ECE 468
Problem Set 5: Dataflow analysis

1. Show the results of running a *liveness* analysis on the following piece of code: For each line of code, show which definitions reach that line of code by indicating the line number the definition occurred in.

```
    1: x = 4;
    2: y = 7;
L1  3: if (x > c) goto L4
    4:   if (y > 3) goto L2
    5:     a = x + 1;
    6:     b = a + x;
    7:     goto L3
L2  8:     a = a + x;
    9:     b = x + 1;
L3 10:   y = a + b;
   11:   goto L1;
L4 12: halt
```

   Answer: We will show the results of the available expression analysis below; the liveness analysis proceeds similarly, but with GEN and KILL determined by the variables used and defined in a statement, and the analysis run backwards.

2. Show the results of running an *available expression* analysis on the code, by indicating which expressions are available at each instruction.

   Answer: First, we determine the successors and predecessors for each instruction. We can do this by building the control flow graph, but I'll just present the same information in a table:

Instruction	**Predecessors**	**Successors**
1	None	2
2	1	3
3	2, 11	4, 12
4	3	5, 8
5	4	6
6	5	7
7	6	10
8	4	9
9	8	10
10	7, 9	11
11	10	3
12	3	None

We then build the GEN and KILL sets for each instruction. Note that the expressions in the program are: `4`, `7`, `x > c`, `y > 3`, `x + 1`, `a + x` and `a + b`.

Instruction	**GEN**	**KILL**
1	`4`	`x + 1`, `a + x`
2	`7`	`y > 3`
3	`x > c`	None
4	`y > 3`	None
5	`x + 1`	`a + x`, `a + b`
6	`a + x`	`a + b`
7	None	None
8	None	`a + x`, `a + b`
9	`x + 1`	`a + b`
10	`a + b`	`y > 3`
11	None	None
12	None	None

Recall that this is a forward analysis. We initialize all of the available expression in- and out-sets to $\top$, except for statement 1, whose in-set is initialized to $\emptyset$.

Instruction	IN	OUT
1	$\emptyset$	$\top$
2	$\top$	$\top$
3	$\top$	$\top$
4	$\top$	$\top$
5	$\top$	$\top$
6	$\top$	$\top$
7	$\top$	$\top$
8	$\top$	$\top$
9	$\top$	$\top$
10	$\top$	$\top$
11	$\top$	$\top$
12	$\top$	$\top$

We then begin updating the in and out sets, starting from statement 1. Statement 1's out-set gets updated to `4`. Because the out-set has changed, all of statement 1's *successors* need to be processed. This causes statement 2's in-set to be changed to `4`, and its out-set to be changed to {`4`, `7`}. This causes statement 2's successors to be updated. Statement 3 has two predecessors: 2 and 11, so its in-set is updated to {`4`, `7`} $\cap\top$, leading to the following situation:

Instruction	IN	OUT
1	$\emptyset$	`4`
2	`4`	`4`, `7`
3	`4`, `7`	`4`, `7`, `x > c`
4	$\top$	$\top$
5	$\top$	$\top$
6	$\top$	$\top$
7	$\top$	$\top$
8	$\top$	$\top$
9	$\top$	$\top$
10	$\top$	$\top$
11	$\top$	$\top$
12	$\top$	$\top$

We must now process all successors of statement 3, which means we must process 4 *and* 12. We first process 12, updating its in-set and its out-set. 12 has no successors so does not trigger any new processing. We then process 4:

Instruction	IN	OUT
1	$\emptyset$	`4`
2	`4`	`4, 7`
3	`4, 7`	`4, 7, x > c`
4	`4, 7, x > c`	`4, 7, x > c, y > 3`
5	$\top$	$\top$
6	$\top$	$\top$
7	$\top$	$\top$
8	$\top$	$\top$
9	$\top$	$\top$
10	$\top$	$\top$
11	$\top$	$\top$
12	`4, 7, x > c`	`4, 7, x > c`

Our worklist now contains the successors of 4, statements 5 and 8. We process 5 first, which triggers our processing of 6, which triggers our processing of 7:

Instruction	IN	OUT
1	$\emptyset$	`4`
2	`4`	`4, 7`
3	`4, 7`	`4, 7, x > c`
4	`4, 7, x > c`	`4, 7, x > c, y > 3`
5	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3, x + 1`
6	`4, 7, x > c, y > 3, x + 1`	`4, 7, x > c, y > 3, x + 1, a + x`
7	`4, 7, x > c, y > 3, x + 1, a + x`	`4, 7, x > c, y > 3, x + 1, a + x`
8	$\top$	$\top$
9	$\top$	$\top$
10	$\top$	$\top$
11	$\top$	$\top$
12	`4, 7, x > c`	`4, 7, x > c`

At this point, 8 and 10 (the successor of 7) are on the worklist. Let us turn to statement 8. Recall that 8's in-set is determined by 4's out-set. Processing 8 triggers processing 9, leading to the following situation:

Instruction	IN	OUT
1	$\emptyset$	`4`
2	`4`	`4, 7`
3	`4, 7`	`4, 7, x > c`
4	`4, 7, x > c`	`4, 7, x > c, y > 3`
5	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3, x + 1`
6	`4, 7, x > c, y > 3, x + 1`	`4, 7, x > c, y > 3, x + 1, a + x`
7	`4, 7, x > c, y > 3, x + 1, a + x`	`4, 7, x > c, y > 3, x + 1, a + x`
8	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3`
9	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3, x + 1`
10	$\top$	$\top$
11	$\top$	$\top$
12	`4, 7, x > c`	`4, 7, x > c`

Now we are left with only statement 10 on the worklist. We compute its in-set by intersecting 7 and 9's out-sets, giving us: {`4`, `7`, `x > c`, `y > 3`, `x + 1`}. We then process 10 and 11:

Instruction	IN	OUT
1	$\emptyset$	`4`
2	`4`	`4, 7`
3	`4, 7`	`4, 7, x > c`
4	`4, 7, x > c`	`4, 7, x > c, y > 3`
5	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3, x + 1`
6	`4, 7, x > c, y > 3, x + 1`	`4, 7, x > c, y > 3, x + 1, a + x`
7	`4, 7, x > c, y > 3, x + 1, a + x`	`4, 7, x > c, y > 3, x + 1, a + x`
8	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3`
9	`4, 7, x > c, y > 3`	`4, 7, x > c, y > 3, x + 1`
10	`4, 7, x > c, y > 3, x + 1`	`4, 7, x > c, x + 1, a + b`
11	`4, 7, x > c, x + 1, a + b`	`4, 7, x > c, x + 1, a + b`
12	`4, 7, x > c`	`4, 7, x > c`

Changing 11's outset puts statement 3 on our worklist, which we must process, propagating data to 12 and 4. After processing 3, 4 and 12, we have:

Instruction	IN	OUT
1	∅	4
2	4	4, 7
3	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
4	4, 7, x > c, x + 1, a + b	4, 7, x > c, y > 3, x + 1, a + b
5	4, 7, x > c, y > 3	4, 7, x > c, y > 3, x + 1
6	4, 7, x > c, y > 3, x + 1	4, 7, x > c, y > 3, x + 1, a + x
7	4, 7, x > c, y > 3, x + 1, a + x	4, 7, x > c, y > 3, x + 1, a + x
8	4, 7, x > c, y > 3	4, 7, x > c, y > 3
9	4, 7, x > c, y > 3	4, 7, x > c, y > 3, x + 1
10	4, 7, x > c, y > 3, x + 1	4, 7, x > c, x + 1, a + b
11	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
12	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b

with 5 and 8 on the worklist. We process 5:

Instruction	IN	OUT
1	∅	4
2	4	4, 7
3	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
4	4, 7, x > c, x + 1, a + b	4, 7, x > c, y > 3, x + 1, a + b
5	4, 7, x > c, y > 3, x + 1, a + b	4, 7, x > c, y > 3, x + 1
6	4, 7, x > c, y > 3, x + 1	4, 7, x > c, y > 3, x + 1, a + x
7	4, 7, x > c, y > 3, x + 1, a + x	4, 7, x > c, y > 3, x + 1, a + x
8	4, 7, x > c, y > 3	4, 7, x > c, y > 3
9	4, 7, x > c, y > 3	4, 7, x > c, y > 3, x + 1
10	4, 7, x > c, y > 3, x + 1	4, 7, x > c, x + 1, a + b
11	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
12	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b

Note that even though 5's in-set changed, its out-set *did not change*. This means that we do not have to process 6. We now turn to statement 8:

Instruction	IN	OUT
1	∅	4
2	4	4, 7
3	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
4	4, 7, x > c, x + 1, a + b	4, 7, x > c, y > 3, x + 1, a + b
5	4, 7, x > c, y > 3, x + 1, a + b	4, 7, x > c, y > 3, x + 1
6	4, 7, x > c, y > 3, x + 1	4, 7, x > c, y > 3, x + 1, a + x
7	4, 7, x > c, y > 3, x + 1, a + x	4, 7, x > c, y > 3, x + 1, a + x
8	4, 7, x > c, y > 3, x + 1, a + b	4, 7, x > c, y > 3, x + 1
9	4, 7, x > c, y > 3	4, 7, x > c, y > 3, x + 1
10	4, 7, x > c, y > 3, x + 1	4, 7, x > c, x + 1, a + b
11	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
12	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b

Here, 8's out-set changes, so we must process 9:

Instruction	IN	OUT
1	∅	4
2	4	4, 7
3	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
4	4, 7, x > c, x + 1, a + b	4, 7, x > c, y > 3, x + 1, a + b
5	4, 7, x > c, y > 3, x + 1, a + b	4, 7, x > c, y > 3, x + 1
6	4, 7, x > c, y > 3, x + 1	4, 7, x > c, y > 3, x + 1, a + x
7	4, 7, x > c, y > 3, x + 1, a + x	4, 7, x > c, y > 3, x + 1, a + x
8	4, 7, x > c, y > 3, x + 1, a + b	4, 7, x > c, y > 3, x + 1
9	4, 7, x > c, y > 3, x + 1	4, 7, x > c, y > 3, x + 1
10	4, 7, x > c, y > 3, x + 1	4, 7, x > c, x + 1, a + b
11	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b
12	4, 7, x > c, x + 1, a + b	4, 7, x > c, x + 1, a + b

At this point, 9's out-set does not change. Our worklist is thus empty, and the analysis is complete.

3. In this problem, your goal is to develop a dataflow analysis to find *leaking* values. A common error in programs is forgetting to free a variable that has been `malloc`ed. Write an analysis that can tell, at the *end* of a function, whether any allocations during the function *may not have been freed* by the end of the function. Hint: think about this analysis' relationship to reaching definitions.

   (a) This is a forward bitvector analysis, where each object of interest in the analysis takes on the value 0 or 1. a) What are the objects of interest in this analysis? b) What does it mean for an object to have value 0? c) What does it mean for the object to have value 1?
   Answer: The objects we care about are any variables that have been `malloc`ed;

this is a finite set. If an object has a value 1, it may have been allocated (or, in other words, may not have been freed); if it has a value 0, it is definitely not allocated (so either never allocated or definitely freed). Note that at the beginning of the program, all variables are set to 0: they clearly have never been allocated.

(b) Which statements GEN information for this analysis? Which statements KILL information? Give the GEN and KILL sets for the relevant statements.
Answer: `malloc(x)` GENs x (makes it 1), while `free(x)` KILLs x (makes it 0).

(c) Define IN and OUT for this analysis (don't forget which direction your analysis is running: if you're running forward, IN should be defined in terms of a statement's predecessors, and OUT should be defined in terms of IN. If you're running backwards, OUT is defined in terms of a statement's successors, and IN is defined in terms of OUT). Don't forget to think about how your analysis should behave at merge statements.
Answer: This is a forward analysis, and we care whether a variables has been allocated along *any* path without being freed. Thus, at merge statements we should union together the sets of variables. This will use the same dataflow equations as reaching definitions.

(d) (Tricky question): this problem can be reformulated as a backwards analysis, instead. At each `malloc`, we want to know whether the variable being allocated here will definitely be freed. Answer the previous three questions for this new analysis. Which of the two analyses will give you better results (or will they be the same)?
Answer: This analysis is basically the opposite of the previous one. A value of 1 means that a variable has definitely been freed, a value of 0 means that a variable may not have been freed. `free(x)` GENs x. `malloc(x)` KILLs x (because even if x will be freed after this `malloc`, earlier allocations of x need to be freed before we get to this `malloc`). We initialize all variables to 0 at the *end* of the function. The analysis runs backwards; at merge points, we want to be sure that the variable will be freed along *all* paths leading from this statement, so we merge using intersection. This analysis will use the same dataflow equations as very-busy expressions.