CS 243 Homework 3 Solutions

Winter 2024

Due: February 7, 2024 at 11:59pm

Directions:

- Submit written answers via Gradescope.
- Complete the corresponding Gradiance quizzes by the due date.
- You have **two late days** on assignments for the entire quarter. See course website for details. **There are no late days for Gradiance.**
- This is an individual assignment. You are allowed to discuss the homework with others, but you must write the solution individually. If you look up any material in the textbook or online, you should cite it appropriately.

Problem 1. Optimizing dynamically typed languages

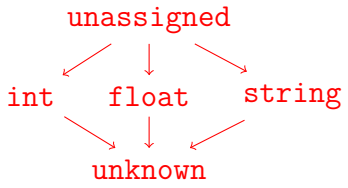
Consider a dynamically-typed language similar to JavaScript and Python. The language has three kinds of instructions:

- $x = T(\ldots)$, where T is one of $\{\texttt{int}, \texttt{float}, \texttt{string}\}$. This sets the variable x to a value of type T.
- $x = y$, where x and y are variables.
- print(x), where x is a variable.

Unlike statically-typed languages like C and Java, the type of a program variable is allowed to change in this language. As an example, the program `x = int(1); x = float(3.14)` will result in variable `x` having an `int` value at first, but afterwards a `float` value.

Suppose we are writing an optimizing compiler for this language. Because there are special instructions for printing an integer (`iPrint`), printing a float (`fPrint`), and printing a string (`sPrint`), your task is to identify those print(...) instructions whose operand type can be determined at compile time. This problem has four parts: (a) Define your dataflow analysis by filling in the table below. (b) State clearly how you perform the optimization. (c) Is your dataflow analysis monotone? Explain your answer. (d) Is your dataflow analysis distributive? Explain your answer.

Part a. Dataflow analysis specification. You may assume that each instruction is in its own basic block.

Direction of your analysis	Forward
Lattice elements and meaning	Map from each variable to one of the following: `unassigned` representing an unassigned variable; `int`, `float`, `string` representing a variable of definitely that type; `unknown` representing more than one possible types
Meet operator	For each variable: 
Is there a top element? If yes, what is it?	Mapping each variable to `unassigned`
Is there a bottom element? If yes, what is it?	Mapping each variable to `unknown`
Transfer function	$\text{OUT}[b, x] = \begin{cases} T & \text{if } b\text{: } x = T(\ldots), \\ \text{IN}[b, y] & \text{if } b\text{: } x = y, \\ \text{IN}[b, x] & \text{otherwise.} \end{cases}$
Boundary condition	Mapping each variable to `unassigned`
Interior points	Mapping each variable to `unassigned`

Alternative Soluion:

Direction of your analysis	Forward
Lattice elements and meaning	Each variable has a separate semi-lattice. Each semi-lattice element is a subset of {`int`, `float`, `string`} representing the types of values that a variable can take at that point in the program.
Meet operator	For each variable: Union $\cup$
Is there a top element? If yes, what is it?	$\emptyset$
Is there a bottom element? If yes, what is it?	{`int`, `float`, `string`}
Transfer function	$\textsc{out}[b,x] = \begin{cases} \{T\} & \text{if } b\colon x = T(\ldots), \\ \textsc{in}[b,y] & \text{if } b\colon x = y, \\ \textsc{in}[b,x] & \text{otherwise.} \end{cases}$
Boundary condition	Mapping each variable to $\emptyset$
Interior points	Mapping each variable to $\emptyset$

Part b. How do you optimize the program?

For each instruction of form $b\colon \text{print}(x)$ where $\textsc{in}[b,y] = T$ for some type T, replace it with the specialized instruction for printing out a value of type T.

Alternative solution: For each instruction of form $b\colon \text{print}(x)$ where $\textsc{in}[b,y] = \{T\}$ for some type T, replace it with the specialized instruction for printing out a value of type T.

Part c. Is your dataflow analysis monotone? Sketch a proof for it if so, or provide a counterexample if not.

Yes, since it is distributive. See below.

Part d. Is your dataflow analysis distributive? Sketch a proof for it if so, or provide a counterexample if not.

Yes. The print case is trivial since the transfer function is the identity. In the case of $b\colon x = T(\ldots)$, for any two mappings V, W and all $v \neq x$:

$$f_b(V \wedge W)[x] = T = T \wedge T = f_b(V)[x] \wedge f_b(W)[x] = (f_b(V) \wedge f_b(W))[x],$$
$$f_b(V \wedge W)[v] = (V \wedge W)[v] = V[v] \wedge W[v] = f_b(V)[v] \wedge f_b(W)[v] = (f_b(V) \wedge f_b(W))[v].$$

In the case of $b\colon x = y$, for any two mappings V, W and all $v \neq x$:

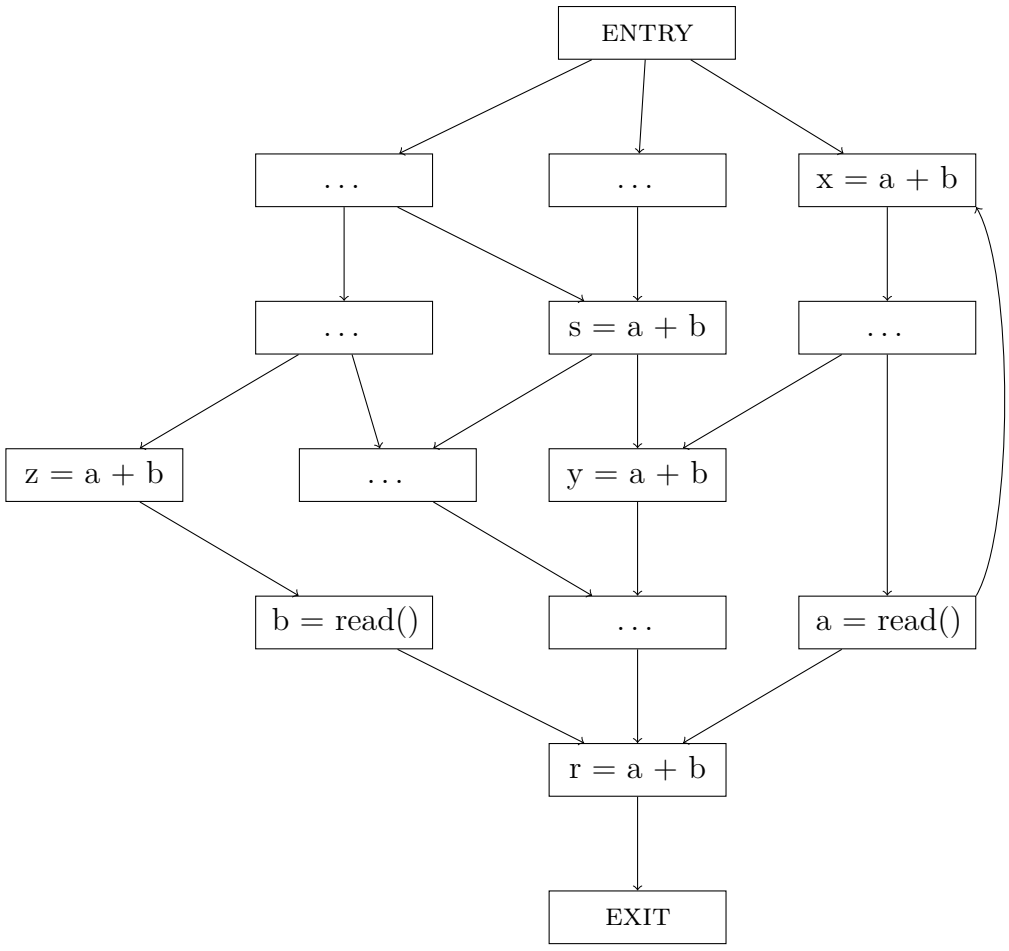
$$f_b(V \wedge W)[x] = (V \wedge W)[y] = V[y] \wedge W[y] = f_b(V)[x] \wedge f_b(W)[x] = (f_b(V) \wedge f_b(W))[x],$$
$$f_b(V \wedge W)[v] = (V \wedge W)[v] = V[v] \wedge W[v] = f_b(V)[v] \wedge f_b(W)[v] = (f_b(V) \wedge f_b(W))[v].$$

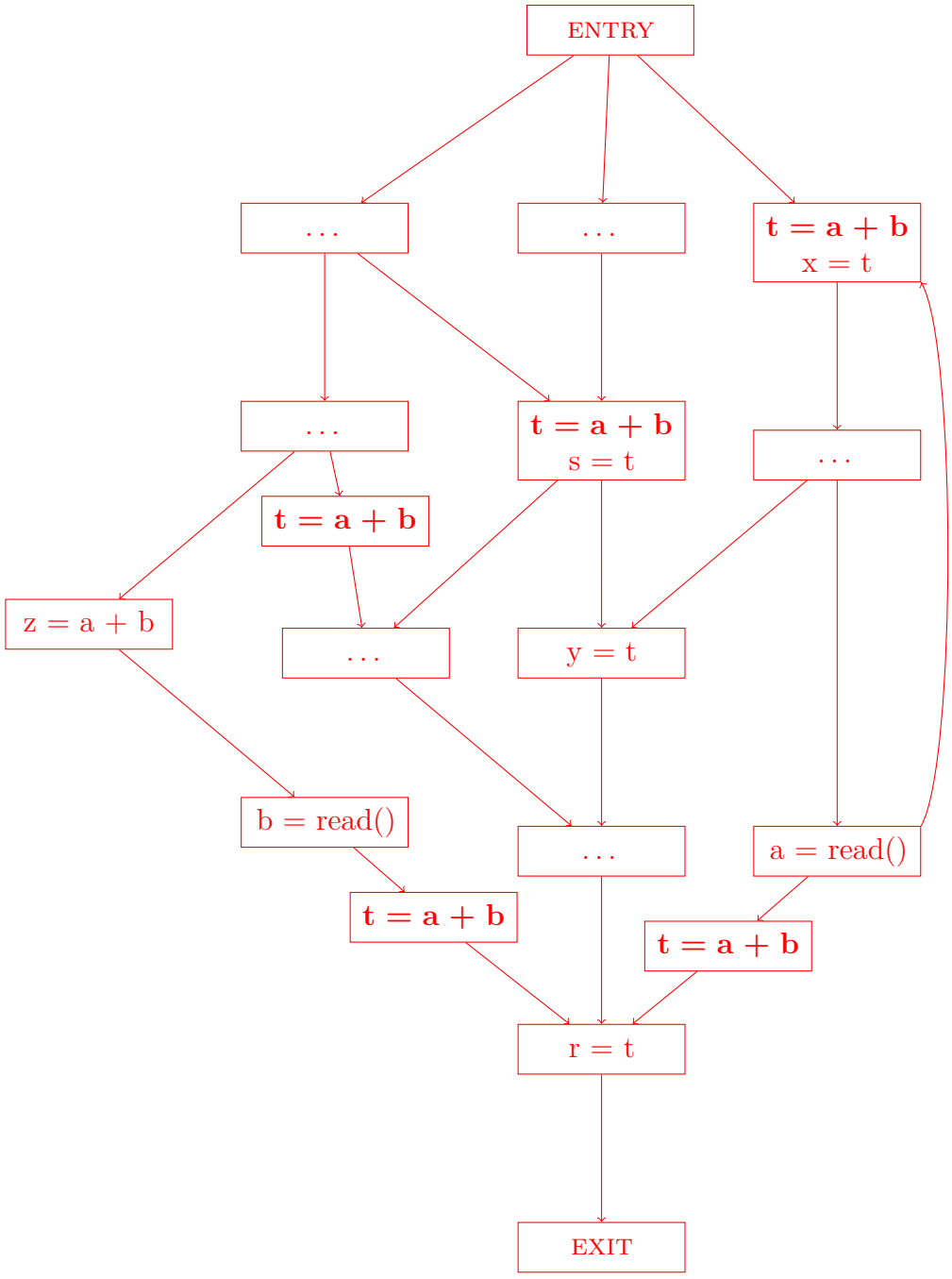
In conclusion, we have that $f_b(V \wedge W) = f_b(V) \wedge f_b(W)$.

Problem 2. Partial Redundancy Elimination.

Apply partial redundancy elimination to the following program. For simplicity, we omitted instructions that do not redefine variables; you must assume that all variables are used in all basic blocks shown. As a special case, the expression "read()" can return different values when called at different times, and thus does not participate in the lazy code motion algorithm.

You do not need to show the intermediate steps – just show the optimized code. You may add basic blocks to the flow graph, but only show those that are not empty in your solution. (Existing "..." basic blocks are not empty. They merely do not redefine any variable.)

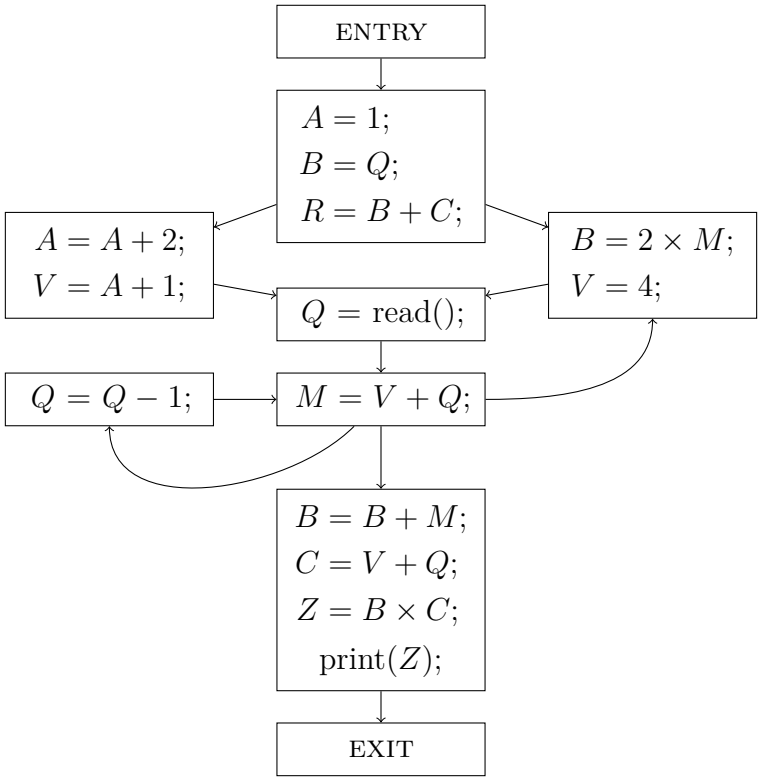



ENTRY
...
...
t = a + b
x = t
...
t = a + b
s = t
...
t = a + b
z = a + b
...
y = t
b = read()
...
a = read()
t = a + b
t = a + b
r = t
EXIT

Problem 3. Your task is to optimize the code below. You are only allowed to run the following four optimization techniques (in any order and multiple times if necessary):

- Dead code elimination using liveness analysis (as discussed in class and Homework 2),
- Constant propagation (as discussed in Lecture 4),
- Partial redundancy elimination (as discussed in Lecture 5),
- Copy propagation (as discussed in § 9.1.5 (pp. 590–591) of the textbook).

You cannot modify the control flow graph or eliminate empty basic blocks, except to preprocess it for PRE. Assume that expressions can take both registers and constants. You should not make any assumptions about the return value of read(), and assume print(Z) is some function that uses Z.



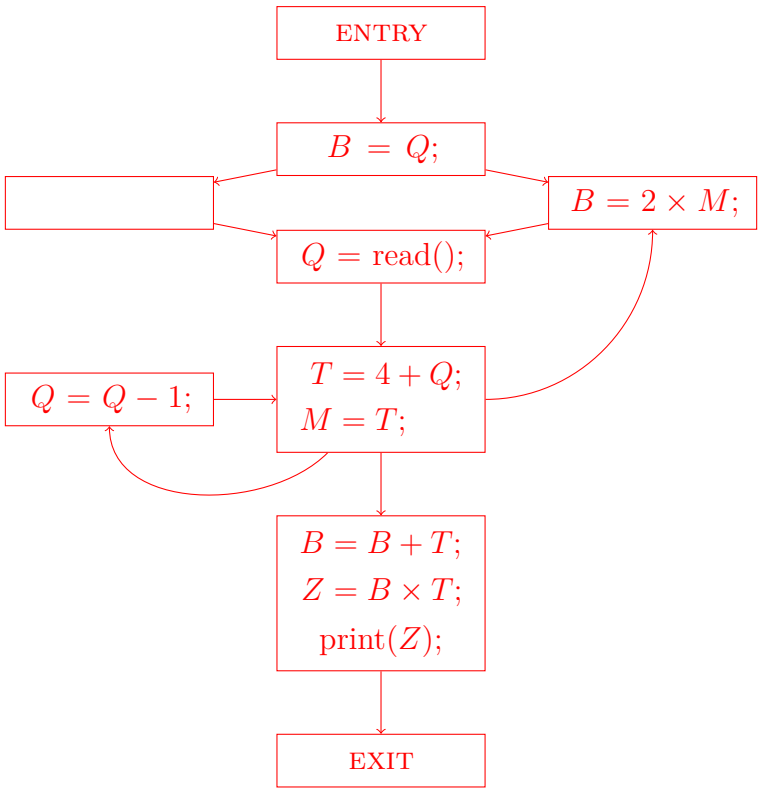
1. What are the optimizations and their order to produce the best optimized code for this specific program? Remember that you may run the same optimization more than once. You are allowed to abuse the optimizations, but try your best to minimize the number of passes of optimization to do the job.

2. What is the final optimized program?

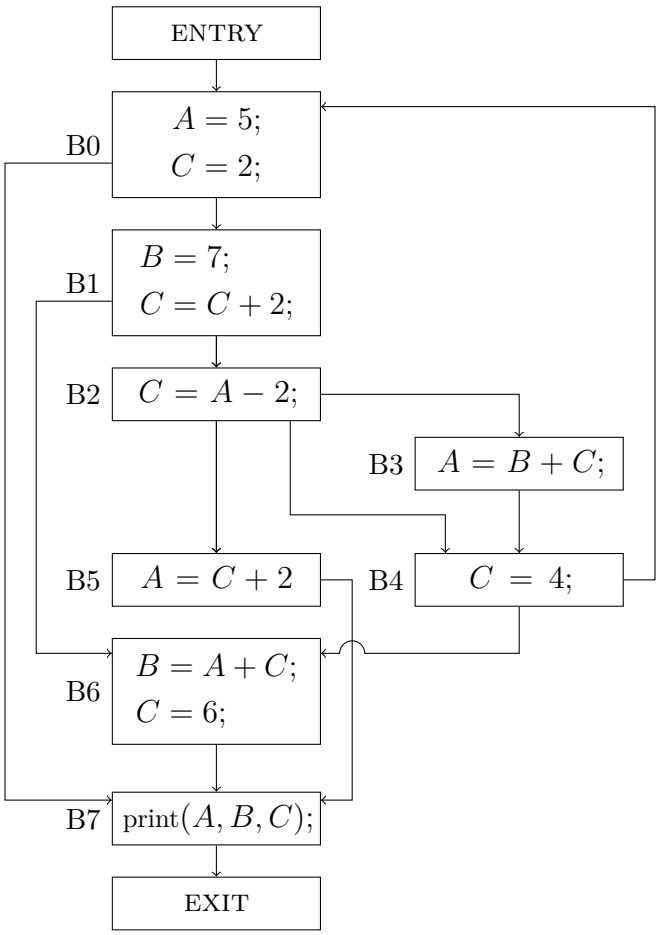
1. Note: there is not one unique answer. Your answer should be reasonable enough – you can run an optimization more times than necessary, but running all four optimizations ten times, for example, is not an acceptable answer.

   Reference solution: First run constant propagation and DCE to get rid of all statements associated with R and A and replace occurrences of V with 4. After that, apply PRE to the expression $4 + Q$. Next, apply copy propagation to make C dead. Finally, run a pass of DCE to eliminate C.

2. The final optimized program:

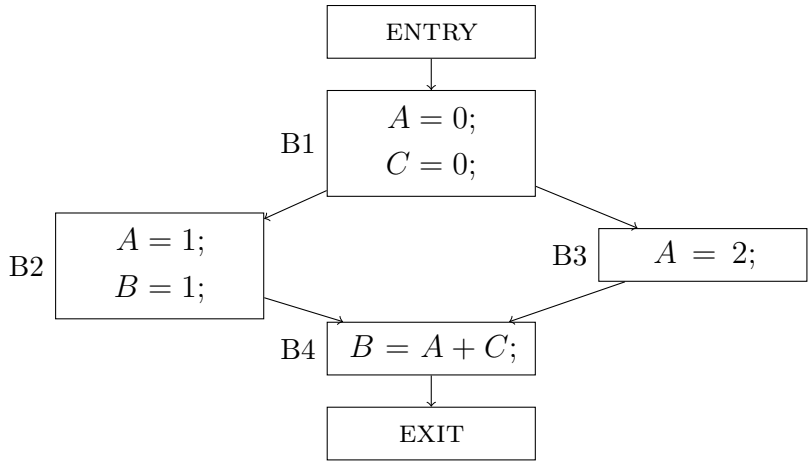


Problem 4. Register Allocation.

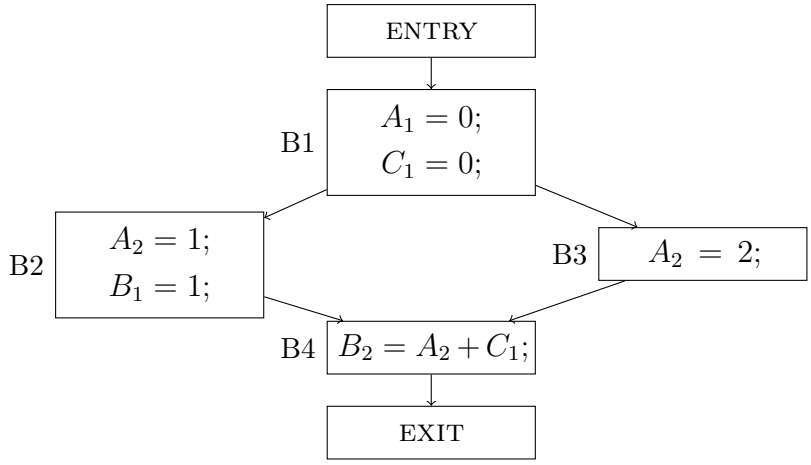


Perform register allocation for the above control flow graph. Specifically, show the results of each of the following steps:

1. Show the merged live ranges for the following program by updating the graph with unique variable notations, e.g., replace definitions and usages of B with B_1, B_2, etc. Note: if different definitions form a merged live range, use the same variable notation (you may convince yourself that this avoids ambiguity when a used variable may come from different definitions). We provide you with an example. For the following simple program:

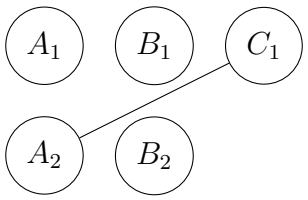


The resulting updated graph is:

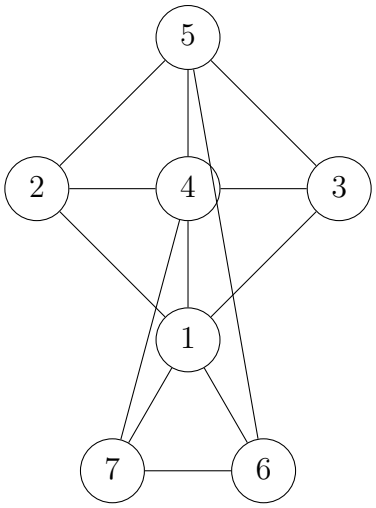


Taking the definitions of A in B2 and B3 as example, A is live at the beginning of B4, and both definitions reach that point, therefore they can be merged. **Note:** you do not need to consider any optimization such as PRE or constant propagation.

2. Draw the register interference graph with edges between nodes that represent merged live ranges. Using the same example program, the register inference graph should look like:

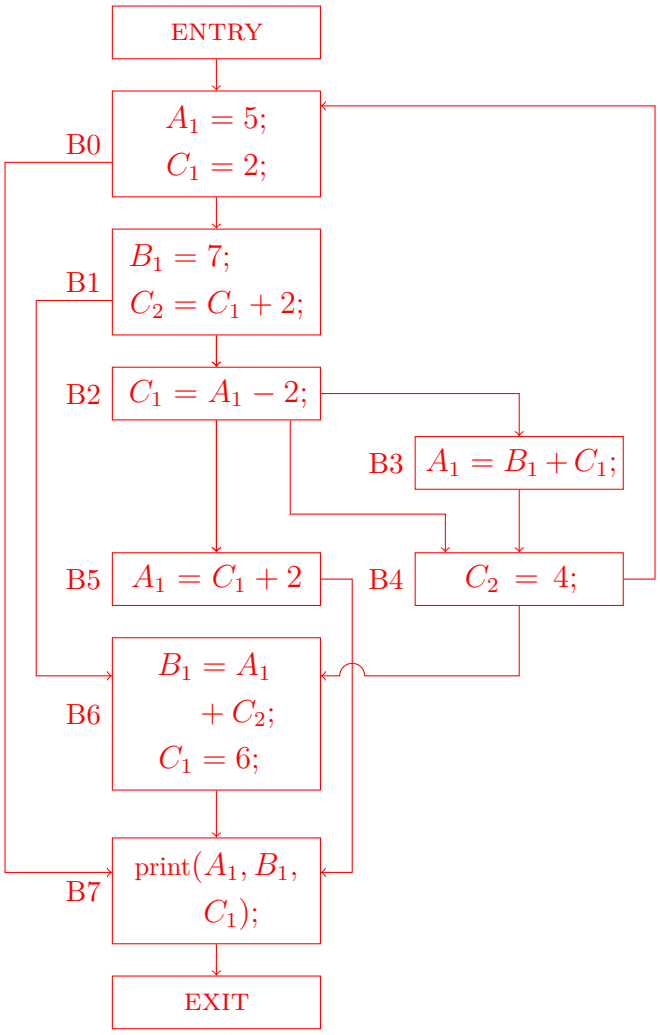


3. Apply the coloring algorithm for a machine with 3 registers to the following interference graph. Show a possible resulting "stack" of registers and show which ones, if any, are marked as spilled. When picking a node to spill, choose the node that is **lowest in numerical order**, and always pick the **first available color**.

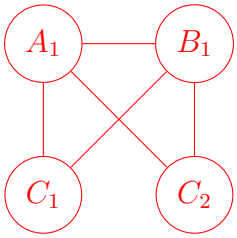


4. Is this graph colorable using 3 or fewer colors? If so, give such a coloring using the fewest number of colors (you do not need to use the coloring algorithm from class). Otherwise, briefly explain why not.

1. The graph with updated variable notations:



2. The register interference graph:



3. As there are no vertices with degree 2 or lower, we pick node 1 to spill. Next, we see that nodes 2, 3, 6, and 7 have opened up, so we pick node 2, then 3. Now all remaining nodes have degree 2, so we pick the remaining nodes in order. The final node ordering is 1 (spilled), 2, 3, 4, 5, 6, 7.

   Now we proceed to color the vertices in reverse order. Let the colors be red, green, and blue. The coloring proceeds as follows:

(a) 7: red

(b) 6: green — neighbor 7 is red

(c) 5: red — neighbor 6 is green

(d) 4: green — neighbors 5 and 7 are both red

(e) 3: blue — neighbors 4 and 5 are green and red

(f) 2: blue — neighbors 4 and 5 are green and red

(g) 1: (spilled) — neighbors 2, 3, 4, 6, and 7 are blue, blue, green, green, and red, so there is no color available.

4. Yes; this graph is 3-colorable, as follows:

(a) 1: green

(b) 2: red

(c) 3: red

(d) 4: blue

(e) 5: green

(f) 6: blue

(g) 7: red

This graph is not 2-colorable, as it contains a clique of size 3 (1-2-4).

Problem 5. Register Allocation and Live Ranges.

Part a. 1. Let n be the largest number of overlapping live ranges seen in a program. Let $v_1, \ldots, v_n$ be the set of n live ranges that overlap. What does this say about the corresponding nodes in the interference graph? Using your answer, give a bound on the minimum number of registers needed to execute this program without spilling or otherwise modifying the live ranges. You may assume the live range analysis is ideal, i.e. if two live ranges overlap, then there exists a program path that may be executed such that both variables are live at the same time.

2. Let n be the maximum number of *other* live ranges any live range overlaps with. For each of the following values of r, is register allocation: (i) always possible (ii) sometimes possible (iii) never possible? For (i) and (iii), give a brief proof sketch. For (ii), show examples for each case. You may assume the live range analysis is ideal.

   (a) $n - 1$ registers

   (b) n registers

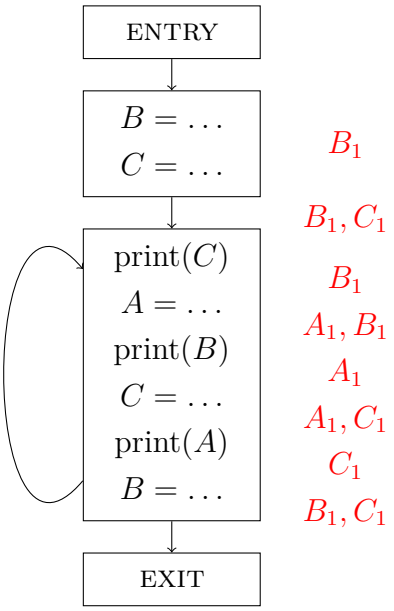
   (c) $n + 1$ registers

Part b. Observe the control-flow graph below and answer the following questions.

1. What is the largest number of overlapping live ranges seen at any program point?

2. What is the minimum number of registers you need in order to successfully assign all variables without spilling?

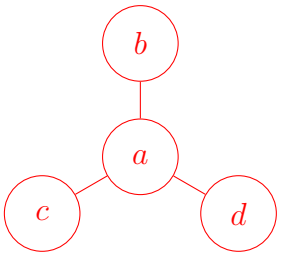
3. Now, imagine that, as part of register allocation, you can insert `MOVE x y` operations that copy a value from register `x` to another register `y`. Can you allocate all of the variables with fewer registers than before? If so, show the modified program and give the new number of required registers.



Part a.

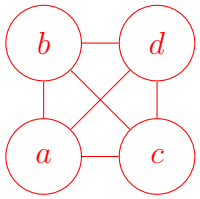
1. The nodes $v_1, \ldots, v_n$ form a clique (i.e. each node is connected to every other) of size n. Therefore, the program requires at least n distinct registers.

2. (a) Sometimes possible. Consider the star graph S_3:



   This graph is 2-colorable but it has maximum vertex degree 3. Therefore, a program with this interference graph (so $n = 3$) can still be allocated with only 2 registers.

   For an example which is not 2-colorable or even 3-colorable, consider K_4, the complete graph with 4 vertices:



   (b) Sometimes colorable. The same examples from the previous answer apply here as well. Fun fact: Brook's theorem states that the only such graphs that are not n-colorable are complete graphs and odd-length cycle graphs.

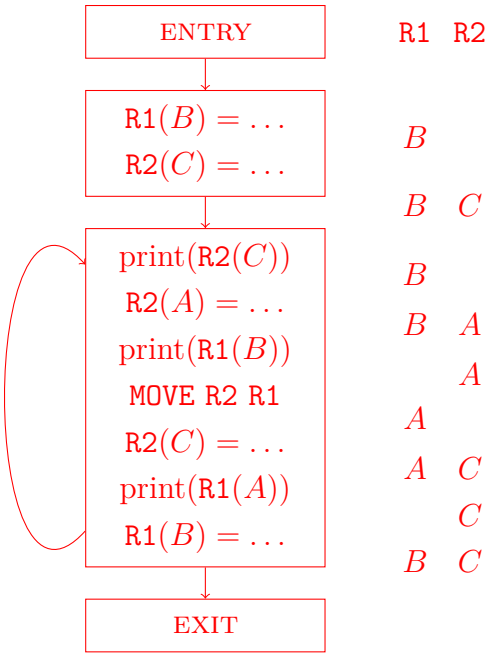
   (c) Always colorable. The algorithm presented in class always finds a coloring, as no matter what the partial coloring is, each node has at most n neighbors, so there is always at least one available color.

Part b.

1. Two live ranges. See the above diagram.

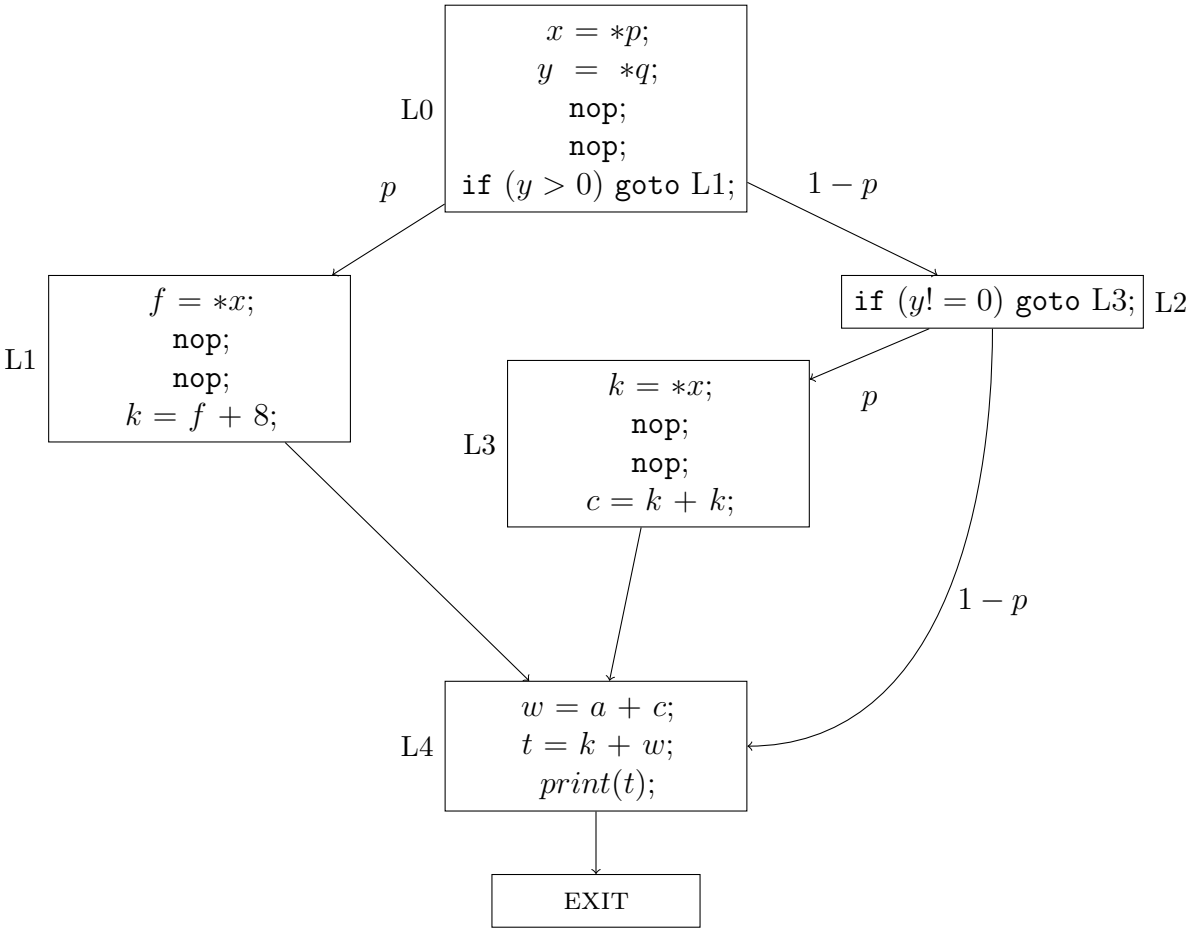
2. Three registers, due to the odd-length cycle A_1–B_1–C_1–A_1 in the interference graph.

3. Yes, with only two registers. Example below. The `MOVE` instruction essentially splits the live range A_1 into two.


ENTRY
R1(B) = ...
R2(C) = ...
print(R2(C))
R2(A) = ...
print(R1(B))
MOVE R2 R1
R2(C) = ...
print(R1(A))
R1(B) = ...
EXIT
R1 R2
B
B C
B
B A
A
A
A C
C
B C

Problem 6. Global Instruction Scheduling

Assume you have a machine with a statically scheduled processor that can only issue one operation every clock. All operations have a latency of one clock cycle, with the exception of its memory load operation, which has a latency of three clock cycles. Consider the following locally scheduled program:



All variables in the program are defined before the entry (L0) of the program. Assume that only variable t is live at the exit of the program. Each branch in the flow graph is labeled with the probability that it is taken dynamically. To answer the following questions, you may apply any of the code motions discussed in class, but no other optimizations.

Is this the best globally scheduled code that can be generated given that $p = 0.8$? If not, provide the improved code along with its expected execution time. **Note: for an `if` branch instruction, assume that if the branch corresponding to "`goto` L" is taken, it costs two cycles; otherwise it costs only one.**

Note: There can be multiple solutions to this problem. We would score well as long as the scheduling is reasonable enough.

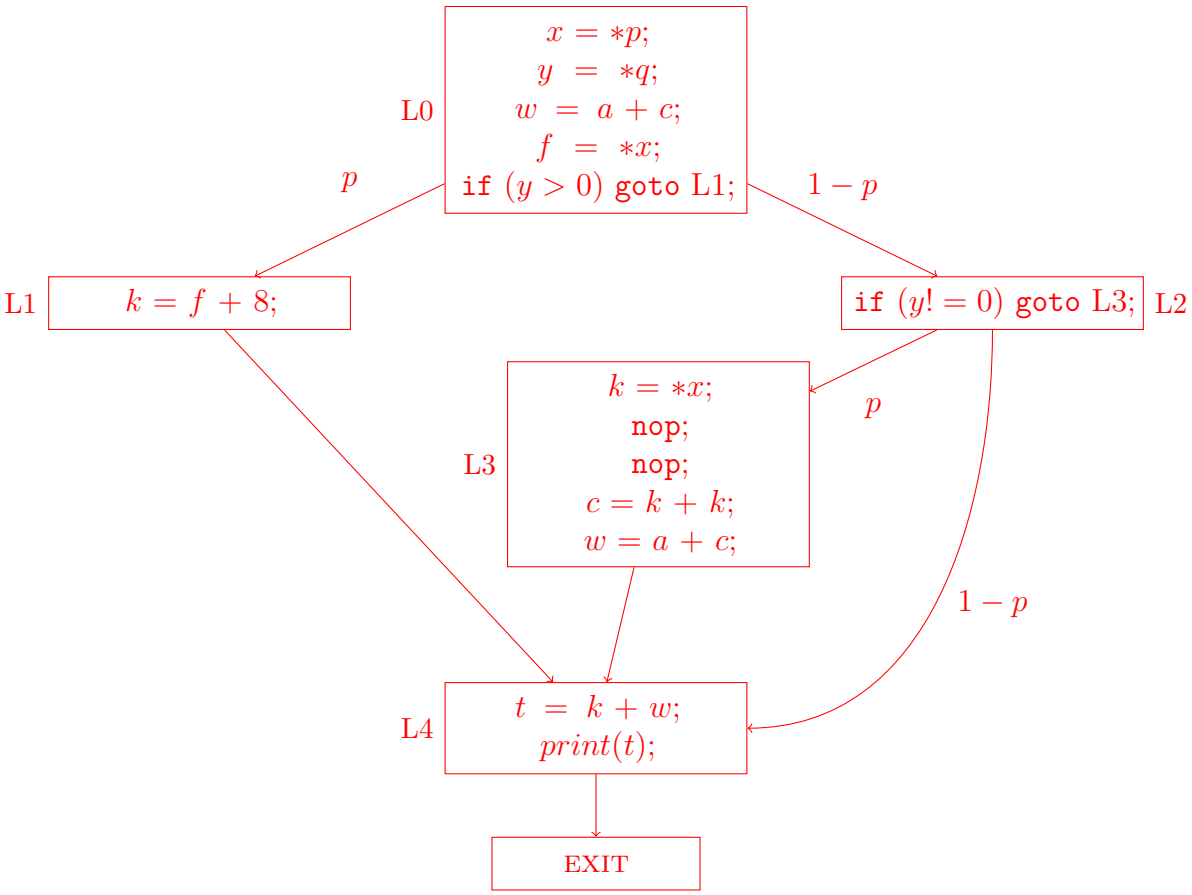
In the original program, the number of clock cycles and the probability for each path are shown as follows:

- L0 → L1 → L4, $p = 0.8$, num_cycles = 13

- L0 → L2 → L4, $p = 0.04$, num_cycles = 9
- L0 → L2 → L3 → L4, $p = 0.16$, num_cycles = 14

The expected execution time is $0.8 \times 13 + 0.04 \times 9 + 0.16 \times 14 = 13$ cycles.

With the following optimized program:

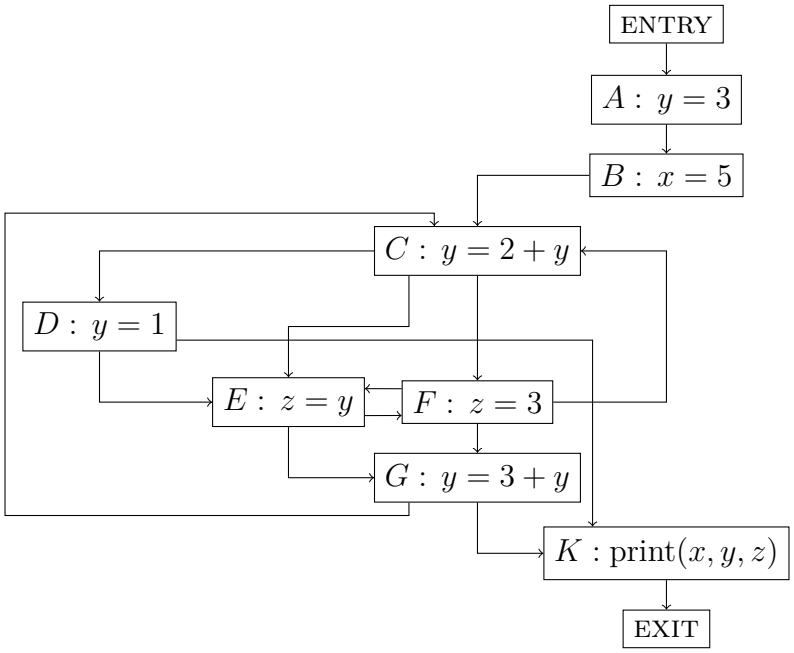


the number of clock cycles and the probability for each path are shown as follows:

- L0 → L1 → L4, $p = 0.8$, num_cycles = 9
- L0 → L2 → L4, $p = 0.04$, num_cycles = 8
- L0 → L2 → L3 → L4, $p = 0.16$, num_cycles = 14

The expected execution time is $0.8 \times 9 + 0.04 \times 8 + 0.16 \times 14 = 9.76$ cycles.

Problem 7. Dominance Relation.



Part a. A basic block d post-dominates another block n (sometimes written d pdom n) if all paths from n to the exit pass through d. Write a dataflow analysis that computes the set of nodes that post-dominate it (its post-dominators). In your analysis, let B be the set of basic blocks in the program, and b_{exit} a dummy basic block for the exit.

Direction of your analysis	Backward
Lattice elements and meaning	Sets of basic blocks
Meet operator	Intersection
Is there a top element? If yes, what is it?	Set of all basic blocks
Is there a bottom element? If yes, what is it?	Empty set
Transfer function	$\text{IN}[b] = \{b\} \cup \text{OUT}[b]$
Boundary condition	$\{b_{\text{exit}}\}$
Interior points	set of all basic blocks

Part b. A node d *strictly* post-dominates n if d pdom n and $d \neq n$. Compute the strict post-dominators for each basic block. You do not have to write the block itself.

$A: C, B, K, b_{\text{exit}}$

$B: C, K, b_{\text{exit}}$

$C: K, b_{\text{exit}}$

$D : K, b_{\text{exit}}$

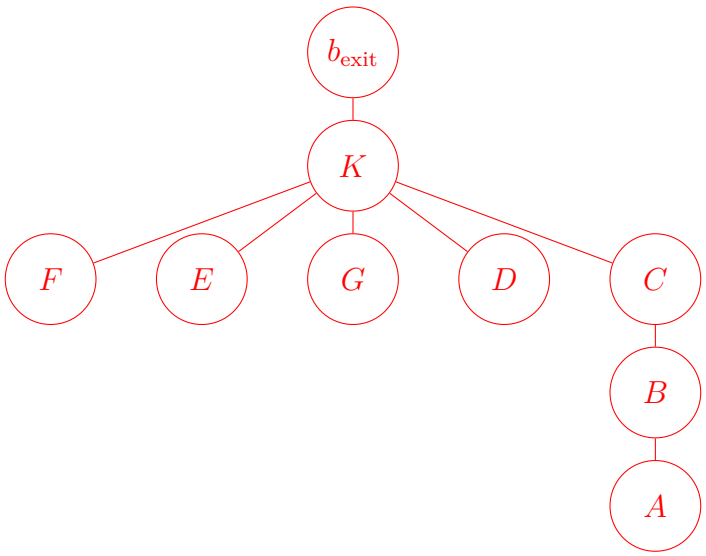
$E : K, b_{\text{exit}}$

$F : K, b_{\text{exit}}$

$G : K, b_{\text{exit}}$

$K : b_{\text{exit}}$

Part c. The immediate post-dominator of a node n is the unique node that strictly post-dominates n but does not strictly dominate any other node that strictly post-dominates n. Each node other than the exit node has an immediate post-dominator. Connect each node with its immediate post-dominator to form the *post-dominator tree*. **Hint:** as its name implies, the post-dominator tree is a tree (it is connected and has no cycles when interpreted as an undirected graph.)



Problem 8. [OPTIONAL] How many hours did it take for you to finish this assignment?