Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-3641 - Lenguajes de Programación I
Septiembre-Diciembre 2010

Carnet:__________________________

Nombre:__________________________

Examen II

(Versión A – 35 puntos)

Pregunta 0	Pregunta 1	Pregunta 2	Total
20 puntos	7 puntos	8 puntos	35 puntos

Pregunta 0 - 20 puntos

Esta pregunta consta de diez (10) subpreguntas de selección, numeradas de 0.0 a 0.9. 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 dos (2) puntos. Tres subpreguntas incorrectas eliminan una correcta. Las subpreguntas omitidas (marcadas en la opción e) no suman ni restan puntos.

0.0. Considere el siguiente programa escrito en pseudocódigo:

```
var n := 10 : int
    m := 7 : int

    fun f () : boolean
    begin
        m := m + 2
        return m < n
    end

    fun g () : boolean
    begin
        n := n + 2
        return m < n
    end

    fun h () : boolean
    begin
        n, m := m + 1, n - 2
        return n != m
    end

begin
    if (f() /\ (g() \/ h())) then
        escribir(n + m)
    else
        escribir(n - m)
    fi
end
```

en el cual `escribir` es una subrutina que escribe en pantalla el valor pasado como argumento, considere que la definición del lenguaje indica que el orden de evaluación de expresiones es de izquierda a derecha.

Considerando que los operadores booleanos /\ (conjunción) y \/ (disyunción), son implementados con evaluación con corto-circuito por el compilador, ¿cuál es el valor que se muestra en pantalla?

a) 18

b) 19

c) 20

d) **21**

e) No sabe / No contesta.

0.1. Continúe con el programa dado en la subpregunta (0.0), pero ahora considerando que los operadores booleanos /\ y \/ son implementados con evaluación sin corto-circuito por el compilador, ¿cuál es valor que se muestra en pantalla?

a) 18

b) 19

c) **20**

d) 21

e) No sabe / No contesta.

0.2. Continúe con el programa dado en la subpregunta (0.0), y continúe considerando que los operadores booleanos /\ y \/ son implementados con evaluación sin corto-circuito por el compilador, considere ahora que el lenguaje no coloca restricción alguna sobre el orden de evaluación de los operandos de un operador, por tanto cada implementación puede usar el orden que más convenga. ¿cuál(es) de los siguientes valores puede, bajo alguna implementación de este pseudolenguaje, ser mostrada en pantalla?

a) **20 (pero no 18 o 21)** *los valores posibles son 20, -4 y 0.*

b) 20 y 21 (pero no 18)

c) 18, 20 y 21

d) Ninguna de las anteriores.

e) No sabe / No contesta.

0.3. Considere el tipo de dato:

```
type referencia = ^estructura
type estructura = record
                    elem0, elem1 : ^T;
                    subestructura0, subestructura1,
                    subestructura2 : referencia
                  end
```

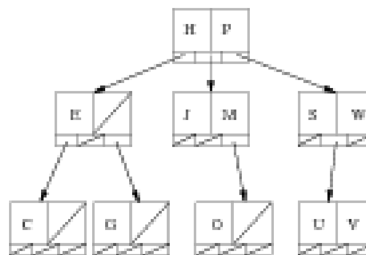
y la siguiente subrutina que provee un iterador verdadero, haciendo uso de la instrucción `yield`:

```
proc recorrido ( r : referencia )
var
    e : T
begin
    if r != nullref then
        begin
            if r^.subestructura2 != nullref then
                for e in recorrido (subestructura2) do
                    yield e
            if r^.elem1 != nullref then
                yield r^.elem1^
            if r^.subestructura1 != nullref then
                for e in recorrido (subestructura1) do
                    yield e
            if r^.elem0 != nullref then
                yield r^.elem0^
            if r^.subestructura0 != nullref then
                for e in recorrido (subestructura0) do
                    yield e
        end
    fi
end
```

Considere ahora la iteración

```
for x in recorrido(a) do
    ...
```

donde `a` es una referencia a la siguiente estructura con elementos asociados `H` y `P`:



donde las letras representan objetos de tipo `T`, a los que se hace referencia en la estructura correspondiente, y las cajas con lineas cruzadas representan referencias nulas.

¿cuál es el primer elemento de la enumeración que es considerado por la iteración?

a) `S`
b) **W**
c) `U`
d) Ninguno de los anteriores.
e) No sabe / No contesta.

0.4. Continúe con el iterador y la iteración dados en la subpregunta (0.3), ¿cuál es el segundo elemento de la enumeración que es considerado por la iteración?

a) `S`
b) `W`
c) `U`
d) **Ninguno de los anteriores. (V)**
e) No sabe / No contesta.

0.5. Continúe con el iterador y la iteración dados en la subpregunta (0.3), ¿cuál es el tercer elemento de la enumeración que es considerado por la iteración?

a) `S`
b) `W`
c) **U**
d) Ninguno de los anteriores.
e) No sabe / No contesta.

Nota: el iterador realiza un recorrido *post-order* sobre la estructura.

0.6. Considere las siguientes definiciones de tipos:

```
type t = array[0..10] of integer;
type u = array[0..10] of integer;
type w = array[2..12] of integer;
type v = u
```

¿cuál de las siguientes afirmaciones es cierta?

a) Los tipos `t` y `u` son equivalentes, sin importar si el lenguaje considera equivalencia por nombre o estructural.
b) Los tipos `u` y `v` son equivalentes sólo si el lenguaje considera equivalencia por nombre.
c) Los tipos `w` y `t` son tipos equivalentes solamente si el lenguaje considera equivalencia estructural.
d) **Los tipos `v` y `t` son estructuralmente equivalentes.**
e) No sabe / No contesta.

0.7. Considere la siguiente sección de código, escrito en Pascal:

```
var a : set of 0..10;
var b : set of 15..20;
var c : set of u..v;

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

¿cuál es el rango más pequeño con el cual debe ser declarada la variable c para que un buen compilador determine estáticamente que la asignación es válida y no se requiere realizar ninguna verificación dinámica asociada a ésta?

a) 0..10

b) **15..20**

c) 10..15

d) 0..20

e) No sabe / No contesta.

0.8. Considere la siguiente declaración de variables, escrita en pseudocódigo:

```
var n := 7 : int;
var m : array [0..n] of array [5..p] of T;
```

Si se quiere calcular la dirección de `m[i][6]`, sabiendo que m esta almacenada en la dirección `5000`, ¿cuál de las siguientes afirmaciones es cierta?

a) La dirección de ese elemento es `5000+(6*n*sizeof(T))+i*sizeof(T)`, siempre que estemos en un lenguaje con almacenamiento de arreglos por columnas (*column-major layout*).

b) La dirección de ese elemento es `5000+sizeof(T)+i*(p-5)*sizeof(T)`, siempre que estemos en un lenguaje con almacenamiento de arreglos por filas (*row-major layout*).

c) **La dirección de ese elemento es `*(5000+i*X)+sizeof(T)`, siempre que estemos en un lenguaje con almacenamiento de arreglos con apuntadores a filas (*row-pointer layout*), donde `X` es el tamaño de un apuntador.**

d) Ninguna de las anteriores.

e) No sabe / No contesta.

Nota:

- Si el arreglo estuviese en *column-major*, la fórmula **correcta** sería `5000+((6-5)*n*sizeof(T))+i*sizeof(T)`.
- Si el arreglo estuviese en *row-major*, la fórmula **correcta** sería `5000+sizeof(T)+i*(p-5+1)*sizeof(T)`.

0.9. Considere las siguientes declaraciones en Pascal:

```
type T = packed record
                w : char;
                x : integer;
                y : boolean;
                z : real;
              end;

var a : packed array [1..10] of T;
```

Contemplando una implementación que utiliza 1 byte para caracter, 4 bytes para entero, 1 byte para booleano y 8 bytes para real, ¿cuántas palabras (de 4 bytes) ocupará el arreglo `a` en memoria?

a) 40.

b) **35.**

c) 50.

d) Ninguna de las anteriores.

e) No sabe / No contesta.

Nota: tanto la estructura como el arreglo están empaquetados así que no hay espacio ni entre los campos de la estructura, ni entre los elementos del arreglo – no importa la alineación.

Pregunta 1 - 7 puntos

A continuación se presenta una función recursiva en pseudocódigo, en ella se ha utilizado el tipo `Lista` que representa una referencia a un elemento de una lista enlazada:

```
type Lista = ^record
                 elem : T;
                 resto: Lista
              end;

fun inversa (Lista l) -> Lista
begin
    if (l = nullref) then
        return l
    else
        var laux : Lista;
        laux := l^.resto;
        l^.resto := nullref;

        return concatenacion(inversa(laux), l)
    fi
end
```

donde `concatenacion` es una función tal que dadas dos listas, posiblemente vacias, retorna la concatenación de ambas.

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

```
fun inversa (Lista l) -> Lista
begin
  return asrevni(l, nullref)
end

fun asrevni(Lista l, Lista a) -> Lista
begin
  if (l = nullref) then
    return a
  else
    var r : Lista;
    r := l^.resto;
    l^.resto := a;
    return asrevni(r,l);
  fi
end
```

- No se puede cambiar la firma de la función original – utiliza una función auxiliar recursiva de cola.
- Técnica de parámetro acumulador.
- No es necesario usar la función de concatenación – trivial agregar **un** elemento al principio del acumulador.
- Refrito de una pregunta formulada en Enero-Marzo 2008.
- Soluciones iterativas inaceptables aunque funcionen – queremos evaluar si saben inducir recursión de cola.

Pregunta 2 - 8 puntos

La detección de "referencias colgantes" (*dangling references*) puede realizarse, al menos probabilísticamente, con el mecanismo de "Cerraduras y Claves" (*Locks-and-Keys*). Para implantar este mecanismo es necesario agregar a cada referencia y cada bloque referenciado una clave; si éstas coinciden entonces la referencia aún es valida. Además, si se desea implementar "recolección de basura" (*Garbage Collection*) usando "conteo de referencias" (*Reference Counts*) en el lenguaje, será necesario incluir en cada bloque el número de referencias hacia ese bloque.

La detección de referencias colgantes puede ser implementado en Haskell haciendo uso de la siguiente función:

```
detectarReferenciaColgante :: Referencia -> [Bloque] -> Bool
```

donde el tipo de datos `Referencia` y `Bloque` son definidos como:

```
type Referencia = (Direccion, Int)
type Bloque = (Direccion, Int, Int, Objeto)
```

donde en la tupla asociada a `Referencia` la primera componente corresponde a la ubicación del bloque, y la segunda a la clave; y en la tupla asociada a `Bloque` la primera componente corresponde a la ubicación del bloque, la segunda corresponde a la clave, la tercera corresponde al número de referencias a ese bloque, y la cuarta corresponde como tal al objeto almacenado en heap.

Dado que se tienen dos referencias validas, es posible asignar una a la otra, y esto modifica el estado actual del *heap* (esto es, la lista de bloques que representa el *heap*), esta operación puede ser implementada en Haskell mediante la siguiente función:

```
asignar :: Referencia -> Referencia -> [Bloque] -> [Bloque]
```

y así el resultado de evaluar `asignar p q heap` corresponde al *heap* resultante luego de realizar la operación `p := q`.

Se desea que Ud.

- Implemente la función `detectarReferenciaColgante`. Puede suponer que un valor positivo representa una clave válida, que toda referencia pasada como argumento tendrá una clave válida y que su dirección corresponderá con la de algún bloque presente en la lista dada como argumento.

- Implemente la función `asignar`. Puede suponer que las referencias dadas como argumento corresponden a referencias no colgantes, cuyas direcciones están presentes en el *heap* pasado como argumento.

Nota: Ud. recibirá **dos (2)** puntos adicionales si implementa ambas funciones usando *exclusivamente* funciones de orden superior.

Mi solución con recursión explícita para `detectarReferenciaColgante`:

- Si la referencia no aparece en el *heap* obviamente se trata de una referencia colgante.
- Si la referencia aparece en el *heap*, tiene que aparecer **una** sola vez pues cada bloque tiene una dirección única. En ese caso, simplemente tengo que determinar si las llaves son iguales (**no** es una referencia colgada) o no (**si** es una referencia colgada).

```
detectarReferenciaColgante :: Referencia -> [Bloque] -> Bool
detectarReferenciaColgante _ [] = True
detectarReferenciaColgante r@(d,k) ((d',k',_,_):bs)
  | d == d'   = k /= k'
  | otherwise = detectarReferenciaColgante r bs
```

Mi solución con recursión explícita para `asignar`, parte de dos suposiciones:

- Ambas referencias están en el *heap* – claro según el enunciado de la pregunta.
- La recolección de basura y la manipulación de las claves en el modelo de referencia son problemas de **otras** funciones – se desprende de nuestro estudio sobre recolección de basura.

Entonces, sólo tengo que recorrer el *heap* **una** vez:

- Si llegué al final del *heap* devuelvo un *heap* vacío para que la recursión combine los elementos.
- Si no estoy al final del heap:
  1. Si encontré el bloque **destino** debo decrementar su contador de referencias, y llegó a cero, también debo invalidar la llave para que pueda detectarse como colgada en el futuro.
  2. Si encontré el bloque **origen** debo incrementar su contador de referencias.
  3. Si es cualquier otro bloque, permanece igual.

```
asignar :: Referencia -> Referencia -> [Bloque] -> [Bloque]
asignar dst        src        []              = []
asignar dst@(d,_)  src@(s,_)  ((r,k,c,o):bs)
  | r == d && c == 1 = (r,0,0,o):bs
  | r == d && c >  1 = (r,k,c-1,o):bs
  | r == s           = (r,k,c+1,o):bs
  | otherwise        = (r,k,c,o):bs
```

Nota: en las respuetas solamente se evalúa el razonamiento si está manifestado en el código – las explicaciones son superfluas.