Problems: Dataflow Analysis

1. This question is about points-to analysis for a language like C that allows pointers, pointers-to-pointers, etc. The ultimate goal of the analysis is to discover, for each node n in the program's control-flow graph and for each variable `x`, what variables `x` might point to at n. Thus, the final answer at node n will be a set of pairs, such as

$$\{\langle \mathtt{x}, \mathtt{y}\rangle, \langle \mathtt{x}, \mathtt{z}\rangle, \langle \mathtt{y}, \mathtt{a}\rangle\},$$

meaning that, at n, (i) `x` might point to `y`, (ii) `x` might point to `z`, and (iii) `y` might point to `a`.

Assume that the programming language includes assignment statements, if-then-else statements, and while loops (but no function calls, no pointer arithmetic, and no heap-allocated storage). Also assume that the assignment statements have been normalized so that there is at most one pointer dereference per statement (e.g., the following statements are okay: `x = &y;` `x = y;` `x = *y;` `*x = y;`, but `*x = *y;` and `**x = y` are not).

(a) Define, as a path problem, a *flow-sensitive* points-to analysis for the language by providing the following:

   i. The domain of dataflow answers (see above), as well as the domain of dataflow functions.
   ii. The extend operation
   iii. The combine operation used to combine dataflow facts at convergence points in the control-flow graph
   iv. The dataflow functions for each kind of edge in the control-flow graph (i.e., the four different kinds of assignments, and the outgoing true-edges and false-edges at a branch node), as well as the initial value to use at the program's start node.

(b) The following program fragment constructs a linked list of unspecified size:

```
1 :  head = NULL;             // head = &null
2 :  while(*){
3 :     temp = malloc();      // temp = &malloc$3
4 :     *temp = head;
5 :     head = temp;
6 :  }
7 :
```

State the

   i. global (combine-over-all-paths) view
   ii. local (equational) view, by giving the set of equations for your program

(c) Solve the equations that you gave in Question **??**.

(d) Why is it interesting to know whether the dataflow functions for a flow-sensitive analysis are distributive?

(e) Are the dataflow functions that you wrote for Question **??** all distributive? (Justify your answer.)

2. This question concerns interprocedural reaching-definitions analysis and the Sharir-Pnueli call-strings approach (augmented with the technique for propagating information from an exit node to a corresponding return node that involves "peeking-back at the dataflow information available at the call-node").
   (a) Give an example program for which tracking call-strings of length up to 2 gives more precise results than tracking call-strings of length up to 1. In particular, show the dataflow information obtained at each call-node, entry-node, exit-node, and return-node.
   (b) For your example with call-strings 1, explain the point at which a call-string-saturation step takes place that is the root cause of the loss of precision. Then explain how that causes less precise information to be propagated from one or more callees to one or more callers.

3. In this question, assume that we have a very simple programming language whose control-flow graphs have just read statements, assignment statements, conditions, and print statements; and there are no pointers, arrays, structs, etc.

   A path that includes no definition of variable x is called an *x-definition-free* path (in our simple language, only read and assignment statements define variables). The classic problem of live variables is defined as follows:

   > Variable x is *live* after node n iff there is an x-definition-free path from a successor of n to a node m that uses x.

   An assignment statement is *useless* if it assigns to a variable that is *not live*. Thus, live-variable analysis can be used to optimize a program by (i) performing live-variable analysis, and then (ii) removing useless assignment statements. A drawback of this approach is that after statements are removed by step (ii), there may be more opportunities to optimize the code by (iii) performing live-variable analysis again, and then (iv) removing the assignment statements that are now useless. This two-step process can be continued until there is a round when no assignment statement is removed.

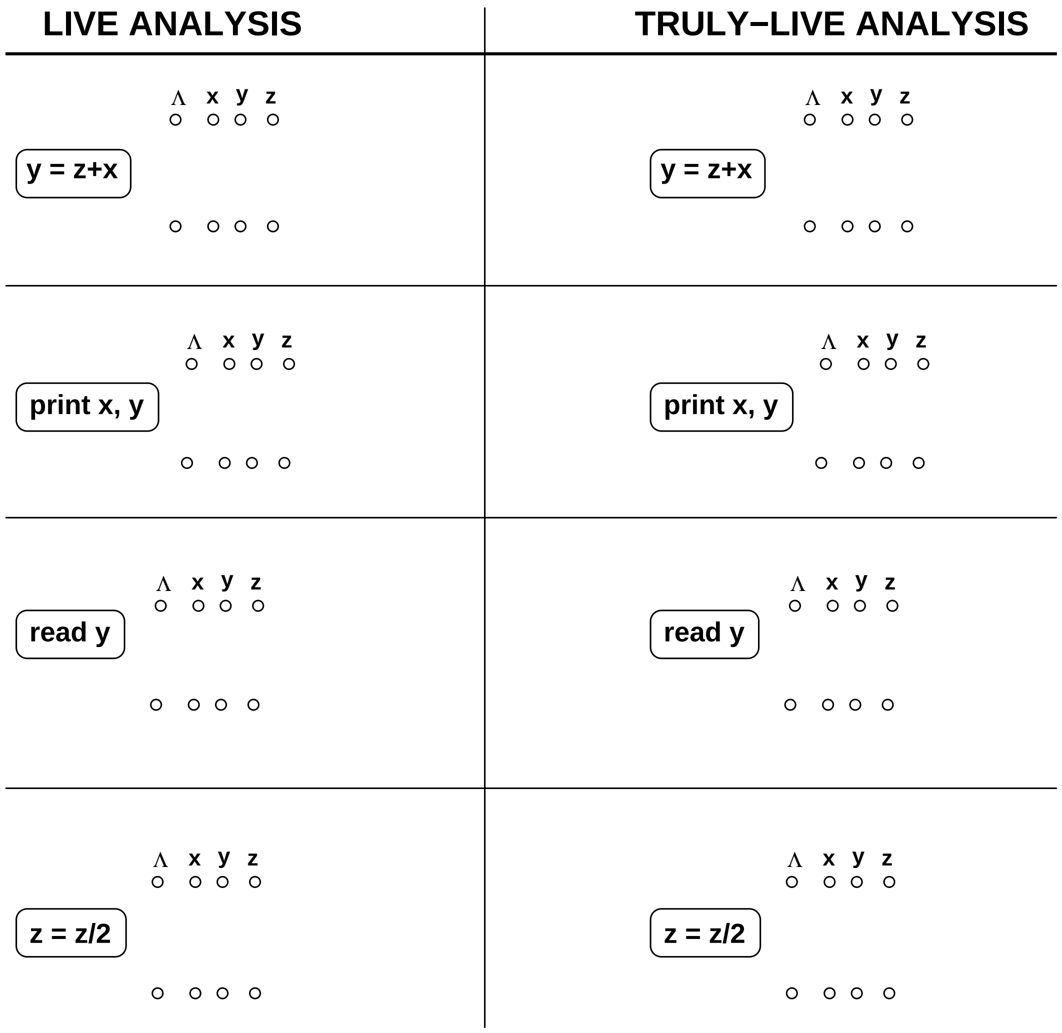
   To avoid the need to iterate—i.e., perform live-variable analysis, find and remove useless assignment statements, perform live-variable analysis, find and remove useless assignment statements, etc.—we can perform a different analysis: *truly-live* analysis, which can be defined as follows:

   > Variable x is *truly live* after node n iff there is an x-definition-free path from a successor of n to a node m that uses x, and:
   > - m is a condition, or
   > - m is a print statement, or
   > - m is an assignment statement that assigns to a variable y that is truly live after m.

   Now an assignment statement is *useless* if it assigns to a variable that is *not truly live*, and we would optimize the code by (i) performing truly-live-variable analysis, and (ii) removing useless assignment statements. A little thought will convince you that no additional assignment statements can be removed if steps (i) and (ii) were to be repeated.

   (a) Like live-variable analysis, truly-live analysis is a backward problem. Define truly-live analysis by specifying each of the following:
   - What is the type of a dataflow value associated with a node?
   - What is the combine operation?
   - What is the "initial" dataflow value (i.e., the value that holds at the *exit* of the main procedure)?

   (b) For each of the four different node-kinds k that we have in a program, we also need to specify the dataflow function for an edge whose target is of node-kind k. To do so, let's think about doing live-variable analysis and truly-live-variable analysis using the *exploded-supergraph* model used in the IFDS framework. Assume that we have a program with three variables: x, y, and z. Fill in the edges of the exploded supergraph in each of the pictures on the next page so that they correctly represent the dataflow function for an edge into the respective CFG node shown. Fill in the graph for live analysis on the left and the graph for truly-live analysis on the right.

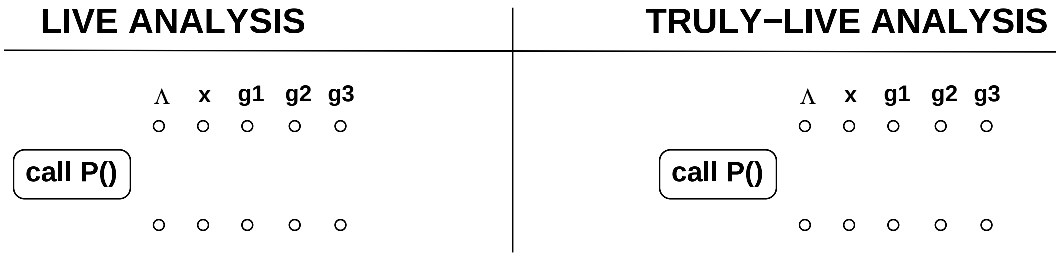

LIVE ANALYSIS
TRULY−LIVE ANALYSIS
Λ x y z
y = z+x
print x, y
read y
z = z/2

(c) Is truly-live-variable analysis a gen/kill problem? Justify your answer.

(d) Recall that when using the exploded supergraph to perform dataflow analysis, *summary edges* are added to the graph to summarize the effects of each procedure call. Assume that we have a program that includes the procedure P shown below, in which $g1$, $g2$, and $g3$ are all global variables.

```
void P() {
   g3 = g2;
   read g2;
   while (g2 > 0) {
       g3 = g2;
       g2 = g2 / 2;
   }
   if (g2 == 10) {
       g1 = g3;
   }
}
```

Assume that procedure *main* has one local variable x, and contains a single call to P. Complete the two pictures below by filling in the summary edges of the exploded supergraph that would be associated with the call to P in *main*—once for live analysis and once for truly-live analysis.



(e) Give an example for which truly-live analysis is better than repeated live-variable analysis; i.e., an example with an assignment statement "$x = \ldots$" such that truly-live analysis will find that x is *not* truly live afterwards (and so the assignment could be removed), but that fact will not be discovered by repeated live-variable analysis, because x is always live afterwards, no matter how often you repeat the steps of performing live-variable analysis and removing useless assignment statements. Hint: your example can involve just a single procedure; you'll need an example with a loop.

4. Recall that we can use the standard lattice framework for dataflow analysis to define an instance of a (forward, intraprocedural) dataflow problem for a given CFG as follows:
   - We define a complete lattice of dataflow values, where the lattice has no infinite ascending chains.
   - We assign a distributive (and hence monotonic) dataflow function to each CFG edge.
   - We specify a special "initial" dataflow value that holds at the start node of the CFG.

   For such problem instances, an algorithm that produces the solution to the dataflow problem (using the CFG) is the following:

```
for the entry node $e$, initialize val[$e$] to the special initial value
for all other nodes $n$, initialize val[$n$] to BOTTOM
repeat {
    for each node $n$, compute val[$n$] = $\bigoplus_{m \to n \in \text{Edges}} f_{m,n}(\text{val}[m])$
} until no change
```

   This algorithm finds the least fixed point of the set of equations that define the dataflow values for each of the CFG nodes. You can think of one iteration of the loop as an application of a function `f`. Using "BOT" to denote the vector of values in which all entries are BOTTOM except for the one for the entry node, whose value is the special initial value, the loop above computes

```
f(BOT)
f(f(BOT))
f(f(f(BOT)))
 ...
```

   until a fixed point is found. That fixed point is a value `v_0` = $\mathtt{f}^k$(`BOT`); i.e., k applications of `f` to `BOT`.

   This question is about doing incremental dataflow analysis: i.e., assume that you have used the above algorithm to solve a dataflow problem. Now someone edits the program, and you want the solution for the updated CFG, but you don't want to start over and compute the new solution from scratch.

   To simplify the problem, we will assume that the shape of the CFG is not changed (it has the same nodes and edges as before). However, the code in some of the nodes—and therefore the dataflow functions associated with those nodes' outgoing edges—has changed. This situation means that the function `f` discussed above has changed to a new function `f_new` (which is still monotonic).

   (a) Here is one way to compute the new solution (remember that `v_0` is the solution to the original problem):

```
v_1 = f_new(v_0) combine v_0
v_2 = f_new(v_1) combine v_1
v_3 = f_new(v_2) combine v_2
...
until no change
```

   Prove (i) that this process is guaranteed to terminate, and (ii) that it will produce a safe solution; i.e., one that is greater than or equal to the least fixed point of function `f_new`.

(b) While the loop above is guaranteed to terminate and to find a safe solution, it is not guaranteed to find the least fixed point of `f_new`. Show that this is true by supplying the following:

- a dataflow problem
- a simple CFG
- the solution to the dataflow problem (dataflow values for each CFG node)
- a change to the code at one CFG node
- the least solution to the new dataflow problem (the solution that would be found by solving the new problem from scratch)
- the (different) solution that would be found using the incremental approach described above in part (a).