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

Examen I

(30 puntos)

Nombre:

Carnet:

1. **(5 puntos)** Considere cuidadosamente las siguientes cuestiones y seleccione exactamente una respuesta de las alternativas que se presentan. Cada cuestión contestada correctamente vale **un (1) punto** pero una cuestión contestada incorrectamente **resta medio punto**. Si Ud. selecciona más de una opción, la pregunta se considera contestada **incorrectamente**.

   (a) En un lenguaje con alcance estático **siempre** es cierto

      i. **Si tiene clausuras, las funciones deben ser de segunda o primera clase.**
      Si el lenguaje tiene clausuras, entonces tiene que poder crear una función en un contexto estático y para eso la función tiene que haber sido pasada como parámetro, asignada a una variable o retornada como resultado de una función.

      ii. No puede haber reservas de memoria dinámicas.
      No, pues el alcance y el almacenamiento son cosas diferentes. Lenguajes con alcance estático como C o Java permiten hacer reserva de memoria dinámica.

      iii. La cadena dinámica es inútil.
      No, pues la cadena dinámica es la que mantiene la relación entre rutina llamada y llamador, para poder regresar de la invocación de rutinas, de modo que es indispensable para la ejecución de los programas.

      iv. Las subrutinas deben tener un número fijo de parámetros.
      No, pues el número de parámetros de una función sólo depende del mecanismo de pasaje de parámetros que es independiente del alcance empleado por el lenguaje. En particular C tiene alcance estático y soporta funciones con un número variable de parámetros, como `printf`.

   (b) Una variable local con almacenamiento global.

      i. Se almacena en la pila de ejecución aunque sea global, por su condición de local a una función.
      No, pues si hemos dicho que tiene almacenamiento global es contradictorio que se almacene en la pila de ejecución.

      ii. Es accesible desde subrutinas anidadas a través de la cadena estática.
      No, pues si tiene almacenamiento global se accede directamente pues su ubicación en memoria es conocida a tiempo de compilación, por lo que la cadena estática es irrelevante para accederle.

      iii. Se elabora solamente si la rutina es invocada, así que su almacenamiento depende del flujo de ejecución del programa.
      No, pues si tiene almacenamiento global su elaboración ocurre en el momento en que se inicia el programa, independientemente de su flujo, cuando se reserva espacio para las variables globales.

      iv. **Se inicializa la primera vez que se ingresa a la rutina y su valor persiste entre invocaciones hasta que termine el programa.**
      Se trata de aquellas variables como las `static` de C que tienen almacenamiento global pero su alcance está limitado al cuerpo de la función o bloque en el cual son declarados.

(c) En C, solamente una de las siguientes afirmaciones es cierta, ¿cuál?

i. Los operadores `++` y `--` prefijos incrementan o decrementan su operando después de proveer su valor al contexto de la expresión.
No, los operadores *prefijos* incrementan o decrementan su operando *antes* de proveer su valor al contexto de la expresión.

ii. **Los operadores `++` y `--` prefijos no son más que *syntactic sugar* para los operadores compuestos `+=` y `-=`.**

iii. Las expresiones `foo = bar++` y `(bar += 1, foo = bar)` son equivalentes.
No. Sea `bar = 41`. La primera expresión resulta en `foo = 41` y `bar = 42` pues el operador `++` es *sufijo* de modo que primero entrega el valor actual del operando (41) y luego incrementa el operando. La segunda expresión resulta en `foo = 42` y `bar = 42` pues el operador de incremento opera sobre `bar` y luego su valor es copiado en `foo` en la asignación.

iv. Los operadores `++` y `--` siempre se traducen a la operación de máquina `INC` o `DEC` (según el procesador).
No, pues el comportamiento de los operadores depende si son prefijos o sufijos, en consecuencia no siempre se pueden traducir a las operaciones de máquina de incremento y decremento.

(d) Los módulos como tipo:

i. Deben operar con alcance cerrado para ofrecer abstracción completa.
No, pues el alcance cerrado controla lo que el módulo puede *importar* dentro de sí, mientras que la abstracción es derivada de lo que el módulo pueda *exportar* fuera de sí.

ii. Manejan múltiples instancias del mismo tipo replicando el código.
No, precisamente los módulos como tipos de datos son un refinamiento de los módulos como administradores para no tener que replicar el código.

iii. **Si tuvieran herencia, serían como las clases de la orientación a objeto.**

iv. Los datos se pasan a las funciones como argumentos explícitos.
No, pues esa es una característica de los módulos como administradores al tener que invocar `funcion(Dato, ...)` en lugar de `Dato.funcion(...)`

(e) En los lenguajes que usan el modelo de referencia

i. Cualquier expresión puede utilizarse como *l-value*.
No, pues independientemente del modelo una expresión aritmética simple no puede ser utilizada como *l-value* pues no hace referencia a una ubicación de memoria.

ii. Los valores primitivos (números, caracteres) siempre son manejados como valores.
No, pues en algunos lenguajes con modelo de referencia los valores primitivos se almacenan **una sola vez** en memoria y todos sus usos se manejan como referencias a ellos (ejemplo en el libro de texto).

iii. Todos los objetos deben elaborarse en el *heap*.
No, pues si existen variables globales estás son *referencias* a objetos en el *heap* pero están en el área global. Lo mismo ocurre con variables locales a funciones o bloques dentro de funciones que son *referencias* a objetos en el *heap* pero están en el registro de activación dentro de la pila.

iv. **Una asignación de la forma `a := b` siempre produce un *alias*.**
Pues si `a` y `b` son variables, entonces son referencias a objetos en el *heap* y al tomar el valor de `b` (que es una referencia) y copiarlo en `a`, ambas variables harán referencia al mismo objeto, siendo aliases del mismo.

2. Recursión:

(a) **(4 puntos)** Considere la siguiente implantación del Algoritmo de Búsqueda Binaria escrita en C++ en la cual se asume un arreglo `A` con números naturales ordenados. El algoritmo retorna -1 en caso que el `Valor` a buscar no se encuentre en el arreglo:

```
int bsearch(int A[], int Inicio, int Final, int Valor)
{
    if (Inicio > Final) {
        return (-1);
    } else {
        int Mitad = (Inicio + Final) / 2;
        if (Valor == A[Mitad])
          return Mitad;
        else if (Valor < A[Mitad])
          return bsearch(A,Inicio,Mitad-1,Valor);
        else
          return bsearch(A,Mitad+1,Final,Valor);
    }
}
```

Identifique y elimine la recursión de cola.

Ambas llamadas recursivas corresponden a recursión de cola, puesto que a su retorno no queda ningún cómputo pendiente en la invocación actual de la función. Para eliminarla es necesario convertir ambas llamadas en una iteración:

```
int bsearch(int A[], int Inicio, int Final, int Valor)
{
    while (Inicio <= Final) {
        int Mitad = (Inicio + Final) / 2;
        if (Valor == A[Mitad]) return Mitad;
        if (Valor < A[Mitad]) {
            Final = Mitad - 1
        } else {
            Inicio = Mitad + 1
        }
    }
    return -1;
}
```

(b) **(4 puntos)** Considere la siguiente función en Haskell para convertir un número entero en una lista con sus dígitos, e.g. `digitos 6942 = [6,9,4,2]`

```
digitos :: Int -> [Int]
digitos n | n < 10    = [n]
          | otherwise = digitos (n `div` 10) ++ [n `mod` 10]
```

Reescriba la función para introducir recursión de cola.

La técnica estándar para introducir recursión de cola consiste en acumular el trabajo hacia la llamada recursiva. En este caso, se utiliza una función auxiliar que traslada la construcción de la lista al parámetro acumulador. La manera *eficiente* de hacerlo es agregando elementos al principio con `(:)` aunque también se puede hacer concatenando listas con `(++)`. La firma de la función principal **no** puede ser cambiada, sino que debe ser reescrita utilizando la función auxiliar

```
digitos n = digitos' n []
    where digitos' n ds
            | n < 10    = (n:ds)
            | otherwise = digitos' (n `div` 10) ((n `mod` 10):ds)
```

3. **(8 puntos)** Suponga un lenguaje *imperativo* con iteradores reales, que son declarados como una rutina cualquiera y donde la producción de valores se indica mediante la instrucción `yield`. Por simplicidad suponga que el lenguaje permite manipular listas como en Haskell, i.e.

   - `a.head` retorna el primer elemento de la lista `a`.
   - `a.tail` retorna la lista `a` **sin** el primer elemento.
   - `a ++ b` concatena las listas `a` y `b`.
   - `[]` es la lista vacía.
   - `[x]` construye una lista con el elemento `x`.

   Escriba un iterador `potencia(a : lista)` que produzca todas las listas que pueden construirse a partir de la lista `a` en el sentido de la operación "partes de un conjunto", retornándolas una a una con cada invocación. Por ejemplo, `potencia([1,2,3])` debería retornar `[]`, `[1]`, `[2]`, `[3]`, `[1,2]`, `[1,3]`, `[2,3]` y `[1,2,3]` (no necesariamente en ese orden). **Pista:** si fijo un elemento y calculo la potencia del resto, entonces tengo todas las partes que no lo contienen y puedo construir todas las partes que si le contienen.

   Nótese que las partes de un conjunto son a su vez un conjunto, por lo tanto no puede incluir valores repetidos. El algoritmo típico para calcular las partes de un conjunto se define recursivamente como:

   (a) El conjunto vacío forma parte de las partes del conjunto de cualquier conjunto.

   (b) Para cualquier conjunto:

      i. Se fija un elemento del conjunto.
      ii. Se calculan las partes del resto del conjunto (que a su vez son conjuntos).
      iii. Para cada conjunto tomado de las partes del resto del conjunto calculados en el paso anterior se emiten dos conjuntos: el conjunto sin modificación y el conjunto agregando el elemento fijo.

   Este algoritmo puede combinarse trivialmente en un iterador recursivo con un componente interno de iteración si se hace uso *consistente* de los tipos de datos empleando *exclusivamente* las funciones permitidas en el enunciado de la pregunta. Se utilizan listas para representar los conjuntos y se fija siempre el primer elemento de la lista.

```
Lista potencia( a : Lista )
{
    if (a == []) {
        yield []
    } else {
        while (p = potencia( a.tail )) {
            yield p;
            yield [a.head] ++ p;
        }
    }
}
```

   Nótese que la función head retorna un *elemento* cuyo tipo no conocemos y no podemos "inventar" pero que en todo caso **no es** una lista así que sería incorrecto definir eso como una variable separada de tipo lista (o cualquier otro tipo inventado) por eso lo fijo *implícitamente* al envolverlo en el constructor de listas.

4. Considere el siguiente programa escrito en pseudocódigo

```
int u = 20;
int v = 11;
int w = 9;
proc add( z : int )
    v := v + u + z
proc bar( fun : proc)
    int u := w;
    fun(v)
proc foo( x : int, w : int)
  int v := x;
  bar(add)
main
  foo(u,17)
  print(v)
end;
```

¿Qué imprime el programa?

(a) **(3 puntos)** Asumiendo que el lenguaje utiliza alcance **estático**.

(b) **(3 puntos)** Asumiendo que el lenguaje utiliza alcance **dinámico** y ***deep binding***.

(c) **(3 puntos)** Asumiendo que el lenguaje utiliza alcance **dinámico** y ***shallow binding.***

En **todos** los casos asuma que las clausuras se construyen en el momento en que las funciones son **pasadas como parámetros.**

Nota: si solamente muestra los resultados (aunque sean correctos) no obtendrá puntos; es **imprescindible** exhibir los contenidos de la pila de ejecución hasta el momento en que se invoca la función `add` incluyendo las cadenas dinámica y estática así como las clausuras construidas.