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

Carnet:_____

Nombre:_____

Examen II

(35 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	**Total**
21 puntos	4 puntos	10 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 o, si desea no contestar la subpregunta, marcar completamente y sin posibilidad de confusión la última opción (e) que indica que Ud. prefiere omitir la respuesta.
Cada una de las subpreguntas tiene un valor de tres (3) puntos. Tres subpreguntas incorrectas eliminan una correcta. Las subpreguntas omitidas (marcadas en la opción e) no suman ni restan puntos.

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 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, 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 `bar := foo`?

a) Si $*(\beta + 4) \neq null \wedge *\beta \neq **(\beta + 4)$, error de referencia colgada;
en caso contrario $\alpha := *\beta$ y $\alpha + 4 := *(\beta + 4)$

b) Si $*(\beta + 4) \neq null \wedge \beta \neq *(\beta + 4)$, error de referencia colgada;
en caso contrario $*\alpha := \beta$ y $*(\alpha + 4) := \beta + 4$

c) Si $*(\alpha + 4) \neq null \wedge *\alpha \neq **(\alpha + 4)$, error de referencia colgada;
en caso contrario $\beta := *\alpha$ y $\beta + 4 := *(\alpha + 4)$

d) Si $*(\alpha + 4) \neq null \wedge \alpha \neq *(\alpha + 4)$, error de referencia colgada;
en caso contrario $*\beta := \alpha$ y $*(\beta + 4) := \alpha + 4$

e) No sabe / No contesta.

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

```
bar := foo
```

¿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 $*(*(\beta + 4) + 12)$;
incrementar $*(*(\alpha + 4) + 12)$.

b) Decrementar $*(*(\beta + 4) + 12)$;
si $*(*(\beta + 4) + 12)) = 0$, liberar el objeto inmediatamente;
incrementar $*(*(\alpha + 4) + 12)$.

c) Decrementar $*(*(\alpha + 4) + 12)$;
si $*(*(\alpha + 4) + 12)) = 0$, liberar el objeto inmediatamente;
incrementar $*(*(\beta + 4) + 12)$.

d) Ninguna de las anteriores.
Nota: la presencia de un descriptor de tipos sugiere que el objeto podría ser de un tipo recursivo, así que es necesario determinar si su contador llegó a cero **y** decrementar recursivamente los contadores de cualquier objeto apuntado.

e) No sabe / No contesta.

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

```
barˆ := fooˆ
```

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) $*(*(\beta + 4) + 16 + k) := *(*(\alpha + 4) + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

b) $*(\beta + 4) + 16 + k := *(*(\alpha + 4) + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

c) $*(*(\alpha + 4) + 16 + k) := *(*(\beta + 4) + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

d) $*(\alpha + 4) + 16 + k := *(*(\beta + 4) + 16 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

e) No sabe / No contesta.

1.4. Sea la siguiente línea en un programa de un lenguaje de programación imperativo arbitrario

```
foo = (bar = 21) + (bar++)
```

Considere las siguientes afirmaciones:

I. No se puede predecir el resultado porque en todo lenguaje los compiladores tienen la libertad de reordenar el orden de las evaluaciones por razones de eficiencia.

II. Si el lenguaje opera como Java, el estado mutable tendrá `bar = 22` y `foo = 42`.

III. El lenguaje no separa instrucciones de expresiones.

IV. No se puede predecir el resultado sin conocer la asociatividad y precedencia del operador de asignación (`=`) en relación al resto.

¿Cuáles afirmaciones son ciertas?

a) I y III solamente.

b) III solamente.

c) II y III solamente.

Nota: La tres es obvia porque está en todas las alternativas y porque la asignación es una instrucción que está mezclada dentro de una expresión. Si el lenguaje opera como Java, entonces evalúa las expresiones *estrictamente* de izquierda a derecha, y los cómputos producen los resultados indicados.

d) III y IV solamente.

e) No sabe / No contesta.

1.5. Considere el siguiente fragmento de un programa en Pascal

```
var a : set of 7..21;
    b : set of 14..21;
    c : set of ?..?;
    i : 1..42;

c := a + b * [1..7, i]
```

Entonces es cierto que

a) Si se declara `c : set of 7..21`, entonces el compilador debe generar código para verificar el resultado de la expresión en el lado derecho antes de hacer la asignación.

b) Si se declara `c : set of 7..21`, entonces el compilador genera un error estático porque la asignación sería imposible.

c) Si se declara `c : set of 0..14`, entonces el compilador genera código que hace la asignación directamente sin necesidad de verificación.

d) Si se declara `c : set of 0..14`, entonces el compilador genera código que verifica el valor resultado de la expresión en el lado derecho antes de hacer la asignación.

e) No sabe / No contesta.

1.6. Considere la siguiente declaración de un arreglo

```
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) $*(*(\delta - 8) + (*\alpha - *\delta) \times 4) + (*\beta - 1) \times sizeof(T)$
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) $*(*(\delta - 8) + *\alpha \times (*(\delta - 4) - *\delta + 1) + (*\beta - 1) \times sizeof(T))$
e) No sabe / No contesta.

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;
    }
  }
}
```

¿Cuál es el *tercer* elemento generado al invocar `grok([4,2,1])`?

a) `[4,2]`
b) `[2,1]`
c) `[2]`
d) `[4,1]`
e) No sabe / No contesta.

Pregunta 2 - 4 puntos

A continuación se presenta una función recursiva en pseudocódigo, que se apoya en el tipo recursivo `Lista` para modelar listas enlazadas:

```
type List = ^record
              elem : T;
              next : List
            end;

fun duplicate(List orig) -> List
begin
  if (orig == null) then
    return orig
  else
    var copy : List;
    new(copy);
    copy^.elem = orig^.elem;
    copy^.next = duplicate(orig^.next);
    return copy
  fi
end
```

Reescriba la función `duplicate` de tal manera que opere con recursión de cola. Si lo considera necesario, puede hacer uso de funciones auxiliares, pero toda función que utilice debe ser **completamente** definida por Ud.

```
fun duplicate(List orig) -> List
begin
  List primero;
  go(orig, primero);
  return primero^.next;
go

fun go(List orig, List final) -> void
begin
  if (orig == null)
    final^.next = null
  else
    new(final^.next);
    final^.next^.elem = orig^.elem;
    go(orig^.next,final^.next);
  fi
}
```

- No se puede cambiar la firma de la función original – utiliza una función auxiliar recursiva de cola.
- Técnica de parámetro auxiliar, pero no lo uso como acumulador sino para tener acceso a la última posición de la lista.
- Ejemplo que sale en la página de Wikipedia sobre Tail Call.
- Soluciones iterativas inaceptables aunque funcionen – quiero evaluar si saben inducir recursión de cola.
- Mi solución modifica "en sitio" gracias a que se trata de apuntadores. Es posible resolverlo usando tres argumentos para `go` uno de los cuales simplemente conserva la posición del primer elemento para retornarlo al final – no me gusta pasar parámetros que no hacen nada.

Pregunta 3 - 10 puntos

Considere el siguiente tipo abstracto y recursivo en Haskell, empleado para representar el rudimentario sistema de tipos anidables de un lenguaje cuyo compilador se quiere implantar:

```
data Tipo = Booleano
          | Entero
          | Flotante
          | Arreglo Int Int Tipo
          | Registro [ (String,Tipo) ]
          | Variante [ (String,Tipo) ]
          deriving (Show,Eq)
```

Como es natural en la implantación de un compilador, es necesario prestar atención a las condiciones impuestas por el diseñador y a las prestaciones de la plataforma. En este sentido:

- Para los tipos primitivos, se ha decidido que un `Booleano` ocupará 1 byte, un `Entero` ocupará 4 bytes y un `Flotante` ocupará 8 bytes.
- Para cada `Arreglo`, se utilizará disposición *row-major* sobre posiciones de memoria contiguas. Note que la definición del tipo de datos permite tener arreglos multidimensionales.
- Para cada `Registro` se utilizará la disposición clásica de modelo de valor, i.e. registros que contienen registros, variantes o arreglos, simplemente incluyen los valores anidados.
- Para cada `Variante` se utilizará la disposición clásica de modelo de valor, i.e. ocupará tanto espacio como la alternativa que ocupe más espacio, tomando en cuenta la posibilidad de tener `Registro`s, `Variante`s y `Arreglo`s incluidos igual que en el caso anterior.
- La plataforma prefiere que todos los componentes de datos estén alineados en direcciones múltiplo de cuatro (4) bytes.

```
boundary = 4
```

Se desea que usted escriba dos funciones Haskell:

```
sizeof :: Tipo -> Int
align  :: [(String, Tipo)] -> [(String, Int)]
```

La función `sizeof` debe calcular el tamaño de un valor para un tipo de datos particular arbitrariamente anidado y, por supuesto, luego de haberlo alineado si fuera ser necesario, e.g.

```
ghci> sizeof Entero
4
ghci> sizeof $ Arreglo 1 5 Entero
20
ghci> sizeof $ Arreglo 1 5 $ Variante [("foo",Entero),("bar",Flotante)]
40
```

La función `align` debe procesar una lista de nombres de campos y tipos asociados, como los que conforman la definición de los tipos `Registro`, y produce el desplazamiento que debe tener cada uno de los campos, sujetos a las reglas de diseño y plataforma discutidas anteriormente, e.g.

```
> align  [("foo",Booleano),
          ("bar",Entero),
          ("baz",Arreglo 0 5 Flotante),
          ("meh",Booleano)]
[("foo",0),("bar",4),("baz",8),("meh",56)]
```

Su implementación de la función `align` **debe** estar escrita utilizando recursión de cola. Solamente puede utilizar funciones provistas en el *Haskell Prelude*. Recibirá dos (2) puntos adicionales si es capaz de expresarla apoyándose en una función de orden superior como las que ha estudiado en el laboratorio.

```
-- Si no es múltiplo de 'boundary' le agrego lo que falta
pad :: Int -> Int
pad offset = let end = offset `mod` boundary
             in offset + if end == 0 then 0
                                     else boundary - end

-- * Los tamaños de los tipos primitivos son directos.
-- * El tamaño de un arreglo es una fórmula trivial.
-- * El tamaño de un variante es el máximo de los tamaños.
-- * El tamaño de un registro requiere alinearlo primero.
sizeof :: Tipo -> Int
sizeof Booleano        = 1                              -- 0.25
sizeof Entero          = 4                              -- 0.25
sizeof Flotante        = 8                              -- 0.25
sizeof (Arreglo i s t) = (s - i + 1) * sizeof t         -- 0.25
sizeof (Variante fs)   = maximum $ map (sizeof.snd) fs  -- 1.00
sizeof (Registro fs)   = pad $ o + sizeof t             -- 2.00
                         where (_,t) = last fs
                               (_,o) = last $ align fs

-- go utiliza recursión de cola directa
-- * Acumulo los campos y offsets asignados, en orden inverso.
-- * Comienzo en offset 0 y en cada llamada se alinea.
-- reverse del Haskell Prelude usa recursión de cola via foldl
align :: [ (String,Tipo) ] -> [ (String,Int) ]
align fields = reverse $ go [] 0 fields
  where
    go a o []        = a
    go a o ((n,t):r) = go ((n,o):a) (pad (o + sizeof t)) r

-- Solución alternativa usando foldl y "tupling" (contador,acumulador)
-- * En la práctica se usaría Data.List.foldl'
align :: [ (String,Tipo) ] -> [ (String,Int) ]
align fields = reverse $ snd $ foldl go (0,[]) fields
  where
    go (o,fs) (i,t) = (pad (o + sizeof t), (i,o):fs)
```