

Assignment 1 — Control Flow Graph

Due Date — 1PM on 21 February 2013

1 Goal

Understand control flow graphs and how they can be generated from code.

2 Background

You are to construct a control-flow graph from the bytecode of a given Java class using the Bytecode Engineering Library (<http://jakarta.apache.org/bcel/>).

To illustrate, consider the following class C:

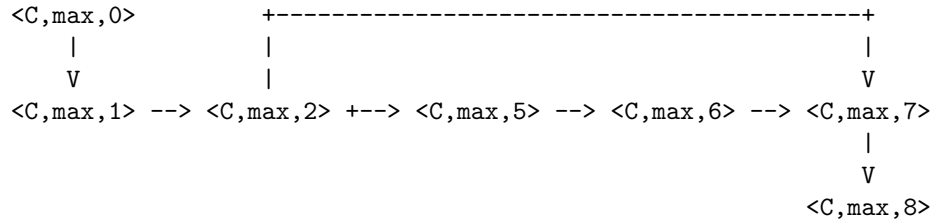
```
public class C {
    int max(int x, int y) {
        if (x < y) {
            return y;
        } else return x;
    }
}
```

You can run `javap -c` to see the corresponding bytecode in readable form.

```
Compiled from "C.java"
public class pset1.C extends java.lang.Object{
public pset1.C();
    Code:
        0:  aload_0
        1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
        4:  return
int max(int, int);
    Code:
        0:  iload_1
        1:  iload_2
        2:  if_icmpge     7
        5:  iload_2
        6:  ireturn
        7:  iload_1
        8:  ireturn
}
```

Ignoring method invocations and representing a node as the tuple `<class,method,position>` we get the following CFG.

```
<C,<init>,0> --> <C,<init>,1> --> <C,<init>,4>
```



3 Interface to Implement

```

class CFG {
    class Node {
        int position;
        Method method;
        JavaClass theclass;
    }
    public void addNode(int pos1, Method m1, JavaClass c1) {}
    public void addEdge(int pos1, Method m1, JavaClass c1,
                        int pos2, Method m2, JavaClass c2) {}
    public boolean isReachable(int pos1, Method m1, JavaClass c1,
                               int pos2, Method m2, JavaClass c2) {}
}

class CFGGenerator {
    public CFG createCFG(String className)
        throws ClassNotFoundException {}
    public static void main(String[] args) {}
};

```

4 Details

First implement the class `CFG` which represents a control flow graph. Implement the class `Node` with necessary helper methods (e.g. constructor, `isEqual`, `toString` etc.). Note that we are ignoring any labels that are traditionally annotated on nodes and edges of a CFG for simplification. Also ignore, for now, edges that correspond to method invocations, or `jsr[w]` or `*switch` bytecodes.

Use the adjacency list representation to store the directed graph. The `addNode` function should add a new node (but not if the node is already there).

The `addEdge` function should add a new edge keeping nodes and edges consistent. The `isReachable` function should return `true` only if both nodes are in the graph and control can flow from the first node to the second node.

In the `main` method, use the first command line argument as the class name to load. In `createCFG` use the following code as a hint to traverse methods and instructions of the loaded class file.

```
JavaClass jc = Repository.lookupClass(className);
ClassGen cg = new ClassGen(jc);
// loop on all methods of this class
for (Method m: cg.getMethods()) {
    // loop on all instructions in this method
    for (InstructionHandle ih:
        new MethodGen(m, cg.getClassName(), cg.getConstantPool())
        .getInstructionList().getInstructionHandles()) {
        int position = ih.getPosition();
        // at this point you have enough information to make a node
        Instruction inst = ih.getInstruction();
        // based on the instruction you may or may not have to add an edge
    }
}
```

5 What's coming up!

In the next assignment, you will extend this to support method invocations and automatic generation of JUnit tests. A better implementation of this assignment will prepare you best for the next assignment.