Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Enero-Marzo 2016

Carnet:_____

Nombre:_____

Examen II

(35 puntos)

Pregunta 1	Pregunta 2	Pregunta 3	Pregunta 4	**Total**
21 puntos	4 puntos	5 puntos	5 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. Cada una de las subpreguntas tiene un valor de tres (3) puntos. Tres subpreguntas incorrectas eliminan una correcta.

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 4 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 `foo := bar`?

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

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

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

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

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, pero antes del valor del objeto en sí.
En la asignación

```
foo := bar
```

¿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) Incrementar $*(\beta + 4) + 8$.
Decrementar $*(\alpha + 4) + 8$;
si $*(\alpha + 4) + 8 = 0$,
repetir para toda dirección mencionada en $*(\alpha + 4) + 4$ antes de liberar el objeto α.

b) Incrementar $*(*(\beta + 4) + 8)$.
Decrementar $*(*(\alpha + 4) + 8)$;
si $*(*(\alpha + 4) + 8) = 0$,
repetir para toda dirección mencionada en $*(*(\alpha + 4) + 4)$ antes de liberar el objeto α.

c) Incrementar $*(\alpha + 4) + 8$.
Decrementar $*(\beta + 4) + 8$;
si $*(\beta + 4) + 8 = 0$,
repetir para toda dirección mencionada en $*(\beta + 4) + 4$ antes de liberar el objeto β.

d) Incrementar $*(*(\alpha + 4) + 8)$.
Decrementar $*(*(\beta + 4) + 8)$;
si $*(*(\beta + 4) + 8) = 0$,
repetir para toda dirección mencionada en $*(*(\beta + 4) + 4)$ antes de liberar el objeto β.

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

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

entendiendo que `^` 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) $*(*(\beta + 4) + 12 + k) := *(*(\alpha + 4) + 12 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

b) $*(\beta + 4) + 12 + k := *(*(\alpha + 4) + 12 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

c) $*(\alpha + 4) + 12 + k := *(*(\beta + 4) + 12 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

d) $*(*(\alpha + 4) + 12 + k) := *(*(\beta + 4) + 12 + k)$, con $k = 0, 4, 8, \ldots$ hasta alcanzar $sizeof(T)$.

1.4. Un fanático proselitista del nuevo lenguaje imperativo PonycornScript está hablando del maravilloso recolector de basura del lenguaje. Nunca dió detalles técnicos concretos, (así son los usuarios de ese lenguaje), sin embargo dijo algo que *seguro* es mentira. ¿Qué dijo?

a) Es capaz de recuperar la memoria asociada a estructuras con referencias circulares, incluso si no son accesibles desde el conjunto raíz del ambiente de referencia.

b) Nunca pierde memoria porque no usa métodos aproximados para la identificación de apuntadores.

c) La implantación aprovecha el hecho que una página más vieja nunca apunta a una página más nueva.

d) Al iniciar el programa, debes indicar cuánto *heap* utilizar, y se empleará exactamente esa cantidad de memoria.

1.5. Considere la siguiente declaración de un arreglo en un lenguaje con modelo de valor

```
foo : array [low .. high] of array [-4..4] 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 enteros como los apuntadores ocupan 4 bytes de memoria cada uno, entonces la fórmula adecuada sería:

a) $*(*(\delta - 8) + (*\alpha - *\delta) \times 4) + (*\beta + 4) \times sizeof(T)$

b) $*(*(\delta + 8) + (*\alpha - *\delta) \times 4) + (*\beta - 4) \times sizeof(T)$

c) $*(*(*(\delta - 8) + (*\alpha - *\delta) \times 4) + (*\beta + 4) \times sizeof(T))$

d) $*(*(\delta + 8) + (*\alpha - *\delta) \times 4) + (*\beta + 4) \times sizeof(T)$

1.6. Considere las siguientes definiciones de tipos en un lenguaje con soporte nativo para conjuntos, en el cual + corresponde a la unión, ∗ corresponde a la intersección, y − corresponde a la diferencia.

```
var a : set of 0..15;
    b : set of 10..25;
    c : set of u..v;

c := b + c * (a - b)
```

¿Cuáles límites `u` y `v` obligarían al compilador a generar código adicional para verificar que no hay un error de tipos?

a) `0..25`

b) `15..30`

c) `10..25`

d) `30..35`

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 (null(s)) {
    yield []
  } else {
    while (e = grok(tail(s))) {
      yield e;
      yield reverse(e) ++ [ head(s) ];
    }
  }
}
```

Al invocar `grok([1,2,3])`, ¿cuál será la *primera* lista de tres elementos que será emitida por el iterador?

a) `[1,2,3]`

b) `[3,2,1]`

c) `[1,3,2]`

d) `[2,3,1]`

Pregunta 2 - 4 puntos

A continuación se presenta un procedimiento recursivo en un lenguaje imperativo en el cual se pueden pasar como argumentos arreglos de tamaño dinámico.

```
int grok(int foo[], int wtf, int bar, int baz) {
  if (wtf == bar)
    return foo[wtf];
  qux := meh(foo,wtf,bar);
  ugh := qux - wtf + 1;
  if (baz == ugh)
    return foo[qux]
  else if (baz < ugh)
    return grok(foo,wtf,qux-1,baz)
  else
    return grok(foo,qux+1,bar,baz-ugh)
}
```

Simplifique el procedimiento para convertir la recursión de cola en iteración estructurada, sin modificar la semántica del mismo, usando la técnica automática discutida en clase. Puede definir variables adicionales si cree que son convenientes.

Nota: Ud. estará actuando como un compilador así que **no** necesita comprender lo que hace el fragmento, sino simplemente transformarlo de forma automática. Si Ud. presenta una implantación iterativa manual astuta porque "dedujo" lo que hace el fragmento, **no** recibirá crédito alguno.

```
int grok(int foo[], int wtf, int bar, int baz) {
  while (42 > 0) {
    if (wtf == bar)
      return foo[wtf];
    qux := meh(foo,wtf,bar);
    ugh := qux - wtf + 1;
    if (baz == ugh)
      return foo[qux]
    else if (baz < ugh)
      bar := qux - 1
    else {
      wtf := qux + 1;
      baz := baz - ugh;
    }
  }
}
```

Pregunta 3 - 5 puntos

Aproveche *exclusivamente* listas por comprensión para escribir la función Haskell

```
allStrings :: String -> [String]
```

tal que genere las *infinitas* palabras posibles construidas utilizando los caracteres del `String` suministrado como argumento. Si cree que le sirve de algo, puede suponer que en el argumento no hay símbolos repetidos.

```
ghci> allStrings []
[""]
ghci> take 7 (allStrings "a")
["","a","aa","aaa","aaaa","aaaaa","aaaaaa"]
ghci> take 10 (allStrings "abc")
["","a","b","c","aa","ab","ac","ba","bb","bc"]
```

Se trata de una lista infinita que puede generarse a partir de sí misma. Como uno de los generadores es infinito, debe estar de primero.

```
allStrings cs = "" : [ (c:w) | w <- allStrings cs, c <- cs ]
```

Pregunta 4 - 5 puntos

Si se tiene una lista de datos para los cuales se quiere calcular estadísticos simples (conteo, promedio, máximo y mínimo), uno puede escribir la función Haskell

```
stats :: [Double] -> (Integer,Double,Double,Double)
stats fs = (n, sum fs / fromIntegral n, maximum fs, minimum fs)
  where n = length fs
```

Esa implantación, a pesar de ser muy compacta y declarativa, tiene el inconveniente de que recorrerá *cuatro* veces la lista, lo cual la hace ineficiente en tiempo, e incapaz de procesar listas de gran tamaño. Use *exclusivamente* recursión de cola para escribir la función nuevamente, pero de manera que se recorra la lista *una* sola vez. Si lo desea, puede usar el *fold* adecuado para este caso.

La solución ideal emplea un `foldl'` porque se trata de una lista finita, con al menos un elemento, que debe recorrerse una sola vez y para operar con aritmética.

```
stats :: [Double] -> (Integer,Double,Double,Double)
stats (f:fs) = (n, s / fromIntegral n, mayor, menor)
  where (n,s,mayor,menor) = foldl' go (1,f,f,f) fs
        go (c,t,h,l) x    = (c+1, t+x, max h x, min l x)
```

También resulta aceptable una solución empleando recursión de cola.

```
stats :: [Double] -> (Integer,Double,Double,Double)
stats (f:fs) = go (1,f,f,f) fs
  where go (c,t,h,l) []     = (c, t / fromIntegral c, h, l)
        go (c,t,h,l) (x:xs) = go (c+1, t+x, max h x, min l x) xs
```