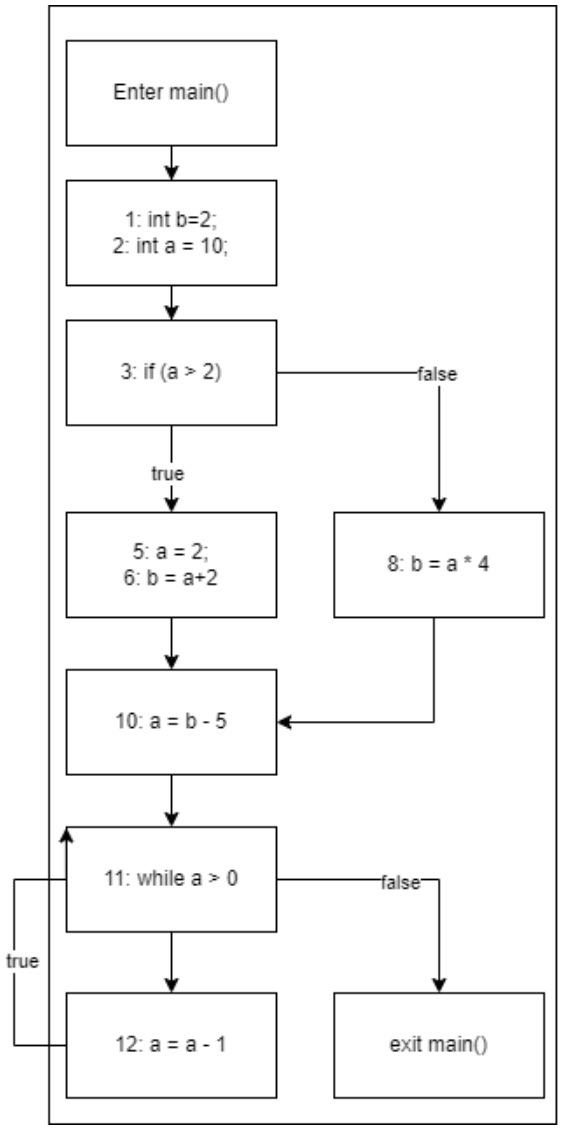
Neha Maddali

HW3

1. CFG:



2. int b = 2; - this defines and initializes the variable b
   int a = 10; - this defines and initializes the variable a
   a = 2; - this redefines the variable a inside the if block
   b = a + 2; - this redefines the variable b inside the if block
   b = a * 4; - this redefines the variable b inside the else block
   a = b – 5; - this redefines the variable a after the conditional statement
   a = a – 1; - this redefines the variable a inside the while loop
3. Use of b at line 2: int b = 2;
   Use of a at line 4: if(a > 2){
   Use of a and b at line 6 (inside the if block): b = a + 2;
   Use of a and b at line 8 (inside the else block): b = a * 4;

Use of a and b at line 10: a = b – 5;
Use of a at line 11: while (a > 0){
Use of a at line 12 (inside the while loop): a = a -1;

4. GEN:
   GEN[2] = {b = 2}
   GEN[3] = {a = 10}
   GEN[5] = {a = 2}
   GEN[6] = {b = a + 2}
   GEN[8] = {b = a * 4}
   GEN[10] = {a = b -5}
   GEN[12] = {a = a -1}

   KILL:
   KILL[2] = {} (no previous definitions of b to kill)
   KILL[3] = {} (no previous definitions of a to kill)
   KILL[5] = {a = 10} (kills a = 10 and redefines to a = 2)
   KILL[6] = {b = 2} (kills b = 2 and redefines to b = a + 2)
   KILL[8] = {b = a + 2} (kills b = a + 2 and redefines to b = a * 4)
   KILL[10] = {a = 10} (kills a = 10 and redefines to a = b -5)
   KILL[12] = {a = b - 5} (kills a = b – 5 and redefines to a = a -1)

   IN[B] = ∪ (OUT(p)) for all p predecessors of B
   IN[4] = OUT[3]
   IN[5] = OUT[4]
   IN[7] = OUT[4]
   IN[9] = OUT[5]
   IN[11] = OUT[10]
   IN[12] = OUT[11]

   OUT[B] = GEN(B) ∪ (IN(B) – KILL(B))
5. Initial OUT[B] bitvectors:
   Entry: [0, 0, 0, 0, 0, 0, 0] (bitvector size = the total number of definitions)
   Node 2: [0, 1, 0, 0, 0, 0, 0] (bit 1 is set to 1: *int b = 2* is a GEN)
   Node 3: [0, 1, 1, 0, 0, 0, 0] (bit 2 is set to 1: *int a =10* are GENs)
   Node 4: [0, 1, 1, 0, 0, 0, 0] (no additional GENs within this block)
   Node 5: [0, 1, 1, 1, 0, 0, 0] (bit 3 is set to 1: *a = 2* is a GEN)
   Node 6: [0, 1, 1, 1, 1, 0, 0] (bit 4 is set to 1: *b = a + 2* is a GEN)
   Node 7: [0, 1, 1, 1, 1, 0, 0] (no additional GENs within this block)
   Node 8: [0, 1, 1, 1, 1, 1, 0] (bit 5 is set to 1: *b = a * 4* is a GEN)
   Node 9: [0, 1, 1, 1, 1, 1, 0] (no additional GENs within this block)
   Node 10: [0, 1, 1, 1, 1, 1, 1] (bit 6 is set to 1: *a = b – 5* is a GEN)
   Node 11: [0, 1, 1, 1, 1, 1, 1] (no additional GENs within this block)
   Node 12: [0, 1, 1, 1, 1, 1, 1] (*a = a – 1* is a GEN)
   Exit: [0, 1, 1, 1, 1, 1, 1]
6. W = {Entry, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, Exit}
   Node 12: OUT[12]' = GEN[12] ∪ (IN[12] – KILL[12])

OUT[12]' = {a = a -1} ∪ ([0, 1, 1, 1, 1, 1, 1] – {a = b – 5})
OUT[12]' = [0, 1, 1, 1, 1, 1, 1] U ([0, 1, 1, 1, 1, 1, 1] - [0, 1, 1, 1, 1, 1, 1])
OUT[12]' = [0, 1, 1, 1, 1, 1, 1]

Node 11: OUT[11]' = GEN[11] ∪ (IN[11] – KILL[11])
OUT[11]' = {} ∪ ([0, 1, 1, 1, 1, 1, 1] – {})
OUT[11]' = [0, 1, 1, 1, 1, 1, 1]

Node 10: OUT[10]' = GEN[10] ∪ (IN[10] – KILL[10])
OUT[10]' = {a = b -5} ∪ ([0, 1, 1, 1, 1, 1, 1] – {a = 10})
OUT[10]' = [0, 1, 1, 1, 1, 1, 1] ∪ ([0, 1, 1, 1, 1, 1, 1] - [0, 1, 1, 1, 1, 1, 1])
OUT[10]' = [0, 1, 1, 1, 1, 1, 1]

Node 9: OUT[9]' = GEN[9] ∪ (IN[9] – KILL[9])
OUT[9]' = {} ∪ ([0, 1, 1, 1, 0, 0, 0] – {}
OUT[9]' = [0, 1, 1, 1, 0, 0, 0]
Not the same as OUT[9], so add the successor of Node 9 to W which is 10

Node 8: OUT[8]' = GEN[8] ∪ (IN[8] – KILL[8])
OUT[8]' = {b = a * 4} ∪ ([0, 1, 1, 1, 0, 0, 0] – {b = a + 2})
OUT[8]' = [0, 1, 1, 1, 1, 1, 0] ∪ ([0, 1, 1, 1, 0, 0, 0] - [0, 1, 1, 1, 1, 1, 0, 0])
OUT[8]' = [0, 1, 1, 1, 1, 1, 0]

Node 7: OUT[7]' = GEN[7] ∪ (IN[7] – KILL[7])
OUT[7]' = {} ∪ ([0, 1, 1, 0, 0, 0, 0] – {})
OUT[7]' = [0, 1, 1, 0, 0, 0, 0]
Not the same as OUT[7], so add the successor of Node 7 to W which is 8

Node 6: OUT[6]' = GEN[6] ∪ (IN[6] – KILL[6])
OUT[6]' = {b = a + 2} ∪ ([0, 1, 1, 0, 0, 0, 0] - {b = 2})
OUT[6]' = [0, 1, 1, 1, 1, 0, 0] ∪ ([0, 1, 1, 0, 0, 0, 0] - [0, 1, 1, 1, 1, 0, 0])
OUT[6]' = [0, 1, 1, 1, 1, 0, 0]

Continue this until the W is empty.

At program point line 12, the definitions in line 6, 8, 10, 12 reach this use.

7. The reaching definition problem is a forward dataflow problem. In forward dataflow analysis, we start with initial information at the entry point of a program and propagate that information through the CFG to compute the dataflow values at various program points, eventually reaching the exit point. In the case of reaching definitions, we start with definitions that reach the entry point and propagate them forward to compute the reaching definitions at different program points.

   The reaching definition is a must problem. In a must dataflow problem, if a fact holds at a particular program point, it is guaranteed to hold for all executions of the program. In the context of reaching definitions, if a definition is determined to reach a certain program point, it means that this definition must reach that point for all possible program executions; it's a definite and certain analysis.