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`?

   La respuesta correcta es la **cuarta** alternativa, i.e. si $*\beta \neq nil \wedge **\beta \neq *(\beta + 4)$, error de referencia colgada; en caso contrario, $\alpha := *\beta$ y $\alpha + 4 := *(\beta + 4)$. Ese resultado se deduce de la Figura (pendiente).

   El condicional comienza por determinar si `bar` está apuntando a algo ($*\beta \neq nil$). Si eso es cierto (nótese la conjunción) y si la cerradura de lo apuntado **no** coincide con la llave que posee `bar` ($**\beta \neq *(\beta + 4)$) entonces hay un error de referencia colgante. Ahora si bar no apunta a nada ($*\beta = nil$ y se hace falsa la condición) o apunta a algo y tenemos la llave ($**\beta = *(\beta + 4)$ y se hace falsa la condición), entonces simplemente copiamos el apuntador de `bar` en `foo` ($\alpha := *\beta$) y copiamos la llava de `bar` en `foo` ($\alpha + 4 := *(\beta + 4)$).

(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.

La respuesta correcta es la **tercera** alternativa, i.e. $*\alpha+4+k := *(*\beta+4+k)$ con k igual a $0, 4, 8, \ldots$ hasta alcanzar el tamaño de T.

La operación en cuestión es una **copia** con la cual se persigue lograr que los contenidos de `bar` sean copiados dentro de `foo`. Sabemos que ambos son válidos, pues ninguno es nulo ni está colgado, por lo tanto **ambos** ya tienen espacio reservado y en consecuencia **ambos** tienen sus llaves y cerraduras correspondientes, de manera que hay que copiar byte a byte solamente los datos que constituyen las estructuras **sin** copiar las llaves. Como la llave está al comienzo de la estructura, y cada llave ocupa cuatro bytes, entonces debemos copiar los contenidos de lo apuntado por `bar` pero a partir del cuarto byte en adelante, en grupos de cuatro bytes, por tanto el dato a copiar es $*(*\beta + 4 + k)$ el cual debe ser almacenado en la dirección correspondiente $*\alpha + 4 + k$

(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.

La respuesta correcta es la **segunda** alternativa, i.e. incrementar el contenido de la dirección $*\beta + 4$, decrementar el contenido de la dirección $*\alpha + 4$, liberar memoria del *heap* si hace falta.

Si ninguna de las referencias es nula, ni está colgada, quiere decir que apuntan a espacios de memoria correctamente reservados los cuales contienen a partir del 4 byte el contador de referencias. Como `foo` va a apuntar a lo mismo que está apuntando `bar`, es necesario incrementar el contador dentro de la estructura apuntada por `bar` $(*\beta + 4)$ y como `foo` dejará de apuntar a la estructura a la que originalmente apuntaba, es necesario decrementar el contador dentro de la estructura apuntada por `foo` $(*\alpha + 4)$. Como es posible que la última operación lleve el contador a cero, entonces se realiza una recolección de basura para recuperar memoria del *heap* en caso de necesitarlo.

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

`m : array [`i_0`..`s_0`] of array [`i_1`..`s_1`] 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]`?

La respuesta correcta es la **segunda** alternativa, i.e. $*(*\alpha + (*\beta - i_0) \times 4) + (*\gamma - i_1) \times 4$, resultado que se deduce de la Figura (pendiente)

pues $*\alpha$ nos posiciona al comienzo del vector de apuntadores, donde debemos desplazarnos $(*\beta - i_0) \times 4$ posiciones para alcanzar el apuntador a la fila, el cual debe ser seguido hasta llegar a la base de la fila que al agregar el desplazamiento $(*\gamma - i_1) \times 4$ nos permite alcanzar la dirección deseada.

(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.

La respuesta correcta es la **tercera** alternativa, como se desprende de leer el primer párrafo de la descripción de los *Dope Vectors* en el libro de texto, página 364.

(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']`.

La respuesta correcta es la **tercera** alternativa. Primero debe observarse que el tipo de `foo` está enteramente contenido dentro del tipo de `baz`, de manera que la operación de unión puede realizarse independientemente del valor particular de `foo` pues está garantizado que es compatible con el tipo de `baz`. Sin embargo, es necesario determinar si el segundo operando de la unión puede producir algún problema; a tiempo de *compilación* se sabe que la intersección entre `bar` y la parte literal del conjunto (`'j'..'n'`) **siempre** es vacía pero que dependiendo del valor que pueda tener `i` a tiempo de *ejecución* va a ser necesario tomar alguna decisión. Como `i` es un caracter entre `'a'` y `'z'`, las cosas que pueden ocurrir a tiempo de *ejecución* son:

i. Si el valor de `i` está entre `'a'` y `'n'`, la intersección siempre es vacía y la asignación puede hacerse incondicionalmente.

ii. Si el valor de `i` está entre `'p'` y `'y'`, la intersección **no** es vacía (contendrá exactamente a `i`), pero la asiganción no puede hacerse porque el tipo de datos de la expresión no sería compatible con el tipo de `baz`.

iii. Si el valor de `i` es `'o'` o `'z'`, la intersección es vacía y la asignación puede hacerse incondicionalmente.

Y por ello sólo es necesaria la verificación indicada en la **tercera** alternativa.

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.

   ```
   divisores :: Integer -> [Integer]
   divisores n = [ i | i <- [1..n], n `mod` i == 0 ]
   ```

   Nótese que de acuerdo con la especificación del problema, la función `divisores` debe retornar **todos** los divisores de `n`, entre los cuales se incluye n. Entonces, para poder calcular si un número es perfecto debemos considerar la suma de sus divisores *excluyendo* al número, esto es, de la lista de divisores que procude la función anterior debemos ignorar el último. El preludio Haskell (que sugerí estudiaran) contiene las funciones `sum` que suma los elementos de una lista e `init` que devuelve todos los elementos de una lista, excepto el último, por tanto podemos escribir

   ```
   perfectos :: [Integer]
   perfectos = [ n | n <- [1..], (sum.init) n == n ]
   ```

   Por supuesto que aquellos que no estudiaron el preludio podían escribir una función auxiliar para hacer la suma y usar exactamente lo que mostré en clase, en particular

   ```
   perfectos:: [Integer]
   perfectos = [ n | n <- [1..], (sum.tail.reverse.divisores) n == n ]
   ```

   donde

   ```
   (sum.tail.reverse.divisores) n = sum(tail(reverse(divisores n)))
   ```

   (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.

   La solución coincide con los ejemplos de generación de alternativas y verificación de soluciones que presenté en clase. Básicamente hay que probar todas las alternativas de valuación para a, b, c y d, utilizando nuestros conocimientos básicos de representación númerica posicional

   ```
   soluciones = [(a,b,c,d) |
       a <- [0..9], b <- [0..9], c <- [0..9], d <- [0..9],
       4*(a*1000+b*100+c*10+d) == (d*1000+c*100+b*10+a) ]
   ```

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.

La restricción de alineación de los campos de las estructuras, combinada con las restricciones de alineación impuestas por el hardware **obligan** a que los tipos de datos tengan que alinearse según describe la siguiente tabla

Tipo	Debe alinearse en el byte
`char`	$0,+2,+4,+6,+8,\ldots,+2n$
`boolean`	$0,+2,+4,+6,+8,\ldots,+2n$
`short`	$0,+2,+4,+6,+8,\ldots,+2n$
`integer`	$0,+4,+8,+12,+16,\ldots,+4n$
`float`	$0,+8,+16,+24,\ldots,+8n$

Ahora, hemos de considerar la distribución de los campos en la estructura, comenzando por la parte común como lo muestra la siguiente tabla. Todos los offsets están en *bytes* como se acostumbra

al escribir representaciones de bajo nivel; los bytes denotados con # corresponden a espacio inútil consecuencia de la alineación.

Offset	Parte Común			
0	a	#	b	b
+4	c	c	#	#
+8	d	d	d	d
+12	#	#	#	#
+16	e	e	e	e
+20	e	e	e	e
+24	f	#	#	#

El registro contiene una parte variante, constituida como la unión de dos registros. Denotaremos como Variante A y Variante B respectivamente a los dos registros contenidos dentro de la unión, y sus representaciones en memoria deben superponerse de manera que sea posible acomodar al **más grande** de los dos. La Variante A comienza por el campo `g` de tipo `integer` que debe estar alineado en un múltiplo de 4, y la Variante B comienza por el campo `l` de tipo `short` que debe estar alineado en un múltiplo de 2; por tanto **ambas** partes variantes comienzan alineadas desde la posición 28 y se representan como muestra la siguiente tabla

Offset	Variante A				Variante B			
+28	g	g	g	g	l	l	#	#
+32	h	h	i	i	m	m	m	m
+36	i	#	j	#	n	#	o	o
+40	k	k	#	#	o	o	o	#
+44	#	#	#	#	#	#	#	#

Adicionalmente, se ha establecido la restricción que el registro total tenga dimensión múltiplo de 8, y por eso debe completarse hasta 48 bytes.

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

El espacio desperdiciado tiene dos casos

- Variante A: se desperdician 18 bytes de 48 para 37.5%.
- Variante B: se desperdician 18 bytes de 48 para 37.5%.

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

El arreglo tiene $U_0 - L_0 + 1 = 7 - 0 + 1 = 8$ filas, cada una de $U_1 - L_1 + 1 = 0 - (-7) + 1 = 8$ columnas, conteniendo $U_2 - L_2 + 1 = 4 - (-4) + 1 = 9$ posiciones. Cada posición debe contener un elemento de 48 bytes, por lo tanto

$$\begin{aligned} espacio(foo) &= filas \times columnas \times posiciones \times 48\,bytes \\ &= 8 \times 8 \times 9 \times 48\,bytes \\ &= 2^3 \times 2^3 \times 3^2 \times 3 \times 2^4\,bytes \\ &= 2^{10} \times 3^3 = 1024 \times 27 \\ &= 27648\,bytes \end{aligned}$$

(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]`?

Aplicamos directamente la fórmula discutida en clase para arreglos organizados en *row-major*, siendo la dirección deseada

$$base + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3$$

donde

$$\begin{aligned} S_3 &= 48\,bytes \\ L_3 &= -4 \end{aligned}$$

$$
\begin{aligned}
U_3 &= 4 \\
S_2 &= (U_3 - L_3 + 1) \times S_3 = (4 - (-4) + 1) \times 48 = 9 \times 48 \\
L_2 &= -7 \\
U_2 &= 0 \\
S_1 &= (U_2 - L_2 + 1) \times S_2 = (0 - (-7) + 1) \times 9 \times 48 = 8 \times 9 \times 48 \\
L_1 &= 0 \\
U_1 &= 7
\end{aligned}
$$

por lo que podemos calcular la dirección como

$$
\begin{aligned}
address(\mathbf{foo}[\mathbf{6}, -\mathbf{2}, \mathbf{3}]) &= address(\mathbf{foo}) + (i - L_1) \times S_1 + (j - L_2) \times S_2 \\
&= 1000 + (6 - 0) \times 8 \times 9 \times 48 + (-2 - (-7)) \times 9 \times 48 + (3 - (-4)) \times 48 \\
&= 1000 + 9 \times 48 \times 48 + 5 \times 9 \times 48 + 7 \times 48 \\
&= 1000 + (9 \times 48 + 45 + 7) \times 48 \\
&= 1000 + (9 \times 48 + 48 + 4) \times 48 \\
&= 1000 + 480 \times 48 + 4 * 48 \\
&= 1000 + 484 \times 48 \\
&= 24232
\end{aligned}
$$

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.