CS143 Written Assignment 4

Your Name – SUNet ID

Due: June 3rd, 2025 EOD

Important Note: for ease of grading, we do not accept hand-written submissions for this WA. Please use the LaTeX solution template. If you do not wish to set up LaTeX locally, we recommend using Overleaf. If you believe you have justifiable reasons to submit a hand-written solution, feel free to contact the staff email list.

Problem 1. Programming in Assembly

Consider the following fictional hardware architecture consisting of two signed 64-bit integer registers `x`, `y`, and a large stack where each stack element holds a signed 64-bit integer.

The hardware architecture supports the following instructions:

- `push r`: Given register `r`, push the value of register `r` to the top of the stack.
- `pop r`: Given register `r`, pop the top of the stack into register `r`.
- `swap`: Swap the value at the top of the stack with the value right beneath it.
- `add r1 r2`: Given two registers `r1`, `r2`, compute $r1 + r2$ and store it into register `r1`.
- `sub r1 r2`: Given two registers `r1`, `r2`, compute $r1 - r2$ and store it into register `r1`.
- `mul r1 r2`: Given two registers `r1`, `r2`, compute $r1 \times r2$ and store it into register `r1`.
- `div r1 r2`: Given two registers `r1`, `r2`, compute $r1$ / $r2$ and store it into register `r1`.
- `jz c`: Given signed 64-bit integer constant literal `c`, if the top of the stack is zero, jump to `c` instructions *after* this instruction. Otherwise, execute the next instruction. For example, `jz 0` is a no-op instruction since the next instruction will be executed no matter the value of the stack top (unless the stack is empty, in which case the machine crashes). As another example, if the stack top is zero, `jz -1` would cause an infinite loop when executed, since it jumps to itself.
- `print r`: Given register `r`, print its value to console.
- `halt`: Stop execution.

All arithmetic operators have the same behavior as C `int64_t` arithmetic, though you should normally not need to worry about this for this problem.

Your program starts with an empty stack, with register `x` holding an integer value $1 \leq n \leq 19$, and register `y` holding zero.

You should write a program that, when executed, prints out n lines and halt, where the i-th (1-indexed) line should contain the value of the factorial of i (for example, $4! = 24$).

Your program should contain one instruction per line. You can also write comments: everything after a `#` character in the same line will be treated as comments and discarded. In programming such an exotic architecture, comments will be very helpful for you to keep track of your program's design and invariants: make use of them!

You can find a simulator of the architecture at Stanford Myth location

```
/afs/ir/class/cs143/bin/wa4_2025/cs143wa4p1
```

On Stanford Myth, run

```
/afs/ir/class/cs143/bin/wa4_2025/cs143wa4p1 <prog_file> <n>
```

to run your program with register `x` holding value n. Alternatively, you can run

```
/afs/ir/class/cs143/bin/wa4_2025/cs143wa4p1 <prog_file> <n> --trace
```

to additionally print out the machine state after executing each instruction, which will be helpful for you to debug your program.

When you are confident with your program, run

```
/afs/ir/class/cs143/bin/wa4_2025/cs143wa4p1 <prog_file> test
```

to run the staff's tests. If all tests passed, it will print out a secret string. **Be sure to include this secret string in your solution!**

Solution:
The secret string is:

Your program:

Problem 2. Single Static Assignment (SSA) Form

Single static assignment (SSA) intermediate representation (IR) is the foundation of most modern compilers (e.g., LLVM, GCC, etc.). The difference between an SSA IR and the IR you learned in the lecture is that in an SSA IR:

1. Each variable is assigned *exactly* once in the IR.
2. A variable must be assigned before it is used.

To illustrate the first rule, the following IR basic block

$$a := 1$$
$$a := a + 1$$
$$a := a \times a$$

is *not* in SSA form because the variable a is assigned three times in the IR.

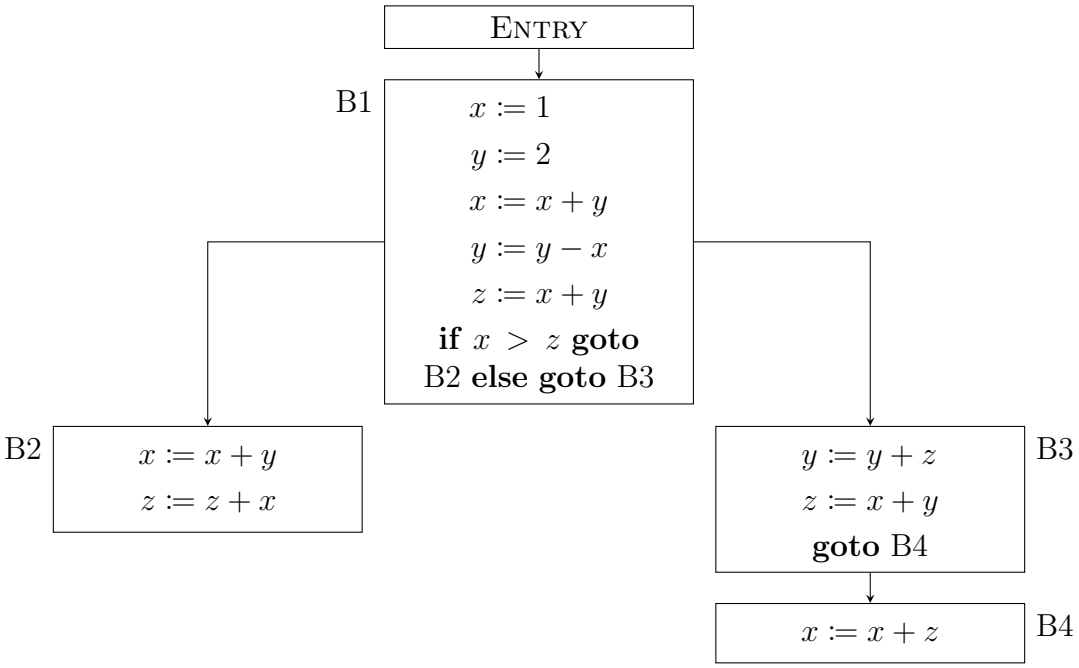
How can we convert a non-SSA IR to an equivalent IR in SSA form? The first key idea is to create a new variable for each assignment. For example, the above basic block can be converted to the following equivalent IR in SSA form:

$$a_1 := 1$$
$$a_2 := a_1 + 1$$
$$a_3 := a_2 \times a_2$$

Note how we created a new variable for each assignment to a, and replaced each use of a to the most recent version of a we created.

If the control flow graph (CFG) of the program resembles a tree, the above technique is enough to convert it to SSA form.

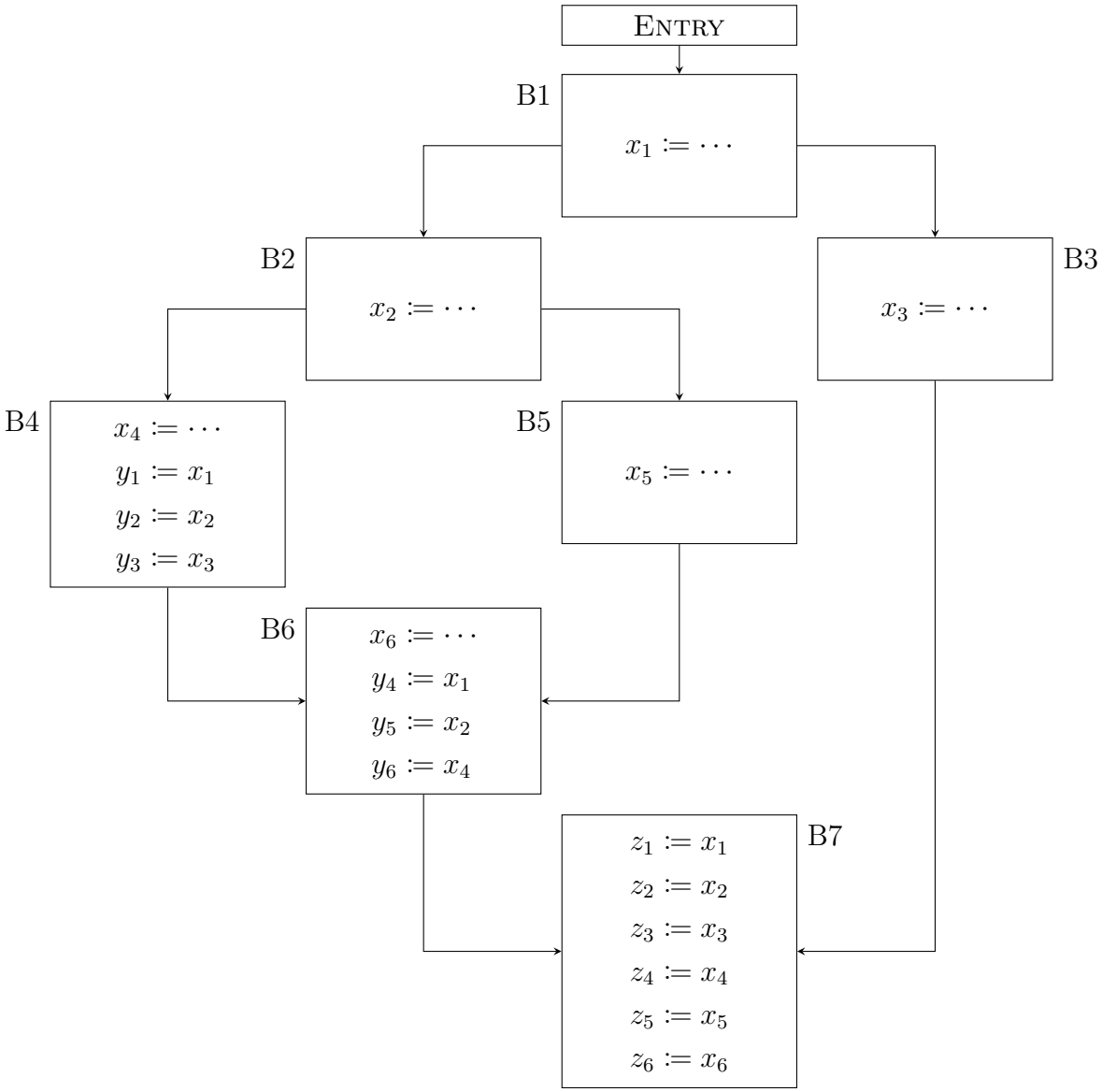
(Task 2.A) To familiar yourself with the process, convert the following IR to SSA form:



Since each variable is assigned exactly once, one can treat this assignation as the *definition* of the variable. The second rule of SSA now becomes natural: a variable must be assigned (defined) before it is used.

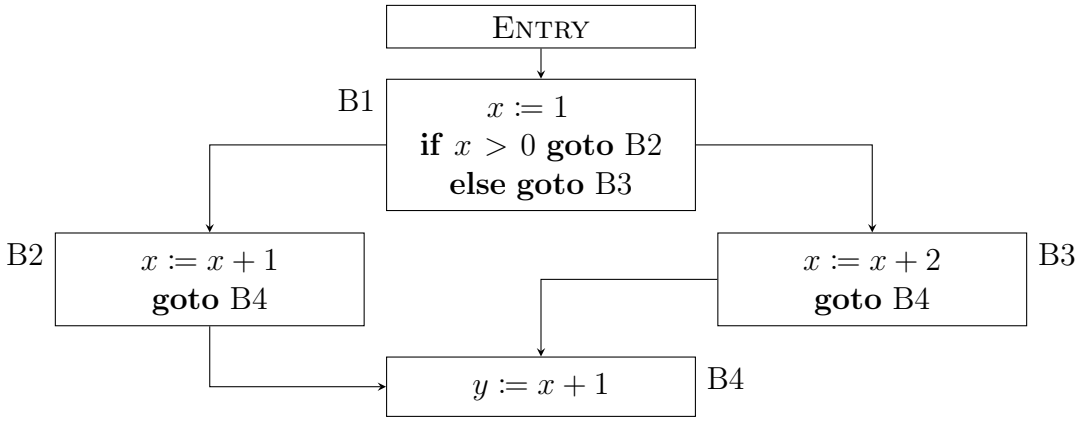
Note that this is a *static* property. Specifically, if a variable v is defined in basic block X and used in basic block Y, then there must be no way in the CFG to reach basic block Y from the entry block without going through basic block X, or the IR is ill-formed (and if $X = Y$, then the definition must precede the use). This *statically* guarantees that any time a variable v is used, it must have been assigned a valid value.

(Task 2.B) Consider the following IR (unrelated parts omitted for brevity):

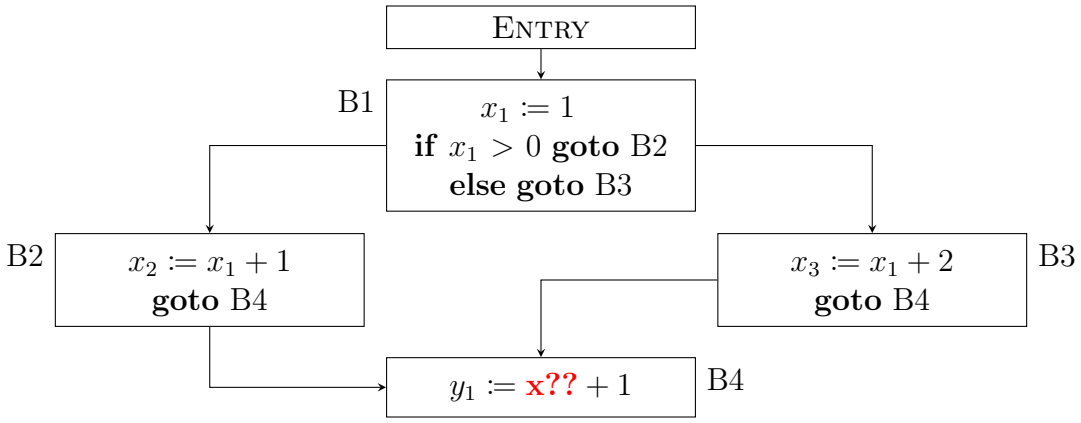


Assume the definition of $x_1, \cdots, x_6$ are all valid. Among $y_1, \cdots, y_6$, $z_1, \cdots, z_6$, which variables are ill-defined in an SSA IR?

Only replacing each variable assignment with a fresh variable is not enough to convert any IR graph to SSA form. For example, for the IR below:



The conversion process would yield the following:



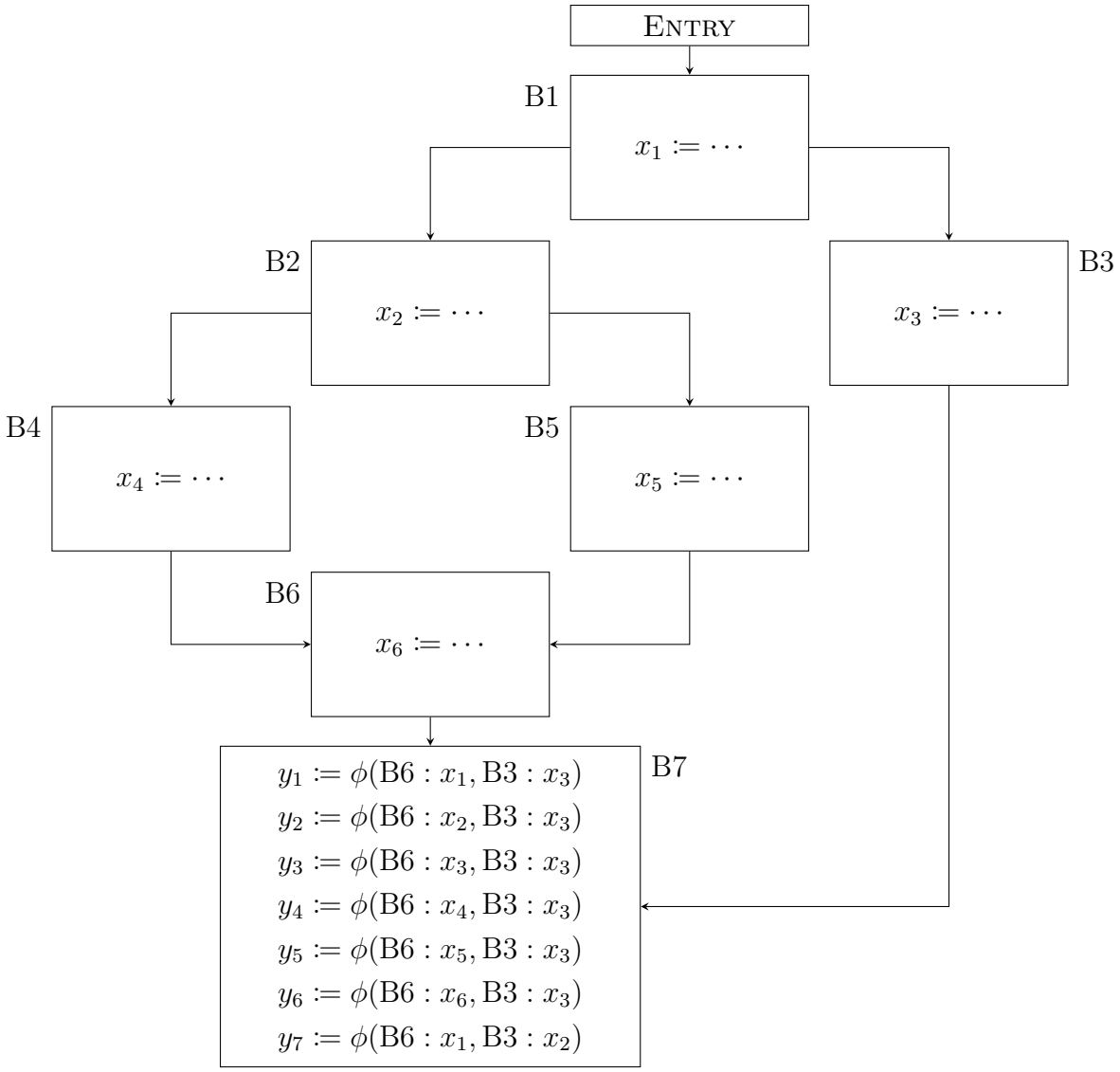
As you can see, the value of variable x in basic block B4 depends on whether B4 is reached from B2 or B3, so it would be wrong to replace the x in B4 with either x_2 or x_3.

Thus, we need a way to represent variables which value depends on where the control flow comes from. This is called ϕ-nodes in SSA. Given a basic block B with n direct predecessors $P_1, \cdots, P_n$, a ϕ-node in basic block B must take n variables $v_1, \cdots, v_n$ as arguments. If basic block B were reached from P_i, then the ϕ node would evaluate to v_i.

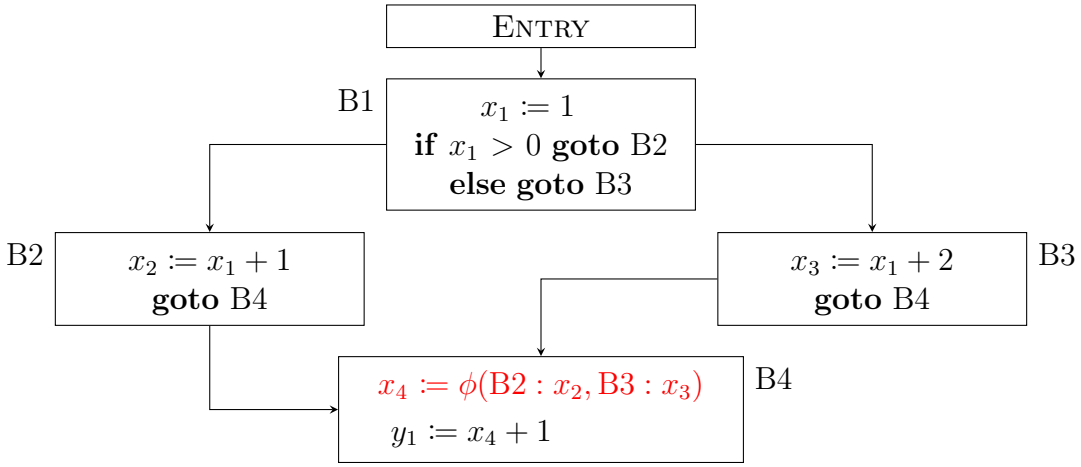
For the purpose of this problem, you should write a ϕ node as $\phi(P_1 : v_1, \cdots, P_n : v_n)$. As a convention, a ϕ node must be directly assigned to a variable: you cannot use a ϕ node as part of an expression. For example, $x \coloneqq \phi(P_1 : v_1, P_2 : v_2) + 1$ is invalid, but you can always workaround by doing $x_1 \coloneqq \phi(P_1 : v_1, P_2 : v_2); x_2 \coloneqq x_1 + 1$ instead. All the ϕ-node assignments must also show up before all the non-ϕ-node assignments in a basic block.

Similar to variables, ϕ nodes subject to the def-before-use requirement. Specifically, for each $1 \leq i \leq n$, if v_i were defined in basic block X, then there must be no way to reach basic block P_i from the entry block without going through basic block X. This statically guarantees that at the time we reach basic block B from P_i, the *corresponding* v_i (that becomes the result value of the ϕ node) must have been evaluated.

(Task 2.C) Consider the following IR (unrelated parts omitted for brevity):



Assume the definition of $x_1, \cdots, x_6$ are all valid. Which of $y_1, \cdots, y_7$ are ill-defined?
With ϕ nodes, we can now convert the earlier IR example to SSA form, as shown below:



But can we do this conversion programatically? It turns out that if we do not care about how many ϕ nodes we create, the conversion is fairly straightforward:

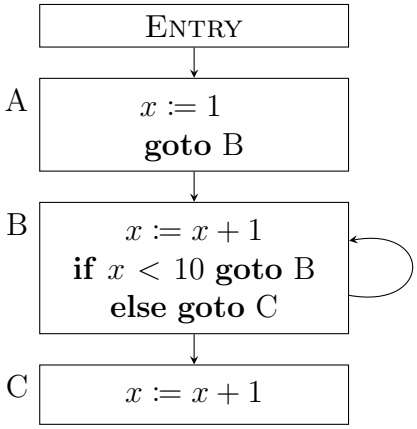
1. Create a new basic block, create assignments in the basic block to initialize every variable to `UNDEFINED`. Make this basic block the new entry block, and let it branch to the original entry block.

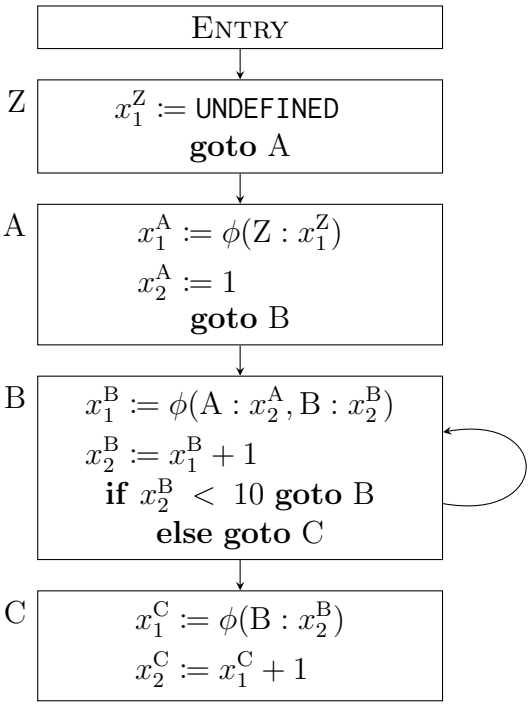
2. For each basic block B except the new entry block, for each variable x, add assignment $x_1^{\mathrm{B}} \coloneqq \phi(P_1 : \square, \cdots, P_n : \square)$ at the start of the basic block to represent the starting value of variable x at basic block B. We will fill in the $\square$ in the ϕ later.

3. Inside each basic block, perform the conversion as described in Task 2.A. Specifically, for each non-ϕ-node assignment, replace each variable in the right-hand side with the most recent version of this variable in this basic block (which always exists since we have created ϕ nodes at the start of the basic block for all variables), and change the left-hand side to a fresh variable.

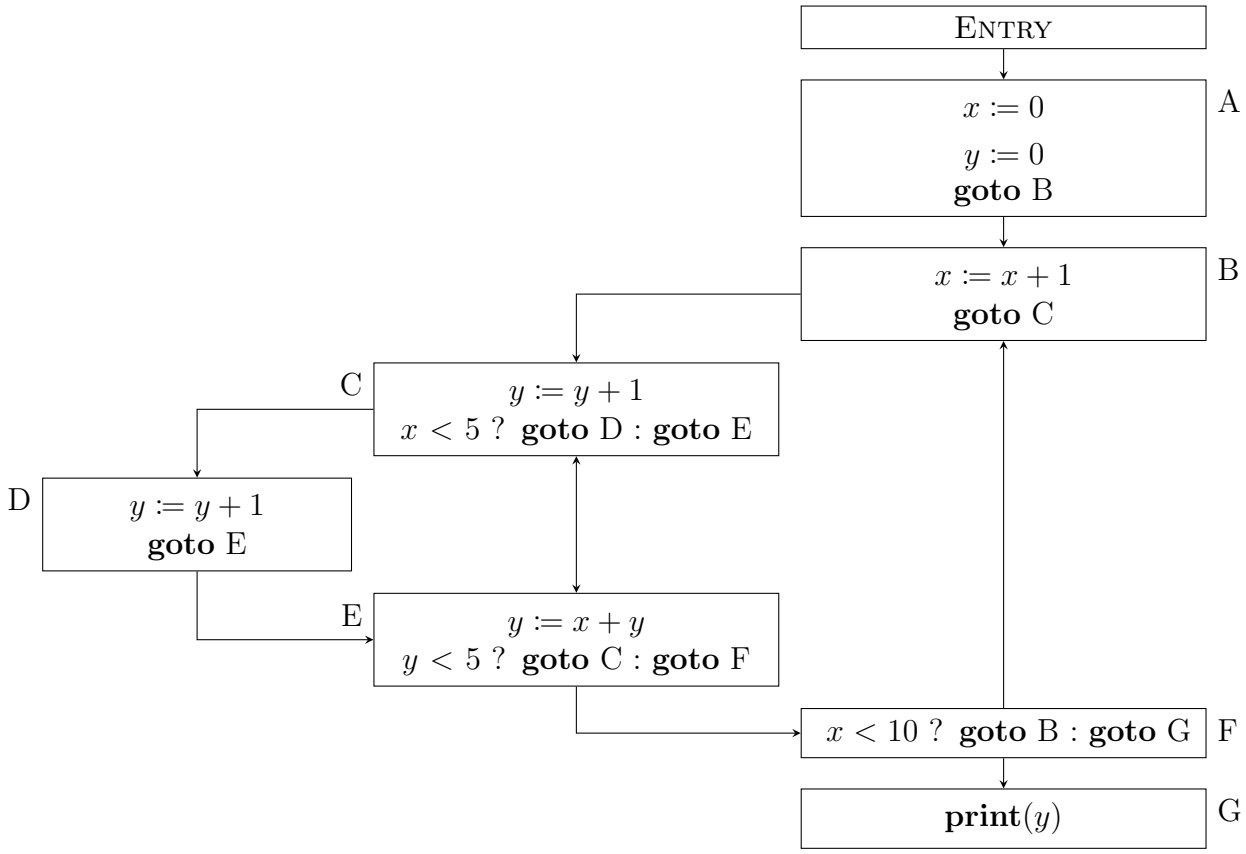
4. For each ϕ node assignment $x_1^{\mathrm{B}} \coloneqq \phi(P_1 : \square, \cdots, P_n : \square)$ created in step 1, set the $\square$ corresponding to basic block P_i to the last version of variable x in block P_i.

For example, given the following non-SSA IR on the left, the algorithm would convert it to the equivalent SSA IR on the right:





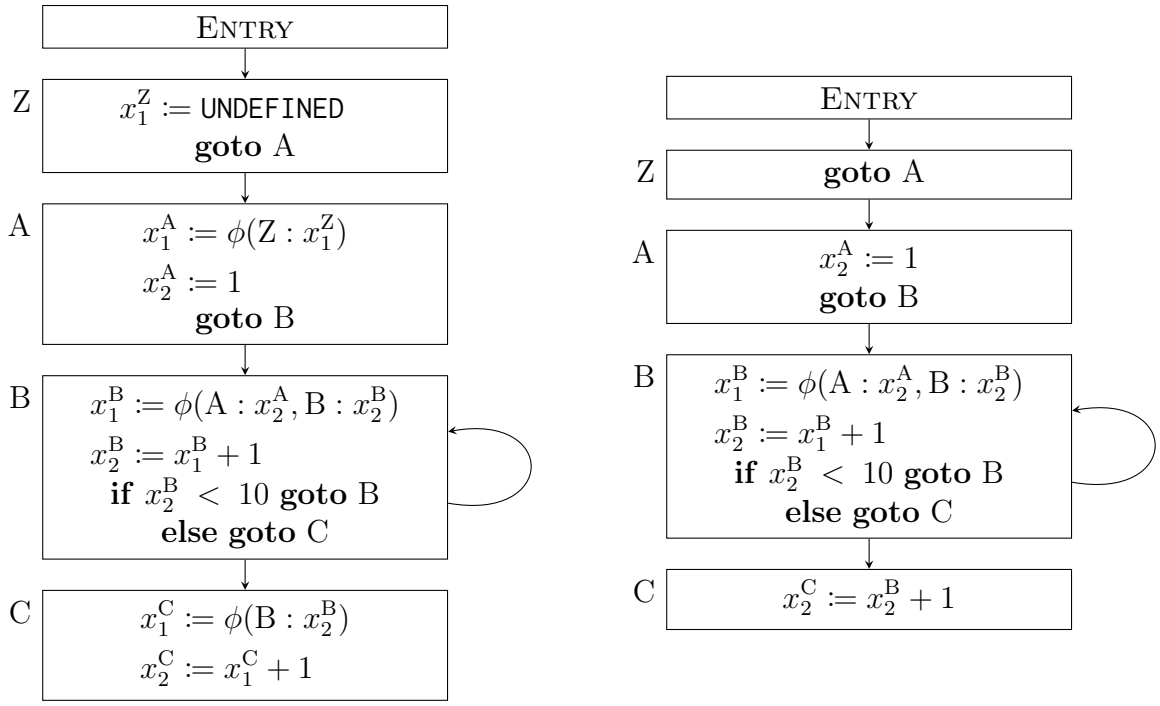
(Task 2.D) Given the following IR, convert the IR to SSA form by executing the algorithm. Note that you should strictly follow the algorithm.



(Task 2.E) Explain why the algorithm always (i.e., not only for the example in Task 2.D, but in general) produces valid SSA IR. That is, you should explain why the following are true: every variable is defined exactly once, every variable satisfies the def-before-use requirement, and every ϕ node satisfies the def-before-use requirement.

(Task 2.F) Dead code elimination is trivial in SSA form. Since SSA statically guarantees that each variable is assigned exactly once, and any use of a variable can only happen after the assignation, a variable assignment is dead code and can be removed if and only if the right-hand side has no side effect, and the variable in the left-hand side has no use in the IR. This process is then repeated until nothing can be deleted. Perform dead code elimination on the SSA IR you get in Task 2.D. Which variables are dead? (You do *not* have to show the SSA IR after dead code elimination.)

You might have noticed that not all ϕ nodes are required. For example, in the SSA IR of the simple loop (the example we used before Task 2.D, figure duplicated on the left below for your convenience), replacing all *uses* of variable x_1^{C}, which is defined as $x_1^{\mathrm{C}} \coloneqq \phi(\mathrm{B} : x_2^{\mathrm{B}})$, with variable x_2^{B} will not change program behavior. After such replacement, x_1^{C} will have no use anymore (since all its uses have been replaced). Since ϕ-nodes have no side effect, and x_1^{C} has no uses, dead code elimination would allow you to remove the definition of x_1^{C} as dead code, thus eliminating a ϕ node from the SSA IR. The figure below on the right shows the simplified SSA IR for the simple loop example, after replacing all uses of x_1^{C} with x_2^{B}, and removing dead variables $x_1^{\mathrm{Z}}, x_1^{\mathrm{A}}$ and x_1^{C}.



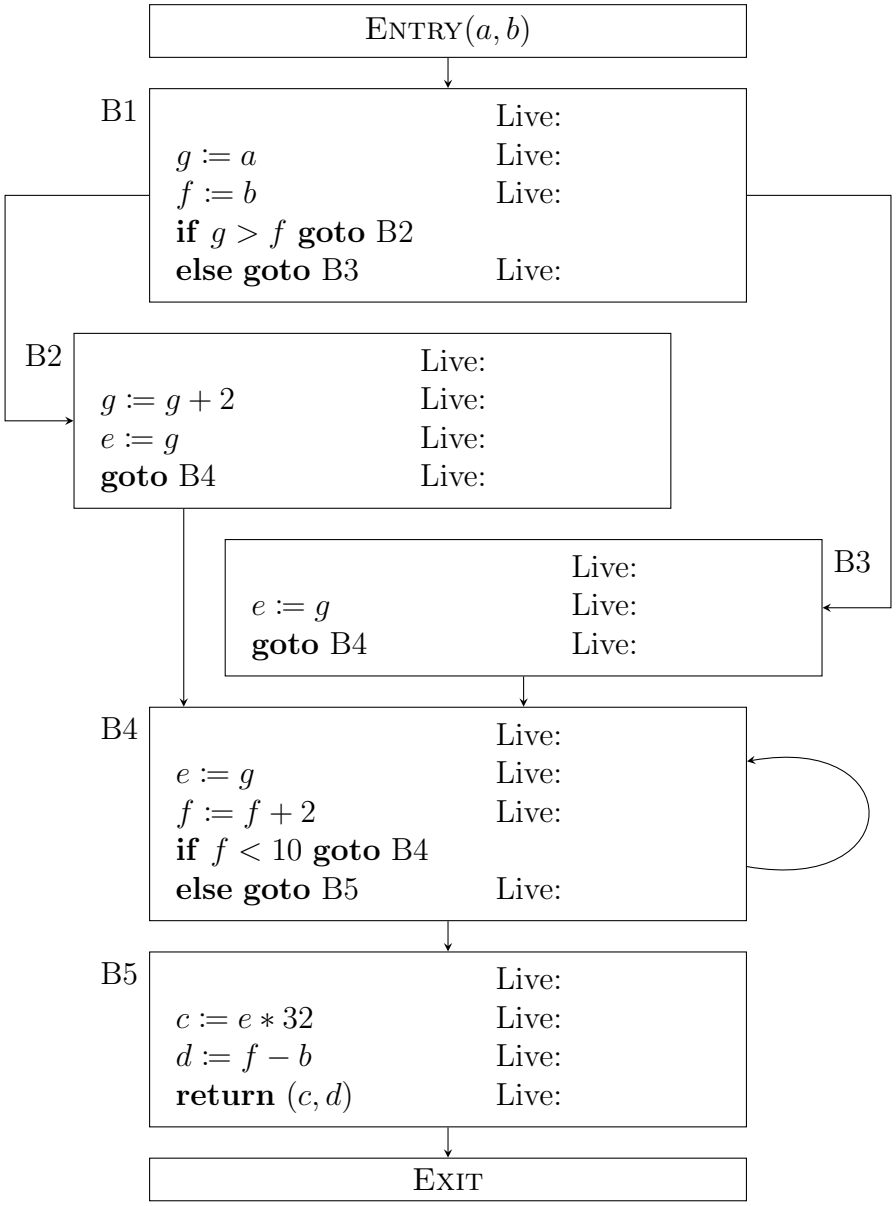
(Task 2.G) For the SSA IR you get in Task 2.F, eliminate as many ϕ nodes as possible using the method described above, and write down the simplified SSA IR you get. (Hint: it should contain exactly four ϕ nodes. Alternatively, you may also solve this problem by directly thinking about which four ϕ nodes you want to *keep* in the IR). For ease of grading, do not rename any variable that you did not eliminate: all the variables that survived your elimination should retain their original names as in Task 2.D.

Closing thoughts: while this problem shows you the definition of SSA and how you can convert a non-SSA IR to SSA form, the algorithm presented in this problem (as you did in Task 2.D) is far from optimal, both in terms of execution time and in terms of the total number of ϕ nodes inserted. In modern compilers, Cytron's fast SSA construction algorithm (discovered in 1989) is usually used, which runs significantly faster and inserts significantly fewer ϕ nodes for real-world input programs than our naive algorithm[1]: in fact, if the input program only uses structured control flow constructs (no `goto`/`break`/`continue`), Cytron's SSA construction algorithm is guaranteed to run in linear time with respect to program size (and inserts at most $O(n)$ ϕ-nodes as a direct consequence). The key towards the better algorithm is to formalize your observations in Task 2.G to figure out the criteria of when a ϕ node is truly needed, and use data structures to support fast detection of such conditions, but this is outside the scope of this problem set. If you are interested, we recommend you to take a look at the Wikipedia page for more information.

[1]Interestingly, you *can* construct pathetic inputs so that Cytron's algorithm degenerates to the same theoretical time complexity as the naive algorithm described in this problem. But for compilers, in most cases, it is the real-world performance on real-world inputs that matters, not the theoretical worst-case performance on artificial adversarial inputs.

Problem 3. Liveness and Register Interference Graph

(Task 3.A) Label each point in the following program with the set of live variables.



(Task 3.B) Show the register interference graph. For ease of grading, also include a list of edges in the form below:

```
a - b;
c - d;
```

(Task 3.C) Give a minimal coloring for the interference graph. You should not use the algorithm in class, as this is not guaranteed to give a minimal result.