Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2009

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 **tres (3) 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 ocupa 4 bytes y cada apuntador ocupa 8 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 `bar:=foo`?

   La respuesta correcta es la **tercera** alternativa, i.e. si $*\alpha \neq nil \wedge **\alpha \neq *(\alpha + 8)$, error de referencia colgada; en caso contrario, $\beta := *\alpha$ y $\beta + 8 := *(\alpha + 8)$. Ese resultado se deduce de la Figura (pendiente).

   El condicional comienza por determinar si `foo` está apuntando a algo ($*\alpha \neq nil$). Si eso es cierto (nótese la conjunción) y si la cerradura de lo apuntado **no** coincide con la llave que posee `foo` ($**\alpha \neq *(\alpha + 8)$) entonces hay un error de referencia colgante. Ahora, si `foo` no apunta a nada ($*\alpha = nil$ y se hace falta la condición) o apunta a algo y tenemos la llave ($**\alpha = *(\alpha + 8)$ y se hace falsa la condición), entonces simplemente copiamos el apuntador de `foo` en `bar` ($\beta := *\alpha$) y copiamos la llave de `foo` en `bar` ($(\beta + 8) := *(\alpha + 8)$).

(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* con *copia profunda*.

La respuesta correcta es la **primera** 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 estucturas **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 $*a + 4 + k$.

(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

```
bar:=foo
```

¿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 **cuarta** alternativa, i.e. decrementar $*(*\beta + 4)$, liberar memoria del *heap* si hace falta, incrementar $*(*\alpha + 4)$ y 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 el contador de referencias a partir del cuarto byte. Como `bar` dejará de apuntar a lo que sea que estaba apuntando, es necesario decrementar el contador de referencias de ese objeto en *heap* $(*(*\beta + 4))$ y si se realiza una recuperación de memoria inmediatamente el recolector de basura podría recuperarla si el contador llegó a cero. Como `bar` va a apuntar a lo mismo que está apuntando `foo`, es necesario incrementar el contador dentro de la estructura apuntada por `foo` $(*(*\alpha + 4))$. La última operación de recuperación de memoria es inocua en cuanto a ésta asignación.

(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 las variables enteras `i` y `j` están almacenadas en las direcciones α y β y la base de `m` ha sido almacenada en la dirección de memoria γ. 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 el valor de `m[i][j]`?

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

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

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

   (a) **(4 puntos)** Utilice listas por comprensión en Haskell para escribir la función

   ```
   partesDe :: [a] -> [[a]]
   ```

   que reciba una lista de elementos cualesquiera sin repeticiones y produzca la lista de todas las listas que pueden producirse a partir de la lista suministrada (sin repetir ninguna), en el sentido de "partes de un conjunto". Por ejemplo

   ```
   ghci> partesDe [1,2,3]
   [[],[1],[2],[3],[1,2],[1,3],[2,3],[1,2,3]]
   ```

   Esta función corresponde al iterador presentado como problema en el primer parcial, de modo que aplica el mismo algoritmo, aplicando la estrategia recursiva de algoritmos sobre listas: caso base sobre lista vacía, caso inductivo usando la cabeza de la lista y recurriendo a la cola. Las "partes" de una lista vacía son una lista que solamente contiene la lista vacía. Para las partes de una lista con al menos un elemento, fijo dicho elemento y luego genero una lista que tenga todas las partes del resto de la lista (recursión) concatenada con otra lista que tiene al elemento fijo al frente de todas las partes del resto de la lista (recursión).

   ```
   partesDe :: [a] -> [[a]]
   partesDe []     = [[]]
   partesDe (x:xs) = partesDe xs ++ [ x:zs | zs <- partesDe xs ]
   ```

   esta solución es intencionalmente ineficiente (trabaja dos veces con la cola) para hacerla más clara. Una pequeña mejora de eficiencia podría ser

   ```
   partesDe :: [a] -> [[a]]
   partesDe []     = [[]]
   partesDe (x:xs) = let ps = partesDe xs
                     in ps ++ [ (x:zs) | zs <- ps ]
   ```

   y como la última expresión se puede convertir en un `map` podríamos tener

   ```
   partesDe :: [a] -> [[a]]
   partesDe []     = [[]]
   partesDe (x:xs) = let ps = partesDe xs
                     in ps ++ map (x:) ps
   ```

(b) **(6 puntos)** Eric Rowland propuso un método que genera números primos. No los genera todos y genera repetidos, pero está basado en una recurrencia muy simple

$$\begin{aligned} p(1) &= 7 \\ p(n) &= p(n-1) + gcd(n, p(n-1)) \end{aligned}$$

donde *gcd* representa el máximo común divisor (función que existe en Haskell con el nombre `gcd`). Si se genera la secuencia de valores $p(2) - p(1), p(3) - p(2), \ldots, p(n) - p(n-1)$ siempre se obtiene como resultado 1 o un número primo. Escriba la función Haskell

```
rowland :: [Integer]
```

que aproveche esa recurrencia para generar los números primos **sin** incluir el 1 y **sin** repeticiones. Puede usar cualquier función del preludio Haskell. **Pista:** la función

```
zipWith :: (a -> a -> b) -> [a] -> [a] -> [b]
```

toma los elementos correspondientes de las listas `xs` y `ys`, les aplica la función binaria `f` y produce una lista con los resultados. Si Ud. sabe el nombre de la función de tres letras provista por Haskell para eliminar repeticiones en una lista, puede usarla, caso contrario deberá definirla.

Hay una solución obvia, escribiendo la función `p` tal cual está en el enunciado. Es obvia pero es horrorosamente ineficiente pues la función `p` es profundamente recursiva, así que el rendimiento de la función `rowland` final sería malo, tardando mucho en ofrecer resultados. Los estudiantes que utilizaron dicha solución obvia han recibido crédito completo por la respuesta, sin embargo es claro que no han aprovechado los conocimiento acerca de transformación de la recursión de la primera parte.

La recurrencia tiene un caso base y debo inducirla hasta el infinito. Además, el n-ésimo caso depende del $(n-1)$-ésimo... similar a lo que ocurre con Fibonacci así que utilizo el mismo truco de *tupling* para calcular. Esto es, la función que genera la secuencia tiene un valor base predefinido (en este caso la tupla $(1, 7)$ indicando $p(1) = 7$) y como es una lista infinita, ese valor está al principio y el resto se calcula agregando tuplas con exactamente la expresión de la recurrencia

```
as = (1,7) : [ (n+1, x + (gcd (n+1) x) | (n,x) <- as ]
```

ahora es necesario calcular la diferencia entre el elemento $n+1$-ésimo y el elemento n-ésimo. Como la lista es infinita es lo mismo que restar cada elemento de la lista de cada elemento de la **cola** de la lista, pero solamente me interesan los valores calculados (el segundo elemento de cada tupla), entonces

```
l = zipWith (-) a2 a1
    where as = (1,7) : [ (n+1, x + (gcd (n+1) x) | (n,x) <- as ]
          a1 = [ n | (_,n) <- as ]
          a2 = tail a1
```

y ahora solamente resta limpiar la lista `l` eliminado todos los 1 y cualquier número duplicado. La función nub de Haskell elimina los duplicados de cualquier lista, por tanto, la solución completa es

```
rowland = nub $ [ n | n <- zipwith (-) a2 a1, n /= 1 ]
          where as = (1,7) : [ ( n+1, x + (gcd (n+1) x) ) | (n,x) <- as
                a1 = [ n | (_,n) <- as ]
                a2 = tail a1
```

Al correrla, se obtienen los resultados prometidos... números primos.

```
> take 10 rowland
[5,3,11,23,47,101,7,13,233,467]
```

3. Considere la siguiente declaración de algún lenguaje de programación

```
foo : array [4..10] of array [-11..-2] of array [-7..7] of
      record
        a : char[3]
        b : char
        c : short
        d : integer
        e : float
        f : *short
        union
          record
            g : float
            h : boolean
            i : char[3]
          end record
          record
            j : integer
            k : *integer
          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
apuntador	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`	$0,+8,+16,+24,\ldots,+8n$
`apuntador`	$0,+8,+16,+24,\ldots,+8n$

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 desplazamientos (*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	a	a	#
+4	b	#	c	c
+8	d	d	d	d
+12	#	#	#	#
+16	e	e	e	e
+20	e	e	e	e
+24	f	f	f	f
+28	f	f	f	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 `float` que debe estar alineado en un múltiplo de 8, y la Variante B comienza por el campo `j` de tipo `integer` que debe estar alineado en un múltiplo de 4; por tanto **ambas** partes variantes comienzan alineadas desde la posición 32 y se representan como muestra la siguiente tabla

Offset	Variante A					Variante B			
+32	g	g	g	g		j	j	j	j
+36	g	g	g	g		#	#	#	#
+40	h	#	i	i		k	k	k	k
+44	i	#	#	#		k	k	k	k

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 10 bytes de 48 para 20.83%.
- Variante B: se desperdician 10 bytes de 48 para 20.83%.

(c) **(2 puntos)** ¿Cuánto espacio ocupa todo el arreglo `foo` en bytes?
El arreglo tiene $U_0 - L_0 + 1 = 10 - 4 + 1 = 7$ filas, cada una de $U_1 - L_1 + 1 = -2 - (-11) + 1 = 10$ columnas, conteniendo $U_2 - L_2 + 1 = 7 - (-7) + 1 = 15$ posiciones. Cada posición debe contener un elemento de 48 bytes, por lo tanto

$$\begin{aligned} espacio(foo) &= filas \times columnas \times posiciones \times 48\, bytes \\ &= 7 \times 10 \times 15 \times 48\, bytes \\ &= 50400\, bytes \end{aligned}$$

(d) **(5 puntos)** Suponga que la declaración de `foo` es para una variable global y el compilador le asignó la dirección base 2202. ¿Cuál es la dirección del elemento `foo[10,-5,-1]`?
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

$$\begin{aligned} S_3 &= 48\, bytes \\ L_3 &= -7 \\ U_3 &= 7 \\ S_2 &= (U_3 - L_3 + 1) \times S_3 = (7 - (-7) + 1) \times 48 = 15 \times 48 \\ L_2 &= -11 \\ U_2 &= -2 \end{aligned}$$

$$
\begin{aligned}
S_1 &= (U_2 - L_2 + 1) \times S_2 = (-2 - (-11) + 1) \times 15 \times 48 = 10 \times 15 \times 48 \\
L_1 &= 4 \\
U_1 &= 10
\end{aligned}
$$

por lo que podemos calcular la dirección como

$$
\begin{aligned}
address(\mathbf{foo}[\mathbf{10}, -\mathbf{5} - \mathbf{1}]) &= address(\mathbf{foo}) + (i - L_1) \times S_1 + (j - L_2) \times S_2 + (k - L_3) \times S_3 \\
&= 2202 + (10 - 4) \times 10 \times 15 \times 48 + (-5 - (-11)) \times 15 \times 48 + (-1 - (-7)) \times 48 \\
&= 2202 + 6 \times 10 \times 15 \times 48 + 6 \times 15 \times 48 + 6 \times 48 \\
&= 2202 + 6 \times 48 \times (10 \times 15 + 15 + 1) \\
&= 2202 + 288 \times 165 + 288 \\
&= 50010
\end{aligned}
$$

4. **(Extra por 4 puntos)** Construya sendos diagramas que describan las estructuras de datos definidas por las siguientes declaraciones en lenguaje C. **Nota:** esta pregunta solamente será tomada en cuenta si Ud. aprueba el resto del examen.

   (a) `int *foo[n]`

   En este caso `foo` es un arreglo de `n` posiciones reservadas en espacio continuo, cada una de las cuales contiene un apuntador a un entero, cuyo espacio aún no está reservado.

   (b) `int (*bar)[n]`

   En este caso `bar` es un apuntador a un arreglo de `n` posiciones, cada una de las cuales contiene un entero. El espacio para el arreglo tendría que ser global o reservarse en el *heap*.

   (c) `int (*baz[n])()`

   En este caso `baz` es un arreglo de `n` posiciones reservadas en espacio continuo, cada una de las cuales contiene un apuntador a una función que retorna un entero.

   (d) `int (*qux())[n]`

   En este caso `qux` es una función que retorna un apuntador a un arreglo de `n` posiciones, cada una de las cuales contiene un entero. El espacio para el arreglo tendría que ser global o reservarse en el *heap*.