Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Septiembre-Diciembre 2013

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 `foo := bar`?

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

```
foo := bar
```

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

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

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

d) Ninguna de las anteriores.
(El *orden* de las operaciones es importante – si se libera el objeto inmediatamente, ya no se tiene acceso a los apuntadores internos para la repetición recursiva).

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

```
fooˆ := barˆ
```

entendiendo que ˆ 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 una discusión acerca de las ventajas que tiene la verificación estática sobre la verificación dinámica de tipos, se emiten los siguientes argumentos:

I. La verificación estática permite generar código ejecutable más rápido.

II. La verificación estática asiste en la detección de errores, reduciendo el volumen de pruebas sobre el código.

III. La verificación estática siempre es sólida e incompleta, pero la dinámica a veces no es sólida.

¿Cuáles de esos argumentos son ciertos?

a) I y II.

b) I y III.

c) II y III.

d) Todos.

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 11..20;
    b : set of 11..25;
    c : set of ?..?;
    i : 10..25;

c := c * a + b * {11..15, i}
```

Entonces es cierto que

a) Si se declara `c : set of 0..15`, entonces el compilador genera un error estático porque la asignación sería imposible.

b) Si se declara `c : set of 0..15`, entonces el compilador debe generar código para verificar el resultado de la expresión en el lado derecho antes de hacer la asignación.

c) Si se declara `c : set of 0..10`, entonces el compilador genera código que hace la asignación directamente sin necesidad de verificación.

d) Si se declara `c : set of 0..10`, entonces el compilador genera código que verifica el valor resultado de la expresión en el lado derecho antes de hacer la asignación.

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 la **dirección** 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) $*(*(\delta - 8) + *\alpha \times (*(\delta - 4) - *\delta + 1) + (*\beta - 1) \times sizeof(T))$
b) $*(*(*(\delta - 8) + (*\alpha - *\delta) \times 4) + (*\beta - 1) \times sizeof(T))$
c) $*(\delta - 8) + *\alpha \times (*(\delta - 4) - *\delta + 1) + (*\beta - 1) \times sizeof(T)$
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(s : List) {
  if (s == []) {
    yield []
  } else {
    while ( e = grok( tail(s) ) ) {
      yield head(s) : e;
      yield e;
    }
  }
}
```

Al invocar `grok([4,2,1])`, ¿cuál será la *segunda* lista de dos elementos que será emitida por el iterador?

a) `[4,1]` – el iterador emite `[4, 2, 1] [2, 1] [4, 1] [1] [4, 2] [2] [4] []`
b) `[1,2]`
c) `[1,4]`
d) `[4,2]`

Pregunta 2 - 4 puntos

A continuación se presenta un procedimiento recursivo en C

```
void regla(int izq, int der, int alt)
{
  int mid;
  if (alt > 0) {
    mid = (izq + der) / 2;
    printf("(%d,%d)",mid,alt);
    regla(izq,mid,alt - 1);
    regla(mid,der,alt - 1);
  }
}
```

Simplifique el procedimiento `regla` convirtiendo la recursión de cola en una iteración estructurada.

Observamos que solamente la última llamada a `regla()` manifiesta recursión de cola, de manera que se puede convertir en una iteración estructurada:

- Notando que basta hacer la sustitución de variables para los parámetros. En la última llamada recursiva, el `mid` *actual* sería el argumento `izq` para la llamada recursiva, `der` permanecería inalterado, y `alt` se decrementaría en uno antes de ser usado como argumento `alt` en la llamada recursiva.
- El selector inicial se usa para detectar el caso base, y puede reemplazarse por un ciclo que use el mismo condicional.

```
void regla(int izq, int der, int alt)
{
  int mid;
  while (alt > 0) {
    mid = (izq + der) / 2;
    printf("(%d,%d)",mid,alt);
    regla(izq,mid,alt - 1);
    izq = mid; alt = alt - 1;
  }
}
```

Pregunta 3 - 5 puntos

En la Teoría de Números, un número entero positivo es **Perfecto** cuando es igual a la suma de sus divisores propios. Así, el primer número perfecto es el 6, porque sus divisores propios son 1, 2 y 3, y el resultado de $1 + 2 + 3$ es precisamente 6. Aproveche *exclusivamente* listas por comprensión para escribir la función Haskell

```
perfectos :: [Integer]
```

que genera la lista infinita de números perfectos.
Basta usar la definición de `divisores` que mostré en clase, pero evitando incluir al propio número en la lista de generación, para que no sea considerado a la hora de sumar.

```
perfectos :: [Integer]
perfectos = [ n | n <- [1..],
                  sum [ d | d <- [1..n-1], n `mod` d == 0 ] == n ]

ghci> take 5 perfectos
[6,28,496,8128,33550336]
```

Pregunta 4 - 5 puntos

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

```
halve :: [a] -> ([a],[a])
```

que recibe una lista arbitraria *finita* y la divide en dos listas tales que sus longitudes sean iguales o a lo sumo se diferencien en uno. La intención de la función es dividir la lista original en dos mitades. Si escribe su solución usando el `fold` adecuado, recibirá dos (2) puntos adicionales. Sólo debe escribir *una* solución.

Notamos que una lista puede ser de longitud par o impar, así que basta tomar siempre *dos* elementos y distribuirlos entre las dos listas resultantes. La recursión tendrá *dos* casos base: la lista vacía y la lista con exactamente un elemento. En el caso recursivo, siempre hay al menos dos elementos para repartir. Usaremos una tupla con listas como argumento de acumulación para la función auxiliar recursiva de cola

```
halve :: [a] -> ([a],[a])
halve xs = go ([],[]) xs
  where go (l,r) []       = (l,r)
        go (l,r) [x]      = (x:l,r)
        go (l,r) (x:y:rs) = go (x:l,y:r) rs
```

El único `fold` adecuado para este escenario es `foldl` porque las listas son finitas y el procedimiento de división en dos partes puede aplicarse por igual de izquierda a derecha y viceversa, siendo preferible el primero. Más aún, usaremos `Data.List.foldl'` que es ligeramente más eficiente

```
halve' :: [a] -> ([a],[a])
halve' xs = foldl' pingpong ([],[]) xs
  where pingpong (l,r) x = (r,x:l)
```