CS738: Advanced Compiler Optimizations
Mid Semester Examination, 2017-18 I

Max Time: 2 Hours | Max Marks: 100

NOTE:

- There are total **4** questions on **3 pages**
- No explanations will be provided. In case of a doubt, make suitable assumptions and justify.
- Presenting your answers properly is your responsibility. You lose credit if you can not present your ideas clearly, and in proper form. Please DO NOT come back for re-evaluation saying, "What I actually meant was . . . ".
- Be precise and write clearly. Remember that somebody has to read it to evaluate!

---

1. Consider a flow graph G with a unique entry node ENTRY that dominates all nodes of G. Prove that every node in G except ENTRY has a unique *immediate dominator*. The immediate dominator of a node n is the *closest strict dominator* of n. [10]

   - Every node except ENTRY has at least one strict dominator. Consider a node Z that has more than one strict dominators. Consider two such dominators X and Y. Then, it can be proved that either X dominates Y or Y dominates X (*proof below). Thus, it possible to find *least* element among all strict dominators of Z. This element is the desired immediate dominator.
   - * Consider a cycle free path from ENTRY→Z. Because both X and Y strictly dominate Z, they must occur on this path. WLOG, assume that X occurs before Y in this path. Thus the path is: Entry $\rightarrow$ X $\rightarrow$ Y $\rightarrow$ Z. We will prove that X must dominate Y.

     Assume the contrary (X does not dominate Y). Then, we have a path Entry $\Rightarrow Y$ free of X. But then, Entry $\Rightarrow$ Y $\rightarrow$ Z is a path to Z free of X. But that also contradicts the fact that X dominates Z.

2. The original definition of *Dominance Frontier* (df) is: A node m is in df(n) if

   (a) n dominates a predecessor of m in the flow graph, and

   (b) n does not strictly dominate m

   Dr Dominoz thinks the following modified definition of df(n) is equivalent as far as computation of SSA form is concerned: A node m is in df(n) if

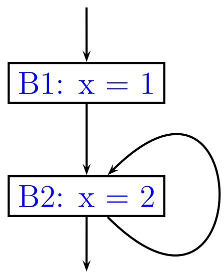
   (a) n dominates a predecessor of m in the flow graph, and

   (b) n does not dominate m

i.e., Dr Dominoz has dropped the term "strictly" from the definition.

You jobs is to either **prove that Dr Dominoz is right** or show that he is **wrong, by giving a counter example**. The proof must work for any arbitrary CFG, while the counter example must show an incorrect SSA form being generated for a CFG.[10]

Counter Example:



For Original definition:
Dominance Frontier of $B1 = \emptyset$
Dominance Frontier of $B2 = \{B2\}$
$Def(x) = \{B1, B2\}$
$DF^1 = \{B2\}$
so we will insert ϕ-statement in block $B2$.

For Modified definition:
Dominance Frontier of $B1 = \emptyset$
Dominance Frontier of $B2 = \emptyset$
$Def(x) = \{B1, B2\}$
$DF^1 = \emptyset$
so it will not insert any $\emptyset$ statement.

3. Consider the following extensions to our 3-address code language:

   - `x.lock`: "locks" a variable **x**
   - `x.unlock`: "unlocks" a variable **x**
   - `x.secureComp` does some "secure computation" on **x**

   In this language, a "secure computation" on a variable **x** is allowed only when it is locked (**x**.lock executed before **x**.secureComp, without an intervening **x**.unlock). We call such secureComp **safe**, otherwise it is **unsafe**.

   The language obviously contains basic constructs like *assignment* statements, *goto* statements, and conditionals (*if-goto*). Following semantic properties hold:

   - All variables are **unlocked** at the *entry*
   - **lock** and **unlock** operations are idempotent (Locking a locked variable is allowed, but it has no effect on the lock-status of the variable. Similarly, unlocking an unlocked variable is also allowed)

   Here are couple of sample programs, `PROGRAM-1` is valid and `PROGRAM-2` is invalid.

```
// PROGRAM-1
  c = 5
  n = 0
  t = c > 0
  if t goto L1
  n.lock
  n.secureComp
  goto L2

L1:
  c.lock
  c.secureComp
  d = e - 5;
  c.unlock
L2:
  n.lock

  n.secureComp

  n.unlock
  c.unlock
```

```
// PROGRAM-2
  c = 5
  n = 0
  t = c > 0
  if t goto L1
    n.lock
    c.secureComp // BAD, c not locked
    c.unlock
    goto L2
  L1:
    n.lock
    n.secureComp
    d  = e - 5;
    c.lock
  L2:
    n.secureComp // OK, n locked on
                 // all paths
    c.secureComp // BAD,  c may not
                 // be locked
    n.unlock
    c.unlock
```

   Design an **intraprocedural** data flow analysis framework to mark unsafe secure computations ("secureComp"). In particular,

   (a) Draw the lattice for the framework, and describe it briefly. [5]

   (b) Describe the meet operator ($\wedge$). [5]

(c) Describe intuitively the meaning of the top and the bottom elements. [5]

(d) Define the flow functions for statements. You do not need to list all types of statements, but use suitable representatives (for e.g. `x op y` to represent binary operators). [10]

(e) Is your framework *Forward* or *Backward*? Justify your answer. Also describe the *BoundaryInfo* (initialization information at the boundary of the flow graph). [2+3]

(a) Lattice (for each variable):

$\top$ = Lock

$\bot$ = Unlock

(b) meet:

$\wedge$	$\top$	$\bot$
$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

(c) $\top$ represents a variable is locked, $\bot$ represents it is unlocked (easy :-))

(d) Flow functions. (This is one of the many possible solutions)

i. S: x.lock

$$\begin{aligned} Gen(S) &= \{x \mapsto Lock\} \\ Kill(S) &= \{x \mapsto Lock, x \mapsto Unlock\} \end{aligned}$$

ii. S: x.unlock

$$\begin{aligned} Gen(S) &= \{x \mapsto Unlock\} \\ Kill(S) &= \{x \mapsto Lock, x \mapsto Unlock\} \end{aligned}$$

iii. S: Any Other Statement

$$\begin{aligned} Gen(S) &= \{\} \\ Kill(S) &= \{\} \end{aligned}$$

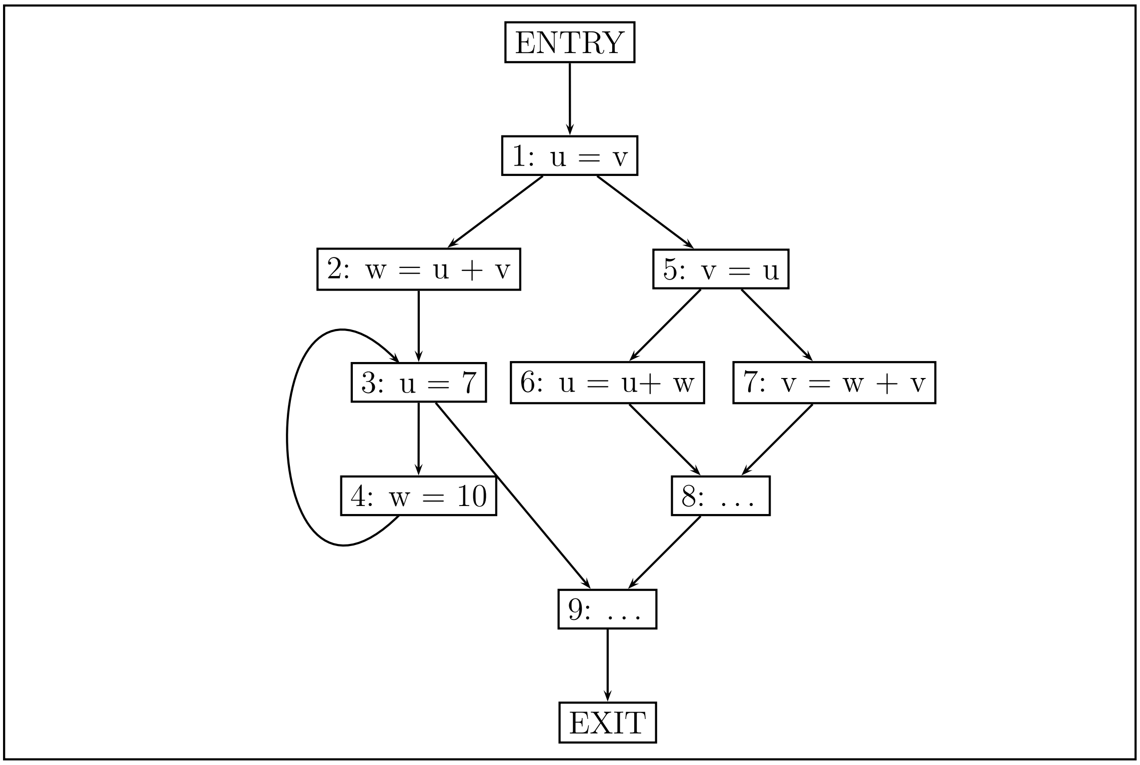
$$Out(S) = In(S) - Kill(s) \cup Gen(S)$$

For a statement S: x.secureComp, if $x \mapsto Lock \notin In(S)$ then computation is unsafe.

(e) My framework is forward (but yours could be backward, depends on the flow functions!). Out is computed in terms of In.

$$BoundaryInfo = \{x \mapsto Unlocked\} \; \forall x$$

4. Consider the following flow graph. Use statement numbers as basic block numbers.



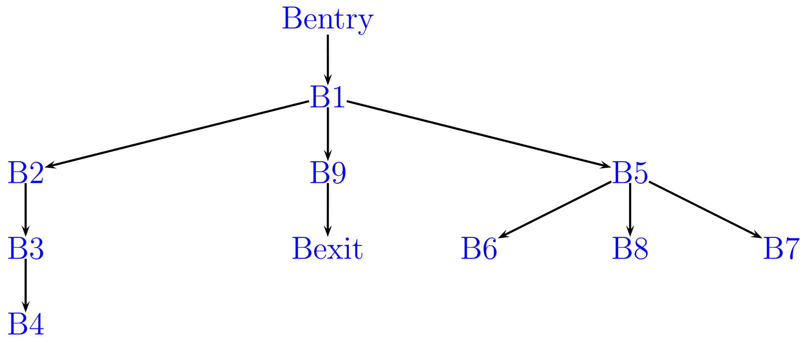
(a) Draw the dominator tree for the graph. [10]

(b) Calculate the dominance frontier for each block. [15]

(c) Calculate the iterated dominance frontiers for the nodes containing the definitions of u, v and w. Assume that ENTRY node contains implicit definitions of u, v, w as `undef`. [3*5 = 15]

(d) Convert the flow graph to minimal SSA form. [10]

Dominator Tree:

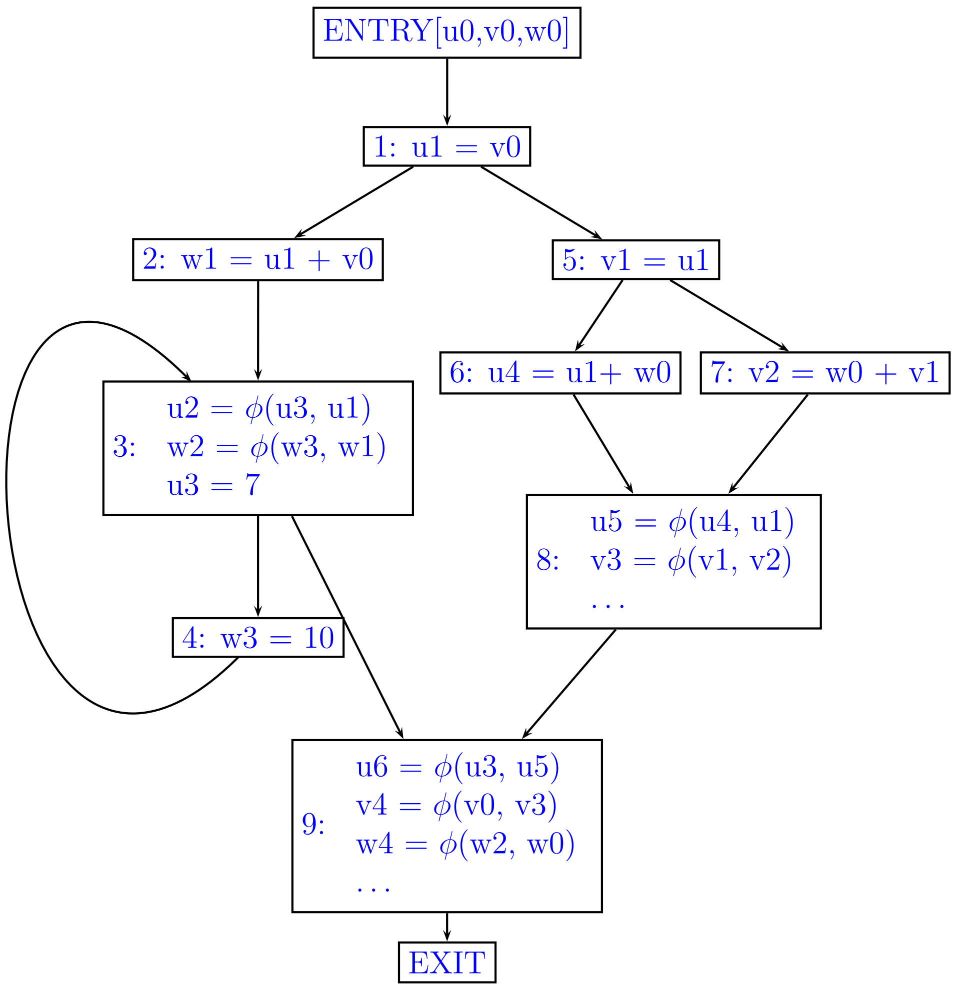


Dominance Frontier:

Node	DF
Entry	{}
1	{}
2	{ 9 }
3	{ 3, 9}
4	{3 }
5	{9}
6	{8}
7	{8}
8	{9}
9	{}
Exit	{}

ENTRY node (1) contains implicit definitions of x, y, z.

Var	Defs	Iterated Dom Frontier of Defs
u	0, 1, 3, 6	{ 3, 8, 9 }
v	0, 5, 7	{ 8, 9 }
w	0, 2, 4	{ 3, 9}



ENTRY[u0,v0,w0]
1: u1 = v0
2: w1 = u1 + v0
5: v1 = u1
6: u4 = u1+ w0
7: v2 = w0 + v1
3: u2 = φ(u3, u1)
w2 = φ(w3, w1)
u3 = 7
8: u5 = φ(u4, u1)
v3 = φ(v1, v2)
. . .
4: w3 = 10
9: u6 = φ(u3, u5)
v4 = φ(v0, v3)
w4 = φ(w2, w0)
. . .
EXIT

THE END