Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2008

Examen II

(30 puntos)

Nombre:

Carnet:

1. **(12 puntos)** Considere cuidadosamente las siguientes cuestiones y seleccione exactamente una respuesta de las alternativas que se presentan. Cada cuestión contestada correctamente vale **dos (2) puntos** pero una cuestión contestada incorrectamente **resta un (1) punto**. Si Ud. selecciona más de una opción, la pregunta se considera contestada **incorrectamente.**

   (a) Considere la siguiente declaración de dos variables de tipo 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 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 y cada dirección ocupa 4 bytes.
   - `nil` es una dirección inválida correspondiente a un apuntador nulo.
   - `*X` se refiere al *contenido* de la dirección de memoria `X`.
   - `X:=Y` significa almacenar el *valor* `Y` en la *dirección* `X`.

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

   i. Si $*\beta \neq nil \wedge *(\beta + 4) \neq *\beta + 4$, error de referencia colgada; en caso contrario, $\alpha := \beta$ y $\alpha + 4 := *(\beta + 4)$.
   ii. Si $*(\beta + 4) \neq nil \wedge *\beta \neq **(\beta + 4)$, error de referencia colgada; en caso contrario, $\alpha := *\beta$ y $\alpha + 4 := *(\beta + 4)$.
   iii. Si $*\beta \neq nil$, error de referencia colgada; en caso contrario, $\alpha := **\beta$ y $\alpha + 4 := **(\beta + 4)$.
   iv. Si $*\beta \neq nil \wedge **\beta \neq *(\beta + 4)$, error de referencia colgada; en caso contrario, $\alpha := *\beta$ y $\alpha + 4 := *(\beta + 4)$.

(b) Continúe considerando la información de la pregunta inmediatamente anterior. Se desea ahora que señale las acciones que deben ser ejecutadas a bajo nivel para la instrucción

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

entendiendo que "^" es el operador de indirección en el lenguaje y asumiendo que ya se verificó que ninguna de las dos referencias es nula ni está colgada. Así mismo, asuma que el lenguaje emplea *modelo de valor* y que esta asignación es permitida aún si el tipo base T es compuesto.

i. $*(\alpha + 4 + k) := **(\beta + 4 + k)$ con k igual a $0, 4, 8, \ldots$ hasta alcanzar el tamaño de T.

ii. $\alpha := *\beta$, $*(\alpha+4) = **(\beta+4)$ y $*\alpha+4+k := *(*\beta+4+k)$ con k igual a $0, 4, 8, \ldots$ hasta alcanzar el tamaño de T.

iii. $*\alpha + 4 + k := *(*\beta + 4 + k)$ con k igual a $0, 4, 8, \ldots$ hasta alcanzar el tamaño de T.

iv. $*\alpha + k := *(*\beta + k)$ con k igual a $0, 4, 8, \ldots$ hasta alcanzar el tamaño de T.

(c) Continúe considerando la información de las dos preguntas anteriores. 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* sería un registro con la clave de acceso, el contador de referencias (también de 4 bytes) y por último el valor del objeto en sí, en ese orden.

En la asignación

```
foo:=bar
```

¿qué 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.

i. Incrementar el contenido de la dirección $*\alpha + 4$,
decrementar el contenido de la dirección $*\beta + 4$,
liberar memoria del *heap* si hace falta.

ii. Incrementar el contenido de la dirección $*\beta + 4$,
decrementar el contenido de la dirección $*\alpha + 4$,
liberar memoria del *heap* si hace falta.

iii. Incrementar el contenido de la dirección $*\alpha + 4$,
incrementar el contenido de la dirección $*\beta + 4$.

iv. Decrementar el contenido de la dirección $*\alpha + 4$,
liberar memoria del *heap* si hace falta,
decrementar el contenido de la dirección $*\beta + 4$,
liberar memoria del *heap* si hace falta.

(d) Considere la siguiente declaración de un arreglo bi-dimensional:

```
m : array [i0..s0] of array [i1..s1] of T
```

donde i_0, s_0, i_1 y s_1 son constantes enteras de valor conocido estáticamente y `T` es un tipo cualquiera. Suponga que la base de `m` ha sido almacenada en la dirección de memoria α y que las variables enteras `i` y `j` están almacenadas en las direcciones β y γ respectivamente. Si el lenguaje de programación almacena los arreglos usando listas de apuntadores a filas (*row-pointer layout*) y cada apuntador ocupa 4 bytes, ¿cuál es la fórmula para determinar la dirección de `m[i][j]`?

i. $*\alpha + (*\beta - i_0) \times 4 + (*\gamma - i_1) \times 4$

ii. $*(*\alpha + (*\beta - i_0) \times 4) + (*\gamma - i_1) \times 4$

iii. $*(*\alpha + (*\beta - i_0) \times (s_1 - i_1 + 1)) + (*\gamma - i_1) \times 4$

iv. $*(*\alpha + (*\beta - i_0 + 1) \times 4) + (*\gamma - i_1 + 1) * 4$

(e) Un *dope vector*

i. **Siempre** incluye el límite superior de todas las dimensiones del arreglo.

ii. **Siempre** contiene ambos límites, haciendo innecesario mantener el tamaño de cada dimensión.

iii. Debe contener **al menos** el límite inferior y el tamaño de todas las dimensiones.

iv. **Nunca** pueden omitirse los límites inferiores ni los tamaños de cada dimensión aún si algunos de ellos son conocidos a tiempo de compilación.

(f) Considere la siguiente declaración en un lenguaje con soporte nativo para conjuntos, en el cual los operadores $+$ y $*$ corresponden a la unión e intersección de conjuntos respectivamente

```
var foo : set of 'e'..'k';
    bar : set of 'p'..'y';
    baz : set of 'a'..'k';
    i   : 'a'..'z';
```

Considere la instrucción

```
baz := foo + (bar * ['j'..'n', i])
```

¿qué tipo de verificaciones y cuándo deben hacerse?

i. A tiempo de compilación se detecta que la instrucción es inconsistente en el uso de tipos.

ii. A tiempo de compilación se detecta que la intersección siempre será vacía, y como el tipo de `foo` está contenido en el tipo de `bar`, no es necesario hacer ninguna verificación a tiempo de ejecución.

iii. A tiempo de ejecución debe emitirse un error siempre y cuando `i` esté dentro del conjunto `['p'..'y']`.

iv. A tiempo de ejecución debe permitirse la asignación siempre y cuando `i` este fuera del conjunto `['o','z']`.

2. Listas por comprensión (*list comprehensions*):

   (a) **(5 puntos)** Un número positivo es **perfecto** si es igual a la suma de sus divisores, excluyéndose a sí mismo, i.e. el número 6 es perfecto pues $1 + 2 + 3 = 6$, pero el número 12 no es perfecto porque $1 + 2 + 3 + 4 + 6 \neq 12$. Escriba la función Haskell

   ```
   divisores :: Integer -> [Integer]
   ```

   que utilice listas por comprensión para determinar los divisores de un número entero, para luego utilizarla en la construcción de otra función Haskell

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

   que utilice listas por comprensión para generar la lista infinita de los números perfectos.

   (b) **(3 puntos)** Considere el siguiente acertijo aritmético, donde A, B, C y D son dígitos decimales y $*$ es el producto de números enteros.

   ```
   ABCD *
      4
   ----
   DCBA
   ```

   Escriba una función en Haskell

   ```
   soluciones :: [(Int,Int,Int,Int)]
   ```

   que encuentre todas las soluciones del problema aprovechando listas por comprensión.

3. Considere la siguiente declaración de algún lenguaje de programación

```
foo : array [0..7] of array [-7..0] of array [-4..4] of
      record
        a : char
        b : short
        c : char[2]
        d : integer
        e : float
        f : boolean
        union
          record
            g : integer
            h : short
            i : char[3]
            j : boolean
            k : short
          end record
          record
            l : short
            m : integer
            n : boolean
            o : char[5]
          end record
        end union
      end record
```

En el lenguaje los tipos de datos se definen

Tipo de Datos	Representación
`char`	1 byte
`boolean`	1 byte
`short`	2 bytes
`integer`	4 bytes
`float`	8 bytes

y por restricciones de la arquitectura de hardware subyacente, cada objeto del tipo básico T debe alinearse en una **dirección par múltiplo de la representación del tipo T** (note que ésto implica que la alineación de cada tipo de datos fundamental es diferente). Así mismo, la dimensión de cualquier registro debe ser múltiplo de 8 bytes.

(a) **(2 puntos)** ¿Cuánto espacio ocupa un elemento del arreglo? Muestre los desplazamientos de cada elemento para justificar su cálculo.

(b) **(1 punto)** ¿Cuál es el porcentaje de espacio desperdiciado por elemento?

(c) **(2 puntos)** ¿Cuánto espacio ocupa todo el arreglo `foo` en bytes?

(d) **(5 puntos)** Suponga que la declaración de `foo` es para una variable global y el compilador le asignó la dirección base 1000. ¿Cuál es la dirección del elemento `foo[6,-2,3]`?

Nota: si solamente muestra los resultados (aunque sean correctos) no obtendrá puntos; es **imprescindible** justificar los resultados presentando la disposición en memoria de la estructura principal y todos los cálculos pertinentes de manera ordenada.