Data-Flow Analysis Exercises

(2) **Problem 2**

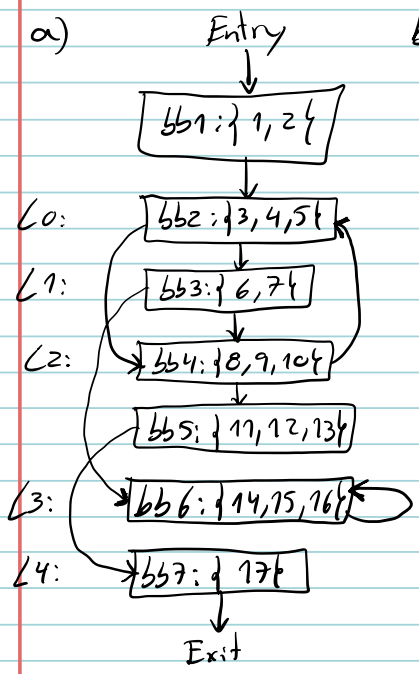
Consider the code shown below, in three-address instruction format.

```
01          a = 1
02          b = 2
03 L0:      c = a + b
04          d = c - a
05          if c < d goto L2
06 L1:      d = b + d
07          if d < 1 goto L3
08 L2:      b = a + b
09          e = c - a
10          if e = 0 goto L0
11          a = b + d
12          b = a - d
13          goto L4
14 L3:      d = a + b
15          e = e + 1
16          goto L3
17 L4:      return
```

For the code shown above, determine the following:

a) The basic blocks of instructions.
b) The control-flow graph (CFG)
c) For each variable, its corresponding *du*-chain.
d) The live variables at the end of each basic block. You do not need to determine the live variables before and after each basic block and justify your answer for the value presented for the basic block containing instructions at line 6 and 7.
e) Is the Live-Variable analysis a forward or backwards data-flow analysis problem? Why? What does guarantee its termination when formulated as an iterative data-flow analysis problem?

a)



b) a: {d1, u3, u4, u8, u9, u14}; {d11, u12}
b: {d2, u3, u6, u8, u14}; {d8, u3, u6, u8, u11, u14}
c: {d3, u4, u5, u9}
e: {d9, u10, u15}, {d15, u15}

c) bb1: {a, b}
bb2: {a, b, c, d, e}
bb3: {a, b, c, d, e}
bb4: {a, b, d, e}
bb5: { }
bb6: {a, b, e}

e)

e) The live variable analysis is a backward data-flow problem as we propagate the information about a future use of a variable backward to specific points of the program. If there is a definition at a specific point backward the solution kills all other uses and resets the information associated with that variable. As with many other iterative formulations of data-flow analysis problems termination is guaranteed by the fact that the lattice, in this case the set of variables, has finite cardinality or length. The flow-function, in this case set-union is monotonic.

4 Problem 4

```
01      a = 1
02      b = 2
03 L0:  c = a + b
04      d = c - a
05      if c < d goto L2
06 L1:  d = b + d
07      if d < 1 goto L3
08 L2:  b = a + b
09      e = c – a
10      if e == 0 goto L0
11      a = b + d
12      b = a – d
13      goto L4
14 L3:  d = a + b
15      c = c + 1
16      goto L1
17 L4:  return
```

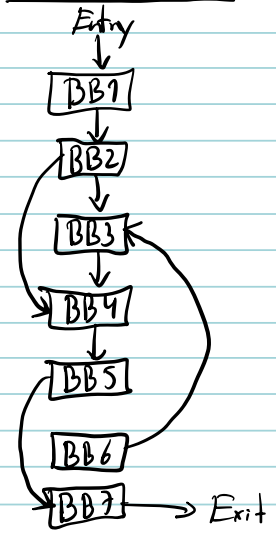
For the code shown above, determine the following:

a) The basic blocks of instructions and the control-flow graph (CFG).
b) The live variables at the end of each basic block. You do not need to determine the Live Variables before and after each basic block but justify your answer for the value presented for the basic block containing instructions at line 6 and 7.
c) Is the Live-Variables analysis a forward or backward data-flow analysis problem? Why and what does guarantee its termination when formulated as a data-flow analysis iterative problem?

a) Basic Blocks Instructions

BB1: {1, 2}
BB2: {3, 4, 5}
BB3: {6, 7}
BB4: {8, 9, 10}
BB5: {11, 12, 13}
BB6: {14, 15, 16}
BB7: {17}

Control-Flow Graph



b) BB1: {a, b}
BB2: {a, b, c, d, e}
BB3: {a, b, c, d, e}
BB4: {a, b, c, d, e}
BB5: {} → é retornado a seguir
BB6: {a, b, c, d, e}

justification: In the block BB6, the var "e" is defined and exists a goto L1, as a result, we have the var e live in the block BB3.

c) The live variable analysis is a backward data-flow problem because we propagate the information about an future use of a variable backward to specific points of the program. If there is a definition at a specific point backward the solution kills all other uses and resets the info associated with that variable. As with many other iterative formulations of data-flow analysis problems termination is guaranteed by the fact that the lattice, in this case the set of variables, has finite cardinality or length. The flow-function, in this case set-union is monotonic.

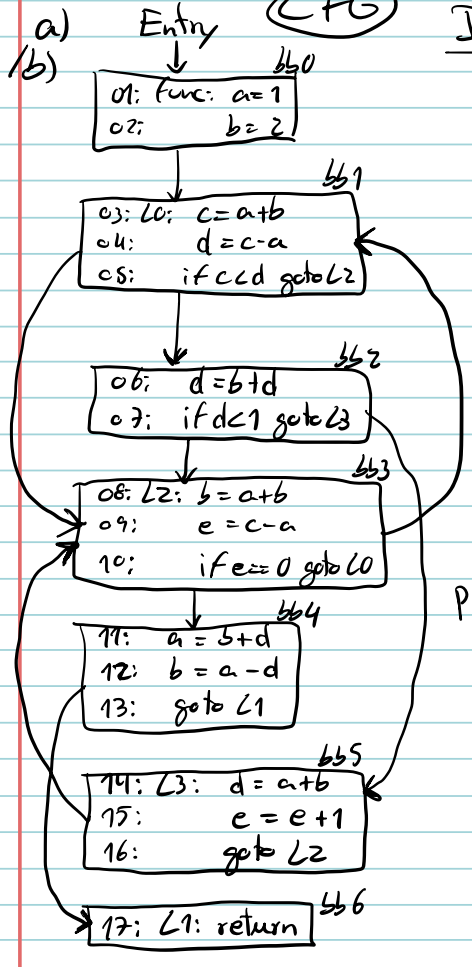
(5) **Problem 5**

For the code shown on the left, determine the following:

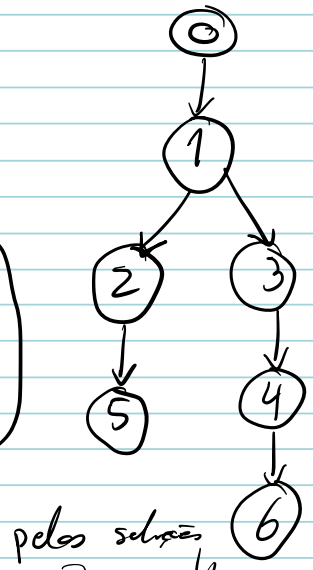
```
01 func:  a = 1
02        b = 2
03 L0:    c = a + b
04        d = c - a
05        if c < d goto L2
06        d = b + d
07        if d < 1 goto L3
08 L2:    b = a + b
09        e = c - a
10        if e == 0 goto L0
11        a = b + d
12        b = a - d
13        goto L1
14 L3:    d = a + b
15        e = e + 1
16        goto L2
17 L1:    return
```

a) The basic blocks of instructions identifying the instructions that constitutes each basic block. Clearly identify the leader instruction of each basic block.
b) The control-flow graph (CFG) and its dominator tree.
c) For each variable, determine its *use-def* chains.
d) The set of available expressions at the beginning of each basic block. You do not need to compute the DFA solution at each step of the iterative formulation but rather argue that your solution is correct by explaining the specific value of Available-Expressions DFA problem.
e) Is the Available Expressions DFA a forward or backwards data-flow analysis problem and why?
f) What does guarantee its termination when formulated as a data-flow analysis iterative problem?

a) b) CFG



Dominator Tree



pelas setas
3 → 4
5 → 3

c) a: {d1, u3, u4, u8, u9, u14}, {d11, u12}
b: {d2, u3, u6, u8, u14}, {d8, u3, u6, u8}, {d6, u14}
c: {d3, u4, u5, u9}
d: {d4, u5, u6, u11, u12}, {d6, u7, u11, u12}, {d14, u11, u12}
e: {d9, u10, u15}, {d15}

separar por causa do if.

6

Problem 6

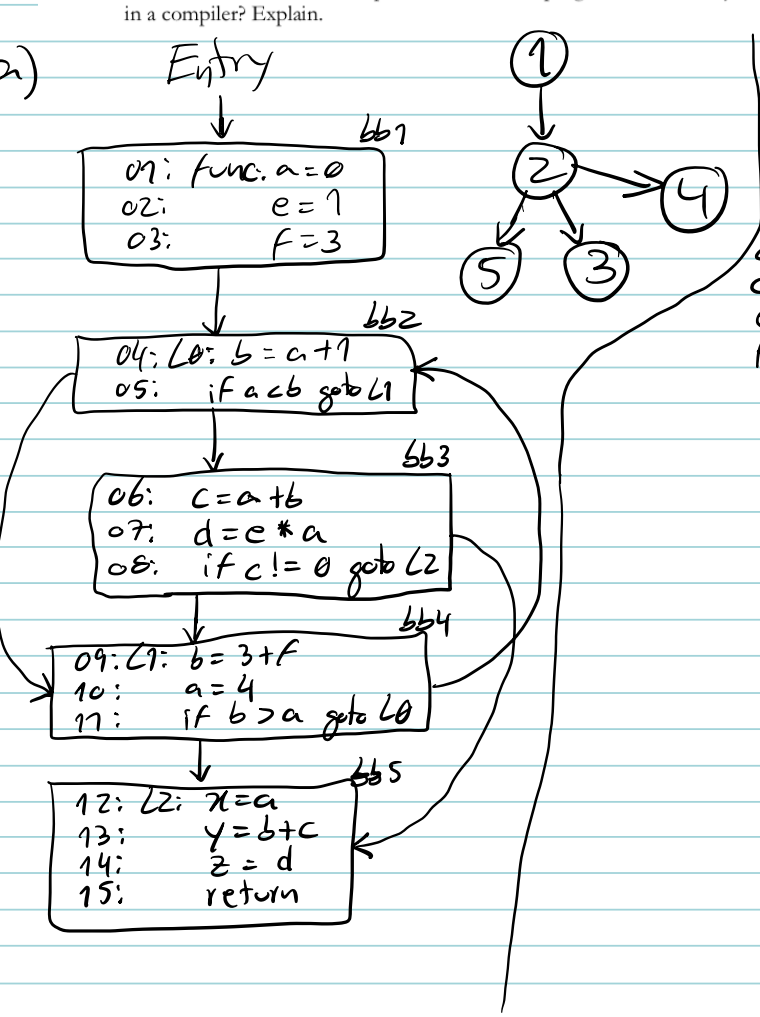
Consider the three-address instruction code below:

```
01 func:   a = 0
02         e = 1
03         f = 3
04 L0:     b = a + 1
05         if a < b goto L1
06         c = a + b
07         d = e * a
08         if c != 0 goto L2
09 L1:     b = 3 + f
10         a = 4
11         if b > a goto L0
12 L2:     x = a
13         y = b + c
14         z = d
15         return
```

For this code determine the following:

a. The set of basic blocks and the corresponding control-flow graph and dominator tree.
b. The DU-chains and Reaching Definitions for all variables `a`, `b`, `c`, `d`, `e` and `f`. Ignore in this analysis the variables `x`, `y` and `z`.
c. Determine a new representation of this code using SSA form representation.
d. Are there any opportunities for constant propagation? How would you detect them from either the information on dominance or via the SSA representation of the program? Which do you think would be the easiest to implement in a compiler? Explain.

a)



b)

DU-Chains

a: {d1, u4, u5, u6, u7, u12}
{d10, u11, u4, u5, u6, u7, u12}
b: {d4, u5, u6, u13}
{d9, u11, u13}
c: {d6, u8, u13}
d: {d7, u14}
e: {d2, u7}
f: {d3, u9}