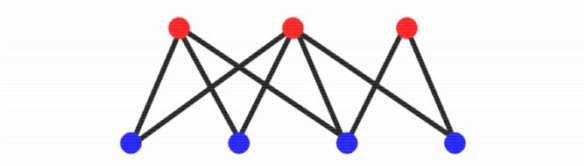
Control 13 de Febrero de 2012

Un grafo no dirigido se llama **bipartito** (o **2-coloreable**) si sus vértices pueden colorearse usando solamente dos colores (Rojo y Azul, p.e.) de tal forma que dos vértices adyacentes tengan distinto color. Un coloreado (compatible) es una asignación de colores a todos los vértices de forma que se cumpla la condición anterior. Estos grafos se suelen dibujar colocando los rojos arriba y los azules abajo, como en la siguiente figura:



El objetivo de este ejercicio consiste en describir en Haskell y Java el algoritmo que describiremos a continuación para comprobar si un grafo conexo es bipartito, devolviendo en su caso una asignación de colores a los vértices.

El algoritmo es una variante del recorrido en profundidad de un grafo. Como *estructura de vértices pendientes de visitar-colorear* se utilizará un `stack` de pares, donde cada par indica un vértice pendiente de explorar y el color compatible que debe tener, que será determinado por el color de alguno de sus adyacentes ya visitado. También vamos a utilizar un diccionario que asocia a cada vértice visitado el color asignado por el algoritmo.

Inicialmente, el diccionario está vacío. Comenzamos asignando un color arbitrario (por ejemplo, *Red*) a uno de los vértices (*src*, o source), y apilamos en el stack el par *(src,Red)*. El resto del algoritmo corresponde al bucle siguiente:

1. Si la pila está vacía, el grafo es bipartito, y el diccionario contiene un coloreado compatible y termina el bucle. En otro caso:
   a. Depilamos de la cima un par formado por un vértice *v* y su color *c*.
   b. Si el vértice *v* no ha sido visitado (es decir, no aparece en el diccionario), significa que no tiene asignado todavía color, pero éste deberá ser *c*. Entonces visitamos este vértice. Para ello introducimos en el diccionario la asociación *v* → *c*. Además, todos los sucesores de *v* que no hayan sido visitados (es decir, que no aparezcan en el diccionario) se apilan en el *stack* junto con el color que deberán tener cuando sean visitados (este color serán común a todos y diferente de *c*, el color de *v*).
   c. En otro caso el vértice *v* fue visitado con anterioridad con cierto color *cd* que será el color asociado en el diccionario. Si el color *cd* no es el mismo que el color *c* extraído de la pila, no existe un coloreado compatible y el grafo no es bipartito.
   d. En otro caso, el color asociado en el diccionario y el color extraído de la pila son los mismos, y volvemos al paso 1 para procesar el resto de la pila.

A) En Java, los colores pueden representarse por el siguiente tipo enumerado:

```
public static enum Color {Red, Blue};

private static Color nextColor(Color c) {
  return (c == Color.Blue) ? Color.Red : Color.Blue;
}
```

La siguiente clase `Pair` (proporcionada junto a resto del código) puede utilizarse para representar pares:

Control 13 de Febrero de 2012

```
public class Pair<A,B> {
  private A a;
  private B b;

  public Pair(A x, B y) { a = x; b = y; }

  public A first() { return a; }
  public B second() { return b; }

  @Override public String toString() { return "Pair("+a+","+b+")"; }
}
```

Para implementar el algoritmo, debemos completar la siguiente clase Java:

```
public class BiPartite<V> {
  public static enum Color {Red, Blue};

  private static Color nextColor(Color c) {
    return (c == Color.Blue) ?Color.Red:Color.Blue;
  }

  private Stack<Pair<V,Color>> stack; // stack with vertices and colors
  private Dictionary<V,Color> dict;   // dictionary: vertex -> color
  private boolean biColored;

  public BiPartite(Graph<V> graph) {
    dict = new HashDictionary<V,Color>();
    stack = new StackList<Pair<V,Color>>();
    biColored = true;

    if (graph.numVertices() == 0)
      return; // empty graph is bipartite

    V src = graph.vertices().iterator().next(); // initial vertex
    stack.push(new Pair<V,Color>(src,Color.Red));

    while (!stack.isEmpty()) {
       // completad el bucle según el algoritmo descrito
    }
  }

  public boolean isBicolored() { // returns true if graph is bipartite
   return biColored;
  }

  public Dictionary<V,Color> biColored() { // returns color assignment
    return bicolored ? dict : null;
  }
}
```

NOTA: cada alumno deberá descargar desde el campus virtual el código correspondiente a las estructuras de datos necesarias para la realización del ejercicio, así como un fichero con la clase anterior BiPartite.java que deberá completar. Para la comprobación de la corrección del ejercicio deberá así mismo utilizar el programa BiPartiteTest.java.

Control 13 de Febrero de 2012

B) En Haskell, las estructuras de datos necesarias pueden manejarse con el código:

```
import Graph
import qualified Dictionary as D
import qualified Stack as S

data Color = Red | Blue deriving (Eq,Ord,Show)

nextColor :: Color -> Color
nextColor Red  = Blue
nextColor Blue = Red
```

Para implementar el algoritmo debemos completar la definición de la función aux:

```
biColored :: (Ord v) => Graph v -> Maybe (D.Dictionary v Color)
biColored g
 | null vs   = Just D.empty  -- empty graph is bipartite
 | otherwise = aux g D.empty (S.push (src,Red) S.empty)
 where
  vs  = vertices g
  src = head vs -- initial vertex

aux :: (Ord v) => Graph v -> D.Dictionary v Color -> S.Stack (v,Color) ->
                  Maybe (D.Dictionary v Color)
aux g dict stack
       | S.isEmpty stack       = Just dict
  // completad el resto de guardas
```

La función recursiva **aux** describirá el ciclo o bucle del algoritmo. El primer argumento corresponde al grafo, el segundo al diccionario de vértices y colores y el tercero a la pila de pares. Si el grafo no es bipartito la función aux devolverá **Nothing**. Si por el contrario es bipartito, la función aux devolverá **Just dic**.

> **NOTA**: cada alumno deberá descargar desde el campus virtual el código correspondiente a las estructuras de datos necesarias para la realización del ejercicio, y en particular el módulo BiPartite.hs que deberá completar. Para la comprobación de la corrección del ejercicio podrá utilizar los ejemplos del final del módulo.