Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3641 - Lenguajes de Programación I
Enero-Marzo 2007

Examen II

(30 puntos)

Nombre:

Carnet:

1. **(8 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**.

   (a) 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 **tercera** alternativa, i.e.

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

   resultado que se deduce de la Figura 1

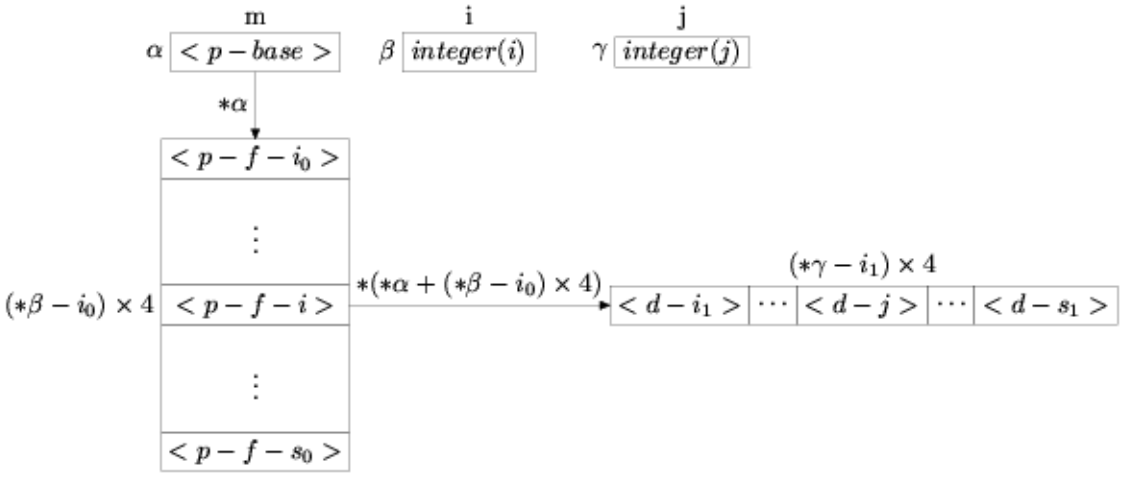

Figure 1: Disposición en Memoria

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.

(b) 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 la llave de acceso y el apuntador propiamente dicho, 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 **segunda** alternativa, i.e. si $*(\beta+4) \neq nil \wedge *\beta \neq **(\beta+4)$, error de referencia colgada; en caso contrario, $\alpha := *\beta$ y $\alpha+4 := *(\beta+4)$. Ese resultado se deduce de la Figura 2

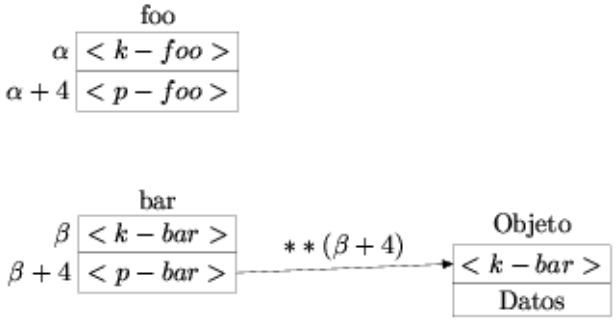

Figure 2: Disposición en Memoria

El condicional comienza por determinar si `bar` está apuntando a algo ($*(\beta+4) \neq nil$). 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 la llave de `bar` en `foo` ($\alpha := *\beta$) y copiamos el apuntador de `bar` a `foo` ($\alpha+4 := *(\beta+4)$).

(c) En lenguaje C, la declaración de un parámetro formal de tipo arreglo puede omitir el tamaño del arreglo (e.g. `int foo[]`) pues éste será determinado por el parámetro real. Sin embargo el tipo base es imprescindible para poder determinar a tiempo de compilación como manejar la aritmética de apuntadores y de indexación. De acuerdo con esto, en el caso de un arreglo bi-dimensional de memoria contigua almacenado por filas, ¿qué se puede omitir en la declaración de un parámetro formal?

i. **Sólo puede omitirse el tamaño de la primera dimensión, e.g.** `T foo[][N]`.

ii. Sólo puede omitirse el tamaño de la segunda dimensión, e.g. `T foo[N][]`.
No, pues sería imposible calcular estáticamente los tamaños de cada fila, necesarios para utilizar *row-major ordering* como es el caso de C.

iii. Pueden omitirse los tamaños de ambas dimensiones, e.g. `T foo[][]`.
No, mismo razonamiento anterior.

iv. Ninguno de los dos puede omitirse, e.g. `T foo[N][M]`.
No, precisamente porque queremos saber **qué** se puede omitir gracias a la estrecha relación apuntador-arreglo de C.

(d) Un recolector de basura por marcado y barrido en un lenguaje con tipos estrictos:

i. Es incapaz de recuperar la memoria de objetos inalcanzables desde el ambiente de referencia.
No, pues precisamente la motivación de un recolector de basura por marcado y barrido es encontrar objetos inalcanzables.

ii. Es capaz de encontrar y recuperar todos los objetos inútiles.
No, pues es posible que un programa reserve un objeto que es semánticamente inútil, pero como es alcanzable no sea liberado.

iii. Es más económico en espacio que un recolector por conteo de referencias.
No, pues requiere apuntadores extra en la implantación inocente, y cualquiera de las implantaciones avanzadas requieren particionar y administrar el *heap* con meta-información adicional.

iv. **Si es implantado de manera inocente, siempre afectará la ejecución del programa de manera notable, pero a intervalos más o menos regulares.**

2. **(5 puntos)** Sea la siguiente función Haskell que produce una lista con las diferentes denominaciones de monedas

```
monedas = [ 1000, 500, 100, 50, 20, 10 ]
```

Se desea que Ud. escriba **una** función en Haskell que calcule **todas** las formas posibles de constituir un monto utilizando monedas y las retorne como una lista de listas. La función **debe** escribirse utilizando **solamente** listas por compresión (*lists comprehensions*) y tener firma

```
enMonedas :: Int -> [ [Int] ]
```

Las combinaciones generadas por la función deben comenzar por aquella que tenga **menos monedas** y debe terminar en aquella que tenga **más monedas**. Por ejemplo,

```
> enMonedas 40
[[20,20],[20,10,10],[10,20,10],[10,10,20],[10,10,10,10]]
```

Note que la segunda, tercera y cuarta combinaciones son idénticas salvo posición de las monedas, y no es necesario que la función tome en cuenta esas cosas.

```
enMonedas 0     = [[]]
enMonedas monto = [ m:ms | m <- monedas, monto >= m, ms <- enMonedas (monto - m) ]
```

3. Considere la siguiente definición de variables utilizando el tipo conjunto de Pascal (note las dos incógnitas)

```
var foo : set of 'd'..'p';
    bar : set of 'm'..'v';
    baz : set of ?..?;
    i   : 'a'..'z';
```

ahora considere la siguiente expresión, en la cual + denota la **unión** de conjuntos, $*$ denota la **intersección** de conjuntos y el uso de corchetes representa la construcción de conjuntos por enumeración de sus elementos.

```
baz := foo + bar * ['d'..'h', i]
```

Para efecutar las operaciones, el lenguaje considera que todos los operandos son del tipo `set of character`, sin embargo es necesario realizar una inferencia detallada sobre los tipos de datos para determinar si el lado derecho de la asignación es compatibles con la variable del lado izquierdo de la misma.

(a) **(2 puntos)** Calcule el subtipo **mínimo** de `set of character` que puede ser inferido **estáticamente** para cada una de las subexpresiones del lado derecho de la asignación, incluyendo por supuesto la expresión total.

Expresión	Tipo Inferido	Justificación
$['d'..'h', i]$	`set of` $'a'..'z'$	El tipo de i contiene a $'d'..'h'$.
`bar`	`set of` $'m'..'v'$	Por definición de la variable `bar`.
`bar` $*['d'..'h', i]$	`set of` $'m'..'v'$	Por definición de intersección de conjuntos.
`foo`	`set of` $'d'..'p'$	Por definición de la variable `foo`.
`foo+bar`$*['d'..'h', i]$	`set of` $'d'..'v'$	Por definición de unión de conjuntos.

(b) **(5 puntos)** Determine para cuáles subtipos de `set of character` de la variable `baz`, según los posibles valores de las incógnitas, el compilador debería generar un error estático, generar código que verifique la posibilidad de error dinámicamente o generar código que realice la asignación incondicionalmente.

- **Error Estático:** cuando a tiempo de compilación se determina que el tipo de la expresión del *r-value* **nunca** va a ser de tipo compatible con el *l-value*. Esto ocurre si:
  - `baz` se define como `set of` $'a'..'c'$.
  - `baz` se define como `set of` $'w'..'z'$.
- **Generar código que verifique la posibilidad de error dinámicamente:** cuando a tiempo de compilación se determina que el tipo de la expresión del *r-value* **a veces** tendrá tipo compatible con el *l-value*. El código de verificación debe establecer si el valor particular del *r-value* está en el rango que lo hace compatible, y en ese caso hace la asignación; caso contrario genera un error semántico a tiempo de ejecución. Eso es necesario si:
  - `baz` se define como `set of` $'a'..'z'$.
  - `baz` se define como `set of` $'a'..'v'$.
  - `baz` se define como `set of` $'d'..'z'$.
- **Generar código que realice la asignación incodicionalmente:** cuando a tiempo de compilación se determina que el tipo de la expresión del *r-value* **siempre** tendrá tipo compatible con el *l-value*. Eso ocurre solamente si `baz` se define como `set of` $'d'..'v'$.

4. Considere la siguiente declaración de algún lenguaje de programación

```
foo : array [2..14] of array [-4..10] of
      record
        a : float
        b : short
        c : char
        d : char
        union
          record
            e : double
            f : short
            g : char
          end record
          record
            h : short
            i : double
            j : integer
          end record
        end union
      end record
```

En el lenguaje los tipos de datos se definen

Tipo de Datos	Representación
`char`	8 bits
`short`	16 bits
`integer`	32 bits
`float`	32 bits
`double`	64 bits

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).

(a) **(2 puntos)** ¿Cuánto espacio ocupa un elemento del arreglo? ¿Cuál es el porcentaje de espacio desperdiciado?

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 alinerse según describe la siguiente tabla

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

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`	`a`	`a`	`a`
+4	`b`	`b`	`c`	#
+8	`d`	#	#	#

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 `e` de tipo `double` que debe estar alineado en un múltiplo de 8, mientras que la Variante B comienza por el campo `h` de tipo `short` que debe estar alineado en un múltiplo de 2; el mínimo común múltiplo de ambos que sea mayor que +12 (offset donde termina la parte común) es +16, por tanto **ambas** partes variantes comienzan alineadas desde allí y se representan como muestra la siguiente tabla

Offset	**Variante A**					**Variante B**			
+12	#	#	#	#		#	#	#	#
+16	e	e	e	e		h	h	#	#
+20	e	e	e	e		#	#	#	#
+24	f	f	g	#		i	i	i	i
+28	#	#	#	#		i	i	i	i
+32	#	#	#	#		j	j	j	j

En consecuencia, cada elemento del arreglo ocupa tanto como la Parte Común y la Variante B, para un total de **treinta y seis (36) bytes**. El espacio desperdiciado tiene dos casos

- Variante A: se desperdician 17 bytes de 36 para 47.22%.
- Variante B: se desperdician 14 bytes de 36 para 38.89%.

Como es imposible determinar a priori, cuál será la distribución de datos que tendrá el arreglo, se utiliza el **peor caso**, i.e. el desperdicio mayor correspondiente a la Variante A.

(b) **(1 punto)** Muestre los desplazamientos de cada uno de los campos.

- **Parte Común**
  - Campo `a`, desplazamiento 0.
  - Campo `b`, desplazamiento +4.
  - Campo `c`, desplazamiento +6.
  - Campo `d`, desplazamiento +8.
- **Variante A**
  - Campo `e`, desplazamiento +16.
  - Campo `f`, desplazamiento +24.
  - Campo `g`, desplazamiento +26.
- **Variante B**
  - Campo `h`, desplazamiento +16.
  - Campo `i`, desplazamiento +24.
  - Campo `j`, desplazamiento +32.

(c) **(2 puntos)** ¿Cuánto espacio ocupa todo el arreglo `foo`?
El arreglo tiene $U_0 - L_0 + 1 = 14 - 2 + 1 = 13$ filas, cada una de $U_1 - L_1 + 1 = 10 - (-4) + 1 = 15$ columnas. Cada posición debe contener un elemento de 36 bytes, por lo tanto

$$\begin{aligned} espacio(foo) &= filas \times columnas \times 36\, bytes \\ &= 13 \times 15 \times 36\, bytes \\ &= 7020\, bytes \end{aligned}$$

(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[7,-2]`?

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

$$\begin{aligned} L_2 &= -4 \\ U_2 &= 10 \\ S_2 &= sizeof(Variante\, B) = 36\, bytes \\ L_1 &= 2 \\ U_1 &= 14 \\ S_1 &= (U_2 - L_2 + 1) \times S_2 = (10 - (-4) + 1) \times 36 = 540 \end{aligned}$$

por lo que podemos calcular la dirección como

$$\begin{aligned} address(\mathbf{foo}[\mathbf{7}, \mathbf{-2}]) &= address(\mathbf{foo}) + (i - L_1) \times S_1 + (j - L_2) \times S_2 \\ &= 1000 + (7 - 2) \times 540 + (-2 - (-4)) \times 36 \\ &= 1000 + 2700 + 72 \\ &= 3772 \end{aligned}$$

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.