CS143 Written Assignment 4
Reference Solution

CS143 Course Staff

Problem 1. Programming in Assembly

Note that you can use `div x x` to create a constant 1 if `x` is not 0, and use `sub x x` to create a constant 0. The key to the rest of the work is to decide your loop invariant, and then write code so that the loop invariant is maintained in each iteration.

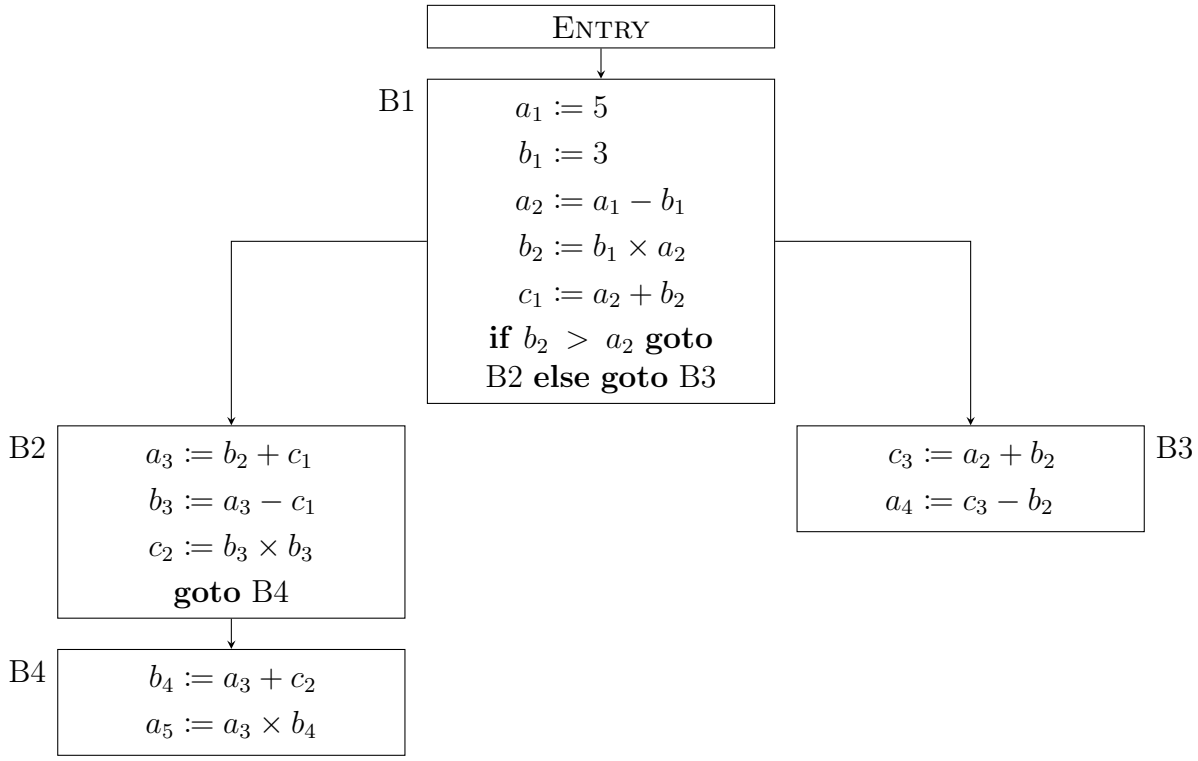
One of many possible solutions:

```
add y x     # x = n, y = n
div y y     # x = n, y = 1
sub x y     # x = n-1, y = 1
push x      # stk = [ n-1 ]
add y y     # y = 2
push y      # stk = [ n-1 | 2 ]
div y y     # y = 1
push y      # stk = [ n-1 | 2 | 1 ]
push y      # stk = [ n-1 | 2 | 1 | 1 ]
# loop start invariant:
# x: garbage, y: garbage  (garbage = we don't care what value it is)
# stk: [ n-i | i+1 | fact(i) | garbage ]
# next thing to print is fact(i) (on first iteration i=1)
#
pop y       # stk = [ n-i | i+1 | fact(i) ]
pop y       # y = fact(i), stk = [ n-i | i+1 ]
print y     # fact(i) is printed
pop x       # x = i+1, stk = [ n-i ]
mul y x     # y = fact(i+1)
push y      # stk = [ n-i | fact(i+1) ]
div y y     # y = 1
add x y     # x = i+2
swap        # stk = [ fact(i+1) | n-i ]
jz 12       # -> loop end if n-i == 0
pop y       # y = n-i
push x      # stk = [ fact(i+1) | i+2 ]
div x x     # x = 1
sub y x     # y = n-i-1
pop x       # x = i+2, stk = [ fact(i+1) ]
push y      # stk = [ fact(i+1) | n-i-1 ]
swap        # stk = [ n-i-1 | fact(i+1) ]
push x      # stk = [ n-i-1 | fact(i+1) | i+2 ]
swap        # stk = [ n-i-1 | i+2 | fact(i+1) ]
sub y y     # y = 0
push y      # stk = [ n-i-1 | i+2 | fact(i+1) | 0 ]
jz -22      # -> loop start, always
# loop end
halt
```

Problem 2. Single Static Assignment (SSA) Form

Task 2.A

Renaming each definition with a fresh subscript and rewriting each use to point to the most recent version yields:



Task 2.B

Recall that a use of variable x defined in block X requires every path from Entry to the use to pass through X. Let us check each variable:

Block B4 (predecessors B2 and B3; note B3→B4 edge exists):

- $y_1 \coloneqq x_1$ — x_1 defined in B1, which dominates B4. **Valid.**
- $y_2 \coloneqq x_2$ — x_2 defined in B2. But the path Entry→B1→B3→B4 reaches B4 without passing through B2. **Ill-defined.**
- $y_3 \coloneqq x_3$ — x_3 defined in B3. The path Entry→B1→B2→B4 reaches B4 without passing through B3. **Ill-defined.**

Block B5 (predecessor B3 only):

- $y_4 \coloneqq x_1$ — x_1 defined in B1, dominates B5. **Valid.**
- $y_5 \coloneqq x_3$ — x_3 defined in B3, and B3 is the unique predecessor of B5; B5 is reached only via B3. **Valid.**

- $y_6 \coloneqq x_4$ — x_4 defined in B4. But B5 is reached only via B3→B5, skipping B4 entirely. **Ill-defined.**

Block B6 (predecessors B4 and B5):

- $z_1 \coloneqq x_1$ — x_1 in B1 dominates B6. **Valid.**
- $z_2 \coloneqq x_2$ — x_2 in B2; the path Entry→B1→B3→B5→B6 misses B2. **Ill-defined.**
- $z_3 \coloneqq x_3$ — x_3 in B3; the path Entry→B1→B2→B4→B6 misses B3. **Ill-defined.**
- $z_4 \coloneqq x_4$ — x_4 in B4; the path Entry→B1→B3→B5→B6 misses B4. **Ill-defined.**
- $z_5 \coloneqq x_5$ — x_5 in B5; the path Entry→B1→B2→B4→B6 misses B5. **Ill-defined.**
- $z_6 \coloneqq x_6$ — defined in B6 itself, before its use. **Valid.**

Answer: The ill-defined variables are $y_2, y_3, y_6, z_2, z_3, z_4, z_5$ (7 in total).

Task 2.C

For a ϕ-node in B6 of the form $\phi(\text{B4} : u, \text{B5} : v)$ to be well-defined, we need: u defined in some block that dominates every path Entry→B4, *and* v defined in some block that dominates every path Entry→B5.

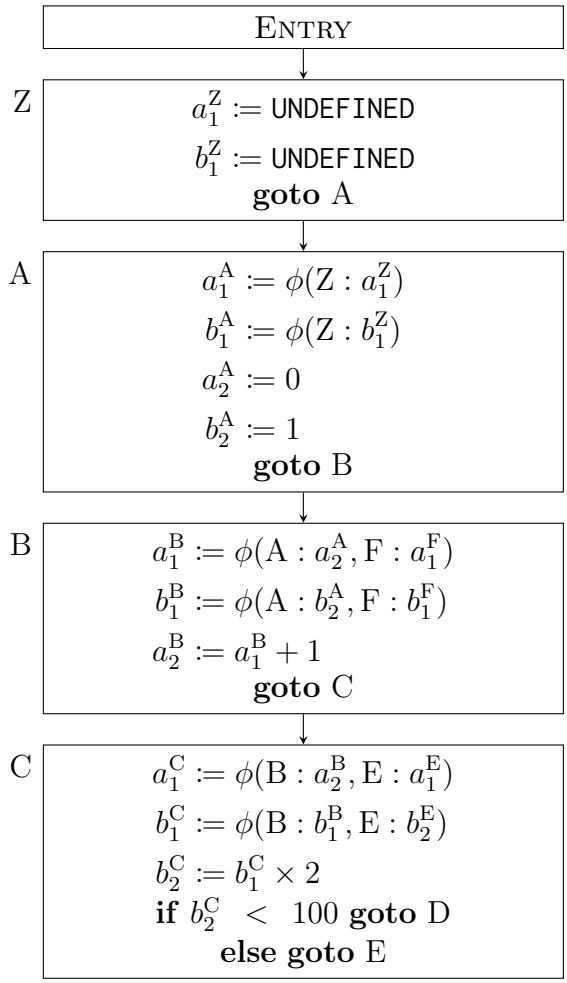
Recall B4's predecessors are B2 and B3, while B5's only predecessor is B3.

- $y_1 = \phi(\text{B4} : x_1, \text{B5} : x_1)$: x_1 in B1 dominates both B4 and B5. **Valid.**
- $y_2 = \phi(\text{B4} : x_2, \text{B5} : x_3)$: B4 slot needs x_2 to be defined on every path to B4. But Entry→B1→B3→B4 skips B2. **Ill-defined.**
- $y_3 = \phi(\text{B4} : x_4, \text{B5} : x_5)$: x_4 is defined in B4 itself, and the B4 slot fires when control flows "from B4" — since execution within B4 reaches its end before transferring to B6, x_4 is defined by then. The B5 slot uses x_5 defined in B5, and similarly x_5 is defined within B5 before B5 transfers control to B6. **Valid.**
- $y_4 = \phi(\text{B4} : x_3, \text{B5} : x_2)$: B4 slot wants x_3 on every path to B4. But Entry→B1→B2→B4 skips B3. **Ill-defined.**
- $y_5 = \phi(\text{B4} : x_1, \text{B5} : x_3)$: x_1 in B1 dominates B4; x_3 in B3, and B5's only path is via B3, so x_3 is defined on every path to B5. **Valid.**
- $y_6 = \phi(\text{B4} : x_5, \text{B5} : x_4)$: B4 slot wants x_5 from B5, but B5 is not on any path Entry→B4 — the slot is ill-defined. (Likewise the B5 slot wants x_4 from B4, equally bad.) **Ill-defined.**
- $y_7 = \phi(\text{B4} : x_3, \text{B5} : x_3)$: B5 slot is fine (as in y_5). But the B4 slot wants x_3 on every path to B4, and Entry→B1→B2→B4 skips B3. **Ill-defined.**

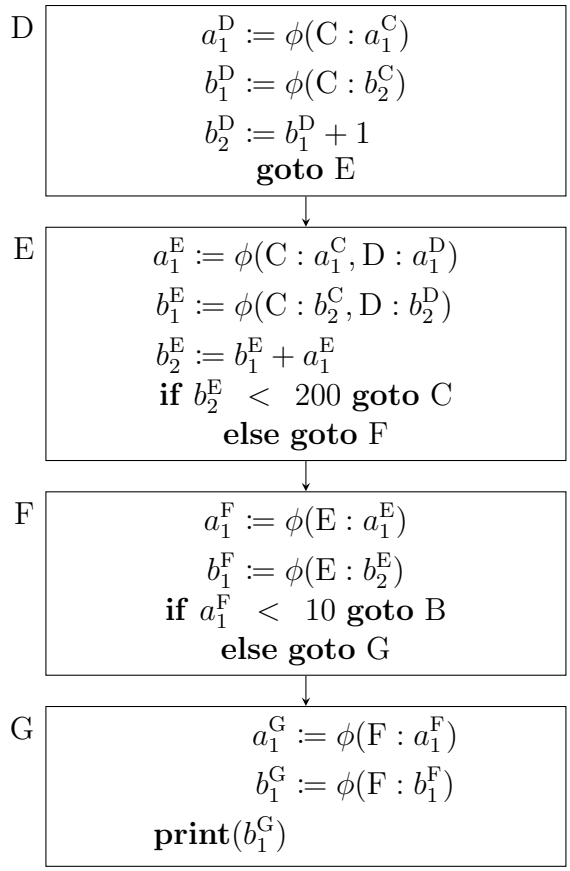
Answer: The ill-defined ϕ-nodes are y_2, y_4, y_6, y_7 (4 in total). The well-defined ones are y_1, y_3, y_5.

Task 2.D

Following the algorithm produces:



(continued on next page)



(The back-edges C→D, C→E, E→C, F→B are implicit in the IR but omitted from the diagram for clarity.)

Task 2.E

(i) Each variable is defined exactly once. Step 2 of the algorithm creates exactly one ϕ-assignment $x_1^B \coloneqq \phi(\cdots)$ for each (block, variable) pair, all using fresh names. Step 3 walks through each non-ϕ assignment in source order and gives the left-hand side a fresh subscript, also unique. Steps 2 and 3 are the only sources of definitions, and by construction every left-hand-side name is fresh, so each variable appears on the left of exactly one assignment in the resulting IR.

(ii) Every variable use satisfies def-before-use. There are two cases.

- *Use within the same block as the definition.* Step 3 processes statements in source order. When it rewrites a right-hand-side occurrence of x, it substitutes the most recent version of x already defined in this block. Such a version always exists: at worst it is the ϕ-node x_1^B inserted at the top of the block in step 2, which by construction precedes every non-ϕ statement.

- *Use in a block B' whose definition is upstream.* Every use of x in B' refers either to $x_1^{B'}$ (set by the start-of-block ϕ) or to a fresh version created by step 3 inside B'. Both are defined inside B' itself, so the issue reduces to whether the ϕ-node arguments are themselves well-defined — which is point (iii).

Note that the new entry block Z initializes every variable to `UNDEFINED`, so even on the very first iteration through a block, the start-of-block ϕ-node has a valid value flowing in from Z (or from any earlier predecessor along that path).

(iii) Every ϕ-node satisfies def-before-use. A ϕ-node $x_1^B \coloneqq \phi(P_1 : v_1, \ldots, P_n : v_n)$ requires: for each i, if v_i is defined in block X_i, then every path Entry$\to P_i$ passes through X_i. By step 4 of the algorithm, v_i is chosen to be the last version of x at the end of block P_i. There are two sub-cases:

- If x is assigned somewhere in P_i, then v_i is the variable defined by the latest such assignment in P_i, which lies inside P_i itself. Every path that ends at P_i trivially passes through P_i (it is the path's last block), so the def-before-use property holds.
- If x is not assigned anywhere in P_i, then $v_i = x_1^{P_i}$, the variable defined by the ϕ-node at the top of P_i. Again v_i is defined in P_i, and the same argument applies.

Since the new entry block Z provides an `UNDEFINED` initial assignment for every variable, the chain of ϕ-nodes always terminates — there is no "hole" in the chain.

Task 2.F

Iterating dead code elimination on the IR from Task 2.D:

- a_1^{G} and b_1^{G}: the only statement in G is **print**(b_1^{G}), which uses b_1^{G} but not a_1^{G}. So a_1^{G} is dead and removed.
- a_1^{A} and b_1^{A}: in A, the assignments are $a_2^{\mathrm{A}} \coloneqq 0$ and $b_2^{\mathrm{A}} \coloneqq 1$. Neither RHS uses a_1^{A} or b_1^{A}. The B-block phi for a uses a_2^{A} (not a_1^{A}), and similarly the B-block phi for b uses b_2^{A}. So a_1^{A} and b_1^{A} are dead and removed.
- Once a_1^{A} and b_1^{A} are gone, their RHS sources a_1^{Z} and b_1^{Z} have no remaining uses, so they too are removed.
- Every other variable is used: b_1^{G} is used by print; a_1^{F} is used by the if-test; b_1^{F} feeds b_1^{G}; a_1^{E} is used by $b_2^{\mathrm{E}} \coloneqq b_1^{\mathrm{E}} + a_1^{\mathrm{E}}$; etc.

Answer: The dead variables are $a_1^{\mathrm{Z}}, b_1^{\mathrm{Z}}, a_1^{\mathrm{A}}, b_1^{\mathrm{A}}, a_1^{\mathrm{G}}$ (5 in total).

Task 2.G

We iteratively eliminate ϕ-nodes whose value is forced to coincide with another existing variable. After each elimination we substitute the eliminated variable everywhere it was used, which can expose further eliminations.

Round 1: single-argument ϕ-nodes. A ϕ-node with only one predecessor argument can always be replaced by that argument.

- In D: $a_1^{\mathrm{D}} \coloneqq \phi(\mathrm{C} : a_1^{\mathrm{C}})$ becomes $a_1^{\mathrm{D}} = a_1^{\mathrm{C}}$. Replace a_1^{D} with a_1^{C} in E's ϕ.

- In D: $b_1^{\mathrm{D}} \coloneqq \phi(\mathrm{C} : b_2^{\mathrm{C}})$ becomes $b_1^{\mathrm{D}} = b_2^{\mathrm{C}}$. Replace b_1^{D} on the RHS of $b_2^{\mathrm{D}} \coloneqq b_1^{\mathrm{D}} + 1$, yielding $b_2^{\mathrm{D}} \coloneqq b_2^{\mathrm{C}} + 1$.
- In F: $a_1^{\mathrm{F}} \coloneqq \phi(\mathrm{E} : a_1^{\mathrm{E}})$ becomes $a_1^{\mathrm{F}} = a_1^{\mathrm{E}}$. Replace a_1^{F} in B's ϕ for a.
- In F: $b_1^{\mathrm{F}} \coloneqq \phi(\mathrm{E} : b_2^{\mathrm{E}})$ becomes $b_1^{\mathrm{F}} = b_2^{\mathrm{E}}$. Replace b_1^{F} in B's ϕ for b.
- In G: $b_1^{\mathrm{G}} \coloneqq \phi(\mathrm{F} : b_1^{\mathrm{F}})$ becomes $b_1^{\mathrm{G}} = b_1^{\mathrm{F}} = b_2^{\mathrm{E}}$. Replace in the print statement.

Round 2: equal-argument ϕ-nodes. After Round 1, E's ϕ for a has become:

$$a_1^{\mathrm{E}} \coloneqq \phi(\mathrm{C} : a_1^{\mathrm{C}}, \mathrm{D} : a_1^{\mathrm{C}})$$

Both arguments equal a_1^{C}, so the ϕ-node always produces a_1^{C}. Eliminate a_1^{E} and substitute a_1^{C} wherever it appears. This affects B's ϕ for a (which after Round 1 reads $a_1^{\mathrm{B}} \coloneqq \phi(\mathrm{A} : a_2^{\mathrm{A}}, \mathrm{F} : a_1^{\mathrm{E}})$ — now becomes $\phi(\mathrm{A} : a_2^{\mathrm{A}}, \mathrm{F} : a_1^{\mathrm{C}})$) and the use $b_2^{\mathrm{E}} \coloneqq b_1^{\mathrm{E}} + a_1^{\mathrm{E}}$, which now reads $b_2^{\mathrm{E}} \coloneqq b_1^{\mathrm{E}} + a_1^{\mathrm{C}}$.

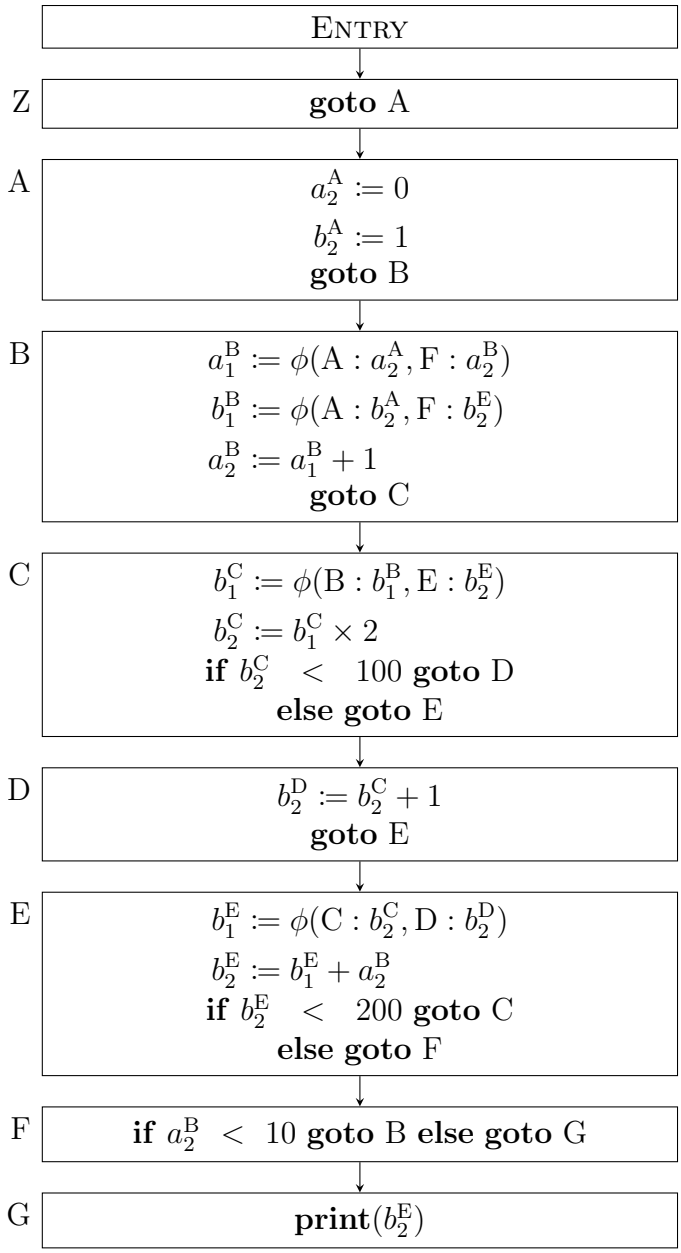
Round 3: trivial self-referential ϕ-nodes. C's ϕ for a now reads:

$$a_1^{\mathrm{C}} \coloneqq \phi(\mathrm{B} : a_2^{\mathrm{B}}, \mathrm{E} : a_1^{\mathrm{C}})$$

The E-slot is just a_1^{C} itself — the value flowing back around the inner loop has not been changed inside the loop body. So a_1^{C} is forced to equal a_2^{B} on every iteration. Eliminate a_1^{C} and substitute a_2^{B} wherever it appears. This updates B's ϕ for a (F-slot now a_2^{B}) and the use $b_2^{\mathrm{E}} \coloneqq b_1^{\mathrm{E}} + a_2^{\mathrm{B}}$. F's branch test uses $a_1^{\mathrm{F}} = a_1^{\mathrm{E}} = a_1^{\mathrm{C}} = a_2^{\mathrm{B}}$, so it becomes a test on a_2^{B}.

No further eliminations are possible: the remaining ϕ-nodes have multiple, distinct arguments that depend on the loop iteration.

Resulting simplified SSA IR (exactly four ϕ-nodes):



The four surviving ϕ-nodes are:

1. $a_1^B := \phi(A : a_2^A, F : a_2^B)$ — needed because the value of a entering B differs on the first entry (from A, value 0) versus subsequent iterations of the outer loop (from F, value a_2^B).

2. $b_1^B := \phi(A : b_2^A, F : b_2^E)$ — similarly, b's value entering B differs on first entry (1) versus outer-loop back-edge.

3. $b_1^C := \phi(B : b_1^B, E : b_2^E)$ — b's value entering C differs depending on whether control comes from B (first iteration of inner loop) or from E (back-edge of inner loop).

4. $b_1^E := \phi(C : b_2^C, D : b_2^D)$ — b's value entering E differs depending on whether D was taken (the conditional increment) or skipped.

1 Problem 3

Consider the following assembly-like pseudo-code, using 6 temporaries (abstract registers) a to f:

```
1     a := b + d
2     b := a + d
3     d := a - b
4     c := d + d
5     if c > 100:
6         c := c + d
7     else:
8         d := 1
9     e := d - c
10    f := e - c
```

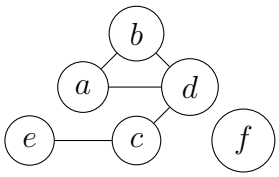
1. At each program point, list the variables that are live. Note that **b** and **d** are inputs for the given code and **f** is a live value on exit.

   Solution:

```
1                           {b,d}
2         a := b + d        {a,d}
3         b := a + d        {a,b}
4         d := a - b        {d}
5         c := d + d        {c,d}
6         if c > 100:
7                           {c,d}
8             c := c + d    {c,d}
9         else:
10                          {c}
11            d := 1        {c,d}
12        e := d - c        {c,e}
13        f := e - c        {f}
```

2. Draw the register interference graph between temporaries in the above program as described in class.

   Solution:

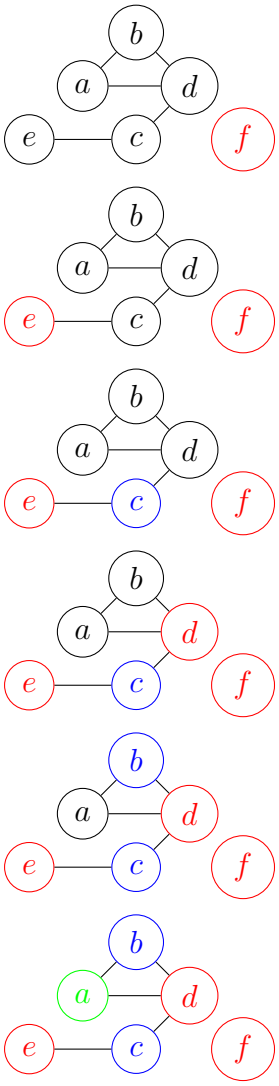


3. Provide a lower bound on the number of registers required by the program induced from the interference graph. Can you explain why?

   Solution: 3. It is mainly because there is a triangle in the graph.

4. Using the algorithm described in class, provide a coloring of the graph in part(b). The number of colors used should be your lower bound in part (c). Provide the final k-colored graph (you may use the tikz package to typeset it or simply embed an image), along with the order in which the algorithm colors the nodes.

   Solution:



5. Based on your coloring, write down a mapping from temporaries to registers (labeled r1, r2, etc.).

   Solution:

```
1        a: r1
2        b: r2
3        c: r2
4        d: r3
5        e: r3
6        f: r3
```