CS6013 – MODERN COMPILERS
END SEMESTER EXAM

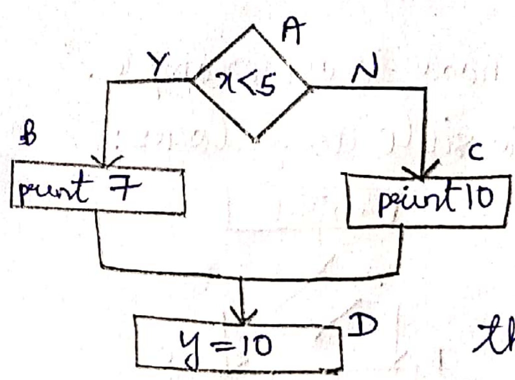
ARJUN BHARAT
CS17B006
03/05/2020

Question I:

a) Examples

i)
```
if(x<5)
    print 7;
else
    print 10;
y=10;
```

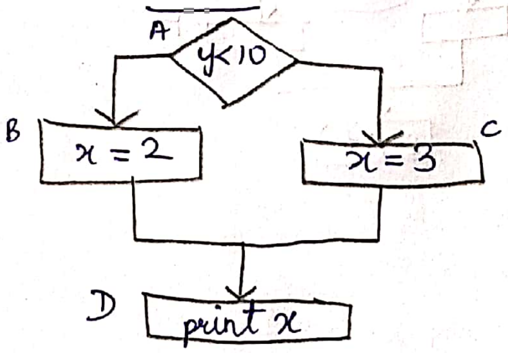
CFG



Here, D has immediate dominator to be A that differs from the predecessors B and C.

ii) Code
```
if (y<10)
    x=2;
else
    x = 3;
print(x);
```

CFG



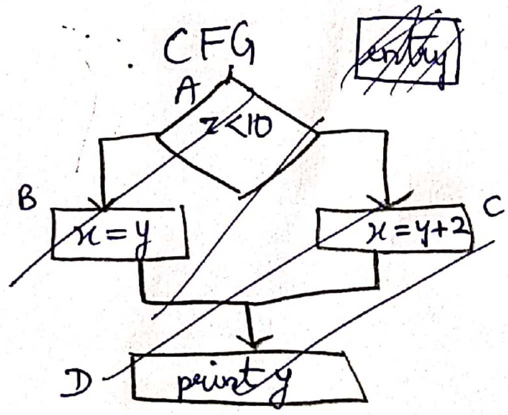
D has predecessors B and C, but neither of them dominate it.

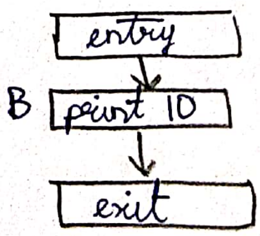
iii) Code :

~~if (z<10)~~
~~x=y~~
~~else~~
~~x = y+2~~
~~print y~~

```
void foo()
{
  print 10
}
```

CFG





The strict dominator of B is only the entry block which is also the immediate dominator.

b)

i) One leaf, which is the exit node.
(Assumption: for every return statement, the value is stored in a temporary -ret and we jump to exit node where the return happens after restoring stack pointer etc.

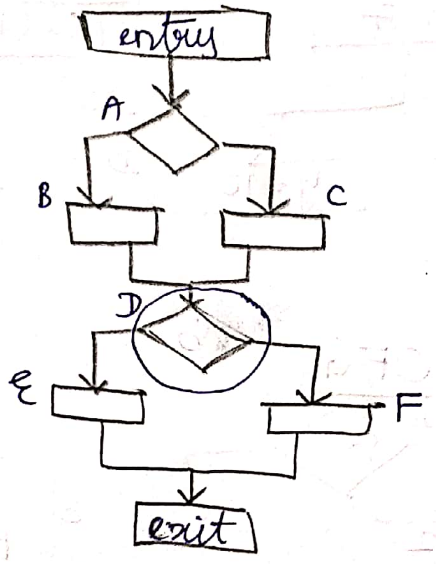
tion 2

void fn;

ii) $F_{e/false} \sqcap (F_{e/true} \circ F_t)$ (We can assume $F_e = F_{e/true} = F_{e/false}$

iii) Yes, the exit node is an example.

iv) Yes, this is possible as follows:



v) False

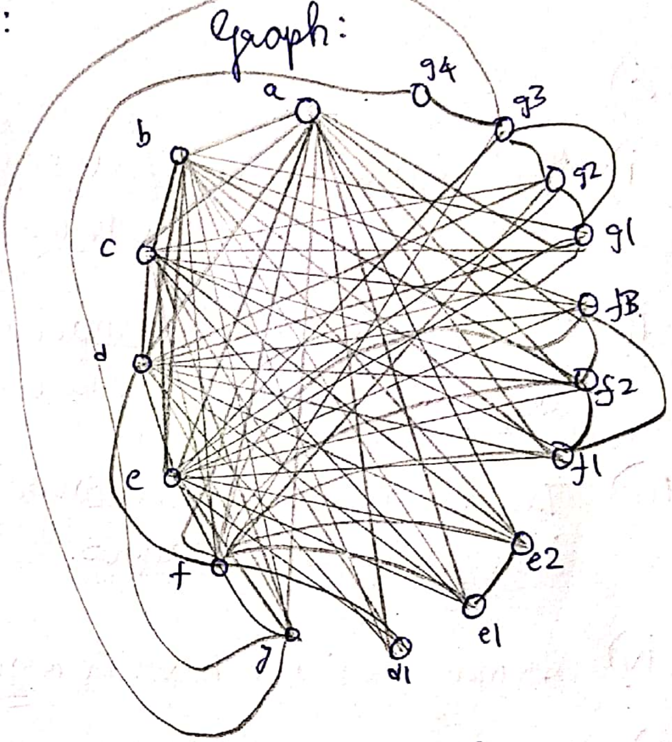
vi) False.

stion 2

```
void foo()
{
1  int a,b,c,d1,d,e1,e2,e,f1,f2,f3,f,g1,g2,g3,g4,g;
2  a=1;
3  b=a+2;
4  c=a+b;
5  d1= a+b;
6  d=d1+c;
7  e1= a-b;
8  e2=c-d;
9  e= e1+e2;
10 f1= a*b;
11 f2 = c* d;
12 f3 = f1+f2;
13 f = f3+e;
14 g1= a^b;
15 g2= c^d;
16 g3= e^f;
17 g4= g1+g2;
18 g = g3+g4;
19 print(g);
}
```

Liveness ranges:

a: [2,14]
b: [3,14]
c: [4,15]
d: [6,15]
e: [9,16]
f: [13,16]
g: [18,19]
d1: [5,6]
~~d~~ e1: [7,9]
e2: [8,9]
~~e3: E~~
f1: [10,12]
f2: [11,12]
f3: [12,13]
g1: [14,17]
g2: [15,17]
g3: [16,18]
g4: [17,18]



Clearly, (a, b, c, d, e, f) form a clique which needs 6 colours at least. Also, d1 has 6 neighbours at least (a, b, c, d, e, f) and hence needs a 7th colour.

Therefore, the graph is not 6 colourable.

b) Argument: because a is live out at S and not defined at S, a must be live in at S.
∴ a is live out from R which defines it, and from Q as well.

We see that $IN(P) = \{\hat{a}, b\}$, $OUT(P) = \{a, b, c\}$

$IN(Q) = \{a, c\}$ $OUT(Q) = \{a, b\}$

$IN(R) = \{b, c\}$ $OUT(R) = \{a, b\}$

Assumption: c may or may not be live out at R and Q, & it is not clear if c has a use in S. Similarly, d may or may not be live out at Q as we do not know if S uses d.

c).

i) This allows us to spill only if necessary while colouring, thus reducing the number of spills.

ii) The live ranges will have changed due to load on use, store on def policy because of a new temporary being added.

iii) The said move-node cannot be coalesced due to their interfering live ranges.

iv) ~~As many available machine registers there are (pre-coloured)~~. None.

v) Yes, provided live ranges do not interfere.

vi) No, as they always interfere.

Question3

Consider the following classes as part of the program:

```
class A
{
public void foo()
  { System.out.println(10);
  }
public void bar(B b)
  {
      b.foo(); //S3
  }
public void start()
  {
    B b = new B(); //L4
    this.bar(b); //S1
    b = new C(); //L5
    this.bar(b); //S2
  }
}
class B extends A
{
public void foo()
  { System.out.println(20);
  }
}
class C extends B
{
public void foo()
  { System.out.println(30);
  }
}}
```

Flow sensitivity

```
public static void main()
{ A a = new A(); //L1
  B b = new B(); //L2
  C c = new C(); //L3
  a = b;
  b = c;
  a.foo(); //S
}
```

Flow sensitive analysis would allow inlining at site S because a points to L_2 only, but flow insensitive analysis would say that a points to $\{L_1, L_2\}$ or $\{L_1, L_2, L_3\}$ and hence no inlining.

Context sensitivity

```
public static void main()
{
    A a = new A();
    a.start();
}
```

If we used context sensitive analysis, we would analyse call sites S_1 and S_2 with b having points to sets $\{L_4\}$, $\{L_5\}$ respectively. This would allow inlining of foo at S_3.

Without context sensitivity, we would use a conservative points-to set of b at S_1, S_2, meaning that we cannot inline S_3's call to foo.

b)

i) Assuming n allocation statements, 2^n.

ii) ~~we~~ for $\forall$ locals $\in$ function, $p(a) = \{\}$

iii) The stack p, ~~because~~ assuming multi layered object fields are not very common. (otherwise, heap would have more)

iv) The stack p.

Question 4

Question 4

a) i) Loop inversion

→ Transforms a while loop into a do-while loop under an if statement.

L1:
if(!cond) goto L2
goto L3
L2:
//S body
goto L1
L3:

```
while (cond)          →     if (cond) //can be removed if taken.
{                           { do
  S                           {
}                               S
                              } while (cond)
                            }
```

→ feasible under all circumstances

→ Profitable only when the loop body is taken for sure, because then on removal of if we would have half the number of branch statements.

ii) Loop unrolling:

→ unravels loop iterations and reduces loop duration

```
for (i=0; i<n; i++)    →   n' = (n/k) * k , some k
    S                      for (i=0; i<kn'; i++)
                           { [S, i++] // k-1 times
                             S
                           }
```

(can be done even for while loop)

→ Feasible only if loop is countable: i.e loop has an invariant condition.

→ Profitable when branch instructions are expensive (branch and increment are cut down) and big icache.

Loop tiling: Splits a for loop into 2 nested ones.

```
for (i=0; i<n; i++)   →   for (i1=0; i1<n/Bi; i1++)
    S                     { for (i2=0; i2<Bi; i2++)
                              { S // i = B i1 + i2
                          } }
```

→ ~~Sat~~ feasible when loop invariant conditions are present, and S has no loop dependency.

→ ~~E~~ Profitable → when B_i is related to cache line size (which allows quick loads from Cache). Jump instruction should be cheap.

b)

i)

```
for (i=0; i<n; i++)
{
    y = a[2*i]+5;
    a[3*i+1] = x;
}
```

we need $2i_2 - 3i_1 = 1$ ⇒ by GCD test, exists.

in fact, $i_2=2$, $i_1=1$ is a case where true dependency occurs.

ii)

Question 5

a) Minimal SSA inserts a φ-node for all globals where applicable, whereas pruned SSA inserts a φ node only where a global is live in there.

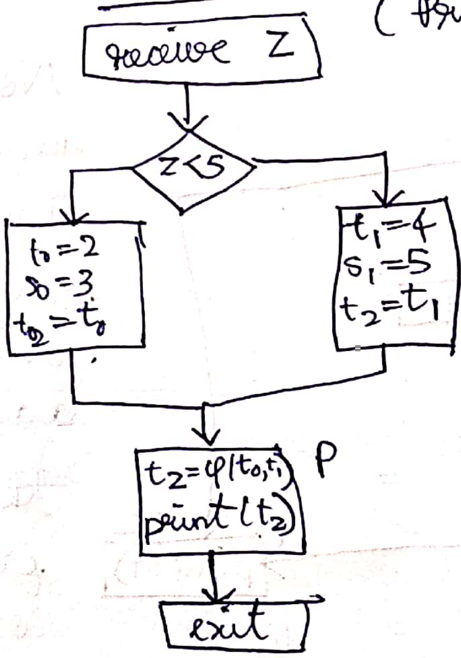
b) Minimal SSA is more conservative, and is hence suboptimal. It will not be incorrect.

c) assume s and t are both globals.

```
void bar (int z)
{ if(z<5)
    { t=2;
      s=3;
    }
  else
    { t=4;
      s=5;
    }
  print(t);
}
```

CFG

(Pruned SSA)



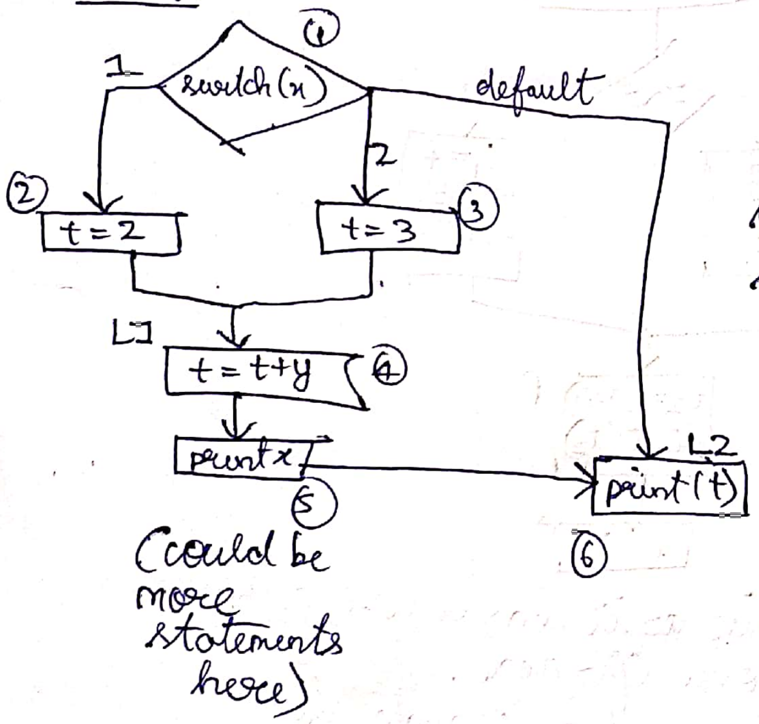
at block P, only t is live-in as it has a use, and pruned SSA adds only one φ-node.
Minimal SSA would have added a φ node for s as well.

Scanned with CamScanner

d) Consider the code:

```
switch (x)
{
  case 1: {t=2; goto L1;} break;
  case 2: {t=3; goto L1;} break;
  default: goto L2; break;
}
  L1:  t = t+y;
       print(x);
  L2:  print(t);
```

CFG



Here, $S = \{2,3,4\}$ and
$DF(S) = \{4, 5\}$

Now see that at ⑥, we need a ϕ node for t because there is a path to it with defs for t.
but $DF^+(S) = \{4,5,6\}$ which ensures that we add enough ϕ nodes.

Hence, Iterated dominance frontier preserves correctness.

Question 6

b) False

d) False

e) True

f) True

g) True

h) False