistencia de los contadores de referencias? Suponga que ya se determinó que ninguna de las dos referencias es nula ni está colgada.

a) Decrementar $*(*\beta + 12)$;
si $*(*\beta + 12) = 0$, liberar el objeto inmediatamente;
incrementar $*(*\alpha + 12)$.

b) Decrementar $*(*\beta + 12)$;
si $*(*\beta + 12) = 0$, liberar el objeto inmediatamente;
incrementar $*(*\alpha + 12)$;
repetir recursivamente para toda dirección mencionada en $*(*\beta + 4)$.

c) Decrementar $*(*\alpha + 12)$;
si $*(*\alpha + 12) = 0$, liberar el objeto inmediatamente;
incrementar $*(*\beta + 12)$;
repetir recursivamente para toda dirección mencionada en $*(*\alpha + 4)$.

d) Ninguna de las anteriores.

1.3. Continúe considerando la información de las dos preguntas inmediatamente anteriores. Se desea ahora que señale las acciones que deben ser ejecutadas a bajo nivel para la instrucción

```
bar^ := foo^
```

entendiendo que `^` es el operador de indirección en el lenguaje y suponiendo que ya se verificó que ninguna de las dos referencias es nula ni está colgada. Así mismo, debe suponer que el lenguaje emplea *modelo de valor* con *copia profunda*.

a) $*(*\alpha + 16 + k) := *(*\beta + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

b) $*\alpha + 16 + k := *(*\beta + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

c) $*(*\beta + 16 + k) := *(*\alpha + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

d) $*\beta + 16 + k := *(*\alpha + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

1.4. En las implantaciones de C previas a C99, las conversiones implícitas automáticas (*coercions*) eran, cuando menos, controversiales, como muestra el siguiente ejemplo

```
short int s;
unsigned long l;
s = l;  /* bits menos significativos de l trasladados a s
           e interpretados como si tuviera signo */
```

¿Cuál de las siguientes afirmaciones es falsa?

a) Un número positivo mayor al máximo positivo almacenable en un `short int` puede terminar siendo un número negativo.

b) Un número positivo mayor al máximo positivo almacenable en un `short int` puede terminar siendo un número positivo menor al original.

c) Si n es la cantidad de bits utilizada para almacenar un `short int`, la coerción produce un resultado semánticamente adecuado para enteros entre 0 y $2^{(n-1)}-1$, ambos extremos incluidos.

d) Si n es la cantidad de bits utilizada para almacenar un `short int`, la coerción produce un resultado semánticamente adecuado para enteros entre 0 y $2^n - 1$, ambos extremos incluidos.

1.5. Considere las siguientes definiciones de tipos en un lenguaje con soporte nativo para conjuntos, en el cual + corresponde a la unión, $*$ corresponde a la intersección y los conjuntos literales se expresan listando sus elementos entre llaves.

```
var a : set of 5..10;
    b : set of 20..40;
    c : set of Y0..Y1;
    i : 15..30;

c := a + b * {1..5, i}
```

Sea `set of` x_0 `..` x_1, con $x_0 \leq x_1$, el tipo inferido para el lado *derecho* de la expresión. Para ciertas combinaciones de x_0, x_1, y_0 y y_1 es posible determinar *estáticamente* si los tipos involucrados en la asignación son adecuados o no. ¿Cuál de las siguientes fórmulas determina las combinaciones para las cuales un compilador **no** puede garantizar ni la ausencia ni la presencia de un error de tipos?

a) $y_1 < x_0 \vee x_1 < y_0$

b) $y_0 \leq x_0 \wedge x_1 \leq y1$

c) $(x_0 \leq y_1 \wedge y_1 < x_1) \vee (x_0 < y_0 \wedge y_0 \leq x_1)$

d) Ninguna de las anteriores.

1.6. Considere la siguiente declaración de un arreglo en un lenguaje con modelo de valor

```
foo : array [low .. high] of array [1.. 42] of T
```

Suponga que el arreglo es variable local de alguna subrutina, con tiempo de vida convencional, siendo elaborado al ingresar a la subrutina y destruido al retornar. Así mismo, suponga que los valores de `low` y `high` serán conocidos al momento de elaboración dinámica del arreglo (*elaboration time constants*).
Se desea que señale la fórmula de cálculo que debe ser utilizada para determinar el **valor** de `foo[i][j]` suponiendo:

- Tanto `i` como `j` están en el rango adecuado para cada una, y están almacenadas en las direcciones α y β respectivamente.
- La variable `foo` tiene un *dope vector* en la dirección de memoria δ.
- El *dope* vector contiene los valores enteros para los límites `low`, `high` y el apuntador al contenido del arreglo, en ese orden.
- El arreglo está representado en *row pointer layout*.

Suponiendo que tanto los enterors como los apuntadores ocupan 4 bytes de memoria cada uno, entonces la fórmula adecuada sería:

a) $*(*(\delta + 8) + (*\alpha - *\delta) \times 4) + (*\beta - 1) \times sizeof(T)$
b) $*(*(\delta - 8) + (*\alpha - *\delta) \times 4) + (*\beta - 1) \times sizeof(T)$
c) $*(*(*(\delta - 8) + (*\alpha - *\delta) \times 4) + (*\beta - 1) \times sizeof(T))$
d) $*(*(*(\delta + 8) + (*\alpha - *\delta) \times 4) + (*\beta - 1) \times sizeof(T))$

1.7 Considere el siguiente fragmento de código de un lenguaje con iteradores reales, que además dispone de la notación y operaciones de listas de Haskell sobre el tipo `List`.

```
grok(n, m : Int) {
  if (n < m) {
    while (l = grok(n+1,m)) {
      yield [n] ++ l
      yield l
    }
  } else {
    yield []
  }
}
```

Al invocar `grok(40,43)`, ¿cuál será la *segunda* lista de dos elementos que será emitida por el iterador?

a) `[41,42]`
b) `[42,41]`
c) `[40,42]`
d) `[42,40]`

Pregunta 2 - 4 puntos

A continuación se presenta un procedimiento recursivo en C

```
void again(int n)
{
  if (B0)
    I0
  else if (B1)
    I1
  else if (B2) {
    I2;
    again(E0);
    again(E1);
  } else if (B3) {
    again(E2);
    I3;
  } else {
    I4;
    again(E3);
  }
}
```

En este procedimiento, `B0`, `B1`, `B2` y `B3` son expresiones booleanas cualesquiera; `I0`, `I1`, `I2`, `I3` e `I4` son instrucciones cualesquiera; y `E0`, `E1`, `E2` y `E3` son expresiones enteras cualesquiera. Sólo puede suponer que no hay ninguna llamada directa o indirecta al procedimiento `again` en las instrucciones o expresiones mencionadas.
Simplifique el procedimiento para convertir la recursión de cola en iteración estructurada, sin modificar la semántica del mismo. Puede definir variables adicionales si cree que son convenientes.

Usando control estructurado

```
void again(int n) {
  while (42) {
    if (B0) {
      I0; return
    } else if (B1) {
      I1; return
    } else if (B2) {
      I2;
      again(E0);
      n = E1; continue
    } else if (B3) {
      again(E2);
      I3; return
    } else {
      I4; n = E3
    }
  }
}
```

Usando invariante combinada

```
void again(int n) {
  bool b0, b1, b2;
  while (!(b0 = B0) && !(b1 = B1) && ((b2 = B2) || !B3)) {
    if (b2) {
      I2; again(E0); n = E1
    } else {
      I4; n = E3
    }
  }
  if (b0)
    I0
  else if (b1)
    I1
  else {
    again(E2);
    I3
  }
}
```

Pregunta 3 - 5 puntos

Una **Tríada Pitagórica** es una tripleta de números enteros (x, y, z) tal que $x^2 + y^2 = z^2$. Aproveche *exclusivamente* listas por comprensión para escribir la función Haskell

```
triads :: Integer -> [(Integer,Integer,Integer)]
```

que genera la lista de Tríadas Pitagóricas tales que ninguno de sus componentes es mayor que el primer argumento de la función, y en la lista no hay tríadas repetidas independientemente del orden de sus componentes, esto es, si en la lista aparece la tríada `(3,4,5)` entonces **no** debe aparecer `(4,3,5)`.

```
triads n = [ (x,y,z) | x <- [1..n],
                       y <- [x..n],
                       z <- [y..n],
                       x^2 + y^2 == z^2 ]
```

Pregunta 4 - 5 puntos

Use *exclusivamente* recursión de cola para escribir la función Haskell

```
reverse :: [a] -> [a]
```

que recibe una lista arbitraria *finita* y produce una lista pero con los elementos en el orden inverso, i.e.

```
> reverse [1,2,3,4]
[4,3,2,1]
```

Si su solución emplea el `fold` adecuado, recibirá dos (2) puntos adicionales. Sólo debe escribir *una* solución.

Usando recursión de cola

```
reverse = doit []
  where doit rs []     = rs
        doit rs (x:xs) = doit (x:rs) xs
```

Usando el `fold` adecuado

```
reverse = foldl (flip (:)) []
```