Paradigmas de la Programación – Primer Parcial

19 de Abril de 2018

Apellido y Nombre: ____

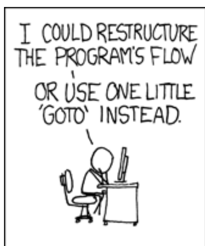
1. **[10 pt.]** La siguiente expresión está mal tipada:

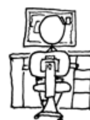
   ```
   f (a, b) = a(b) > b(a) + a
   ```

   Muestre con el árbol para inferencia de tipos dónde se encuentra el conflicto y en qué consiste. Cómo podría resolver este conflicto un lenguaje de tipado fuerte? y uno de tipado débil?

2. **[15 pt.]** El siguiente código (resumido) de un driver SCSI para Linux es spaghetti. ¿Por qué decimos que es spaghetti, cuál es la principal diferencia con el código estructurado? ¿Si el lenguaje obliga a usar código estructurado, qué estructura de datos puede usar el compilador para manejar la memoria de forma más eficiente? Reescriba el driver en pseudocódigo para que sea estructurado y no spaghetti.

```
wait_nomsg:
        if ((inb(tmport) & 0x04) != 0) {
                goto wait_nomsg;
        }
        ...
        for (n = 0; n < 0x30000; n++) {
                if ((inb(tmport) & 0x80) != 0) {
                        goto wait_io;
                }
        }
        goto TCM_SYNC;
wait_io:
        for (n = 0; n < 0x30000; n++) {
                if ((inb(tmport) & 0x81) == 0x0081) {
                        goto wait_io1;
                }
        }
        goto TCM_SYNC;
wait_io1:
        ...
TCM_SYNC:
        ...
```





3. **[15 pt.]** En el siguiente programa en Lua, diagrame el estado en el que se encuentra la pila de ejecución al terminar de ejecutar las líneas señaladas con A, B, C y D en el texto del programa[1].

```
a = 1
b = 2
c = 3

function test()

  local function g()
    local a = 2
    c = a + 4                ---> D
  end

  local function f()
    local c = 5
    b = 4                    ---> C
    g()
  end
                             ---> B
  f()
end
                             ---> A
test()
```

4. **[10 pt.]** De estos dos fragmentos de códigos, el primero en Pascal y el segundo en C++, cuál es un ejemplo de sobrecarga y cuál es un ejemplo de polimorfismo? Justifique su respuesta.

```
function Add(x, y : Integer) : Integer;
begin
    Add := x + y
end;

function Add(s, t : String) : String;
begin
    Add := Concat(s, t)
end;
```

```
class List<T> {
    class Node<T> {
        T elem;
        Node<T> next;
    }
    Node<T> head;
    int length() { ... }
}
```

5. **[15 pt.]** El siguiente fragmento de código funciona porque se aplica un tipo de polimorfismo que recurre al subtipado que se articula a través de la herencia. En este lenguaje, el subtipado permite que una función se defina para tomar objetos de tipo T pero la función también funciona correctamente si se le pasan objetos de tipo S, siempre que S sea subtipo de T. Esto es así porque se aplica el principio de Liskov, enunciado por Bárbara Liskov y Jeannette Wing de la siguiente forma:

[1] Para ser más precisos, al terminar de ejecutar el equivalente en código máquina a esas líneas del código en Lua.

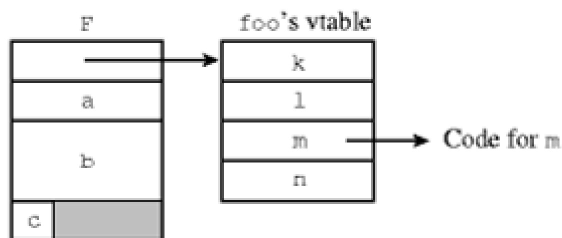
Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Explique con sus propias palabras por qué funciona este código, recurriendo a los conceptos de polimorfismo, subtipado y herencia.

```
abstract class Animal {
    abstract String talk();
}

class Cat extends Animal {
    String talk() {
        return "Meow!";
    }
}

class Dog extends Animal {
    String talk() {
        return "Woof!";
    }
}

static void letsHear(final Animal a) {
    println(a.talk());
}

static void main(String[] args) {
    letsHear(new Cat());
    letsHear(new Dog());
}
```

6. **[15 pt.]** En C++, la `vtable` guarda las funciones definidas como virtuales, tal como se ve en el siguiente gráfico:

```
class foo {
    int a;
    double b;
    char c;
public:
    virtual void k ( ...
    virtual int l ( ...
    virtual void m ();
    virtual double n( ...
    ...
} F;
```



Por defecto, las funciones de una clase no se definen como virtuales sino como estáticas. Teniendo en cuenta que una de las principales decisiones de diseño de C++ es la eficiencia, explique por qué se evita tanto como sea posible que las funciones en C++ se definan como virtuales, explicando qué involucra procesar una función virtual, y recurriendo a los conceptos de overhead y flexibilidad.

7. **[10 pt.]** Estos dos fragmentos de código tienen la misma semántica, pero en uno hay inversión de control e inyección de dependencia, y en el otro no. Explique cuál es cuál. En el que tiene inversión de control, identifique el hot spot donde se puede parametrizar la dependencia que se inyecta en lugar de que esté incrustada en el código.

```
public class EditorDeTexto {

    private CorrectorOrtografico corrector;

    public EditorDeTexto(CorrectorOrtografico corrector) {
        this.corrector = corrector;
    }
}

CorrectorOrtografico co = new CorrectorOrtografico;
EditorDeTexto textEditor = new EditorDeTexto(co);
```

```
public class EditorDeTexto {

    private CorrectorOrtografico corrector;

    public EditorDeTexto() {
        this.corrector = new CorrectorOrtografico();
    }
}
```

8. **[10 pt.]** Lea el siguiente texto y explique con sus propias palabras el nivel de visibilidad `friend` en C++, y relaciónelo con los niveles de visibilidad `private` y `protected`. Explique cómo los niveles de visibilidad se relacionan con el encapsulamiento y con la separación interfaz – implementación, y cuál es su utilidad en el proceso de desarrollo del software.

   In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to "friends".

   Friends are functions or classes declared with the friend keyword.

   A non-member function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend.