1. Consider the following program fragment, written in 3-address code.

```
    a <- 5  
b <- 10  
i <- 1  
goto L4
L1: if i < a goto L2
    c <- i * 2  
goto L3
L2: c <- i * 3  
L3: a <- a + c  
i <- i + 1  
L4: if i < b goto L1  
d <- a + b
```

(a) Identify the basic blocks and draw a control flow diagram using the basic blocks as the nodes.
(b) Number the basic blocks and draw a dominator tree for the procedure.
(c) Identify the dominance frontier of each block that contains one or more assignments.
(d) Put the procedure into SSA form (displayed as a control flow diagram).
(e) Put the procedure into Reference-Safe SSA form, as described in Section 2 of the assigned paper by Amme, et al.

2. Many program transformations used by compilers can be broadly classified into two categories.

1. **Control flow elimination** removes jumps, tests, and associated bookkeeping code, typically by replicating code. It typically makes the program faster, but larger (although subsequent application of other optimizations may shrink the program again). One example of such an optimization is loop unrolling, which usually increases program size, but removes the overhead of performing the loop test.

2. **Redundancy elimination** removes redundant computations. It typically make the program both smaller and faster. One example is hoisting invariant code out of loops.

Consider the performance results for the Swift compiler described in Table 2 of the assigned paper by Scales, et al. Based on these results, argue which category of transformations is more important for Swift. (Note: not all the optimizations described in this table fit into either category; you’ll have to decide which ones do.)