Implementation of passes:

1. Available Expression:
    a. Domain: Set of “Expression” objects. The “Expression” class is defined by us to hold an operator code (of type “unsinged”), a left-hand side value (of type “Value *”) and a right-hand side value (of type “Value *”). For the interest of this assignment, only binary operations are recorded.
    b. Direction: Forward
    c. What’s stored in the “Instruction -> Bit Vector” Map: OUTPUT of all the instructions.
    d. Transfer Function: Simply add the use of any binary operation to the available expression set (by updating the Bit Vector), because no expression can be killed due to the single definition of variables in SSA.
    e. Meet Operation: Intersection, which is implemented as “AND” of all predecessors’ OUTPUT Bit Vectors.
    f. Boundary Condition: The OUTPUT of entry is set to empty set.
    g. Initial Interior Point Values: All are set to full set.
    h. Analysis Result: The content of the “Instruction -> Bit Vector” Map are printed, i.e. the OUTPUT of all the instructions.
2. Liveness:
    a. Domain: Set of variables, which is simply represented by “Value *”. Only instruction defined variables (i.e. of type “Instruction”) and function arguments (i.e. of type “Argument”) are of our interest.
    b. Direction: Backward
    c. What’s stored in the “Instruction -> Bit Vector” Map: INPUT of all the instructions except for the PHI instructions as they are not real executable instructions.
    d. Transfer Function: Add the used variables to the live variables set and remove the defined variable from the set. The defined variable is just the “Instruction” object itself. PHI instructions are dealt specially. Every incoming variable (i.e. operand) of a PHI instruction is marked as live at the end, which is the INPUT of the last instruction, of the predecessor it came from if it is not defined (i.e. killed) by that instruction. The reason why I did this is that every operand of a PHI instruction should be alive along the path it came from until it reaches the block of this PHI instruction.
    e. Meet Operation: Basically, union, which is implemented as “OR” of all successor’ INPUT Bit Vectors. However, if a successor has PHI instructions, I have to use the result given by the first non-PHI instruction as I chose not to store any result for PHI instructions. Moreover, for any successor that has PHI instructions at the beginning of it, I have to record the variables defined by the PHI instructions and exclude all these variables from the live variable set given by the first non-PHI instruction in that block, because they are really not live anymore above that block.
    f. Boundary Condition: The INPUT of exit is set to empty set.
    g. Initial Interior Point Values: All are set to empty set.

h. Analysis Result: The content of the “Instruction -> Bit Vector” Map are printed, i.e. the INPUT of all the instructions. “NOT AVAILABLE” is marked at PHI instructions.

3. Results on the benchmarks:

```
*****************************************
* Instruction-BitVector Mapping
*****************************************
Boundary Condition: {}
Instruction:   %3 = add nsw i32 %0, 50
        {[add %0, 50], }
Instruction:   %4 = add nsw i32 %3, 96
        {[add %3, 96], [add %0, 50], }
Instruction:   %5 = icmp slt i32 50, %3
        {[add %3, 96], [add %0, 50], }
Instruction:   br i1 %5, label %6, label %9
        {[add %3, 96], [add %0, 50], }
Instruction:   %7 = sub nsw i32 %3, 50
        {[add %3, 96], [sub %3, 50], [add %0, 50], }
Instruction:   %8 = mul nsw i32 96, %3
        {[add %3, 96], [sub %3, 50], [mul 96, %3], [add %0, 50], }
Instruction:   br label %12
        {[add %3, 96], [sub %3, 50], [mul 96, %3], [add %0, 50], }
Instruction:   %10 = add nsw i32 %3, 50
        {[add %3, 96], [add %0, 50], [add %3, 50], }
Instruction:   %11 = mul nsw i32 96, %3
        {[add %3, 96], [mul 96, %3], [add %0, 50], [add %3, 50], }
Instruction:   br label %12
        {[add %3, 96], [mul 96, %3], [add %0, 50], [add %3, 50], }
Instruction:   %.0 = phi i32 [ %7, %6 ], [ %10, %9 ]
        {[add %3, 96], [mul 96, %3], [add %0, 50], }
Instruction:   %13 = sub nsw i32 50, 96
        {[add %3, 96], [mul 96, %3], [add %0, 50], [sub 50, 96], }
Instruction:   %14 = add nsw i32 %13, %.0
        {[add %13, %.0], [add %3, 96], [mul 96, %3], [add %0, 50], [sub 50, 96], }
Instruction:   ret i32 0
        {[add %13, %.0], [add %3, 96], [mul 96, %3], [add %0, 50], [sub 50, 96], }
```

Above is the result for available expression analysis.

```
*********************************************
* Instruction-BitVector Mapping
*********************************************
        {%0, %1, }
Instruction:   br label %3
        NOT AVAILABLE
Instruction:   %.01 = phi i32 [ 1, %2 ], [ %6, %7 ]
        NOT AVAILABLE
Instruction:   %.0 = phi i32 [ %0, %2 ], [ %8, %7 ]
        {%.01, %1, %.0, }
Instruction:   %4 = icmp slt i32 %.0, %1
        {%.01, %1, %.0, %4, }
Instruction:   br i1 %4, label %5, label %9
        {%.01, %1, %.0, }
Instruction:   %6 = mul nsw i32 %.01, %.0
        {%6, %1, %.0, }
Instruction:   br label %7
        {%6, %1, %.0, }
Instruction:   %8 = add nsw i32 %.0, 1
        {%6, %1, %8, }
Instruction:   br label %3
        {%.01, }
Instruction:   ret i32 %.01
Boundary Condition: {}
```

Above is the result for live variable analysis.

CSC D70
A2
Wang Ming
Utorid: wangmi56
Student#: 1002105900

3.1

(1) Loop Invariant Instructions:
~~y=5, q=7, h=3, x=1, h=4, x=1, y=7, r=q+5~~

~~(3)~~ S2, S3, S5, S6, S8, S9, S12, S13

(2) S5, S6, S12, S13 : Cannot be moved to preheader because they do not dominate the exit block.

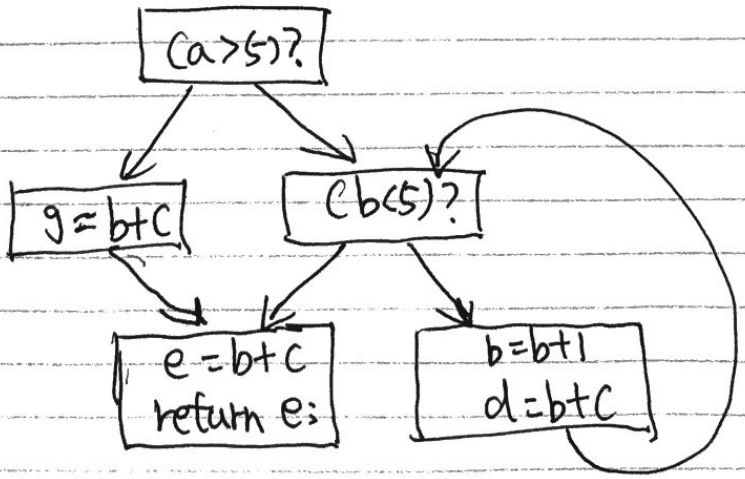
S2, S8, S9: Cannot be moved to preheader because the variables they assign to are assigned more than once. (S2 and S12 both assign to y, S8 and S5 both assign to h and S9 and S6 both assign to x)

S3: Can be moved to preheader because (1) it dominates the exit block, (2) it assign to q which is not assigned to anywhere else in the loop and (3) it dominates all uses — only S13.

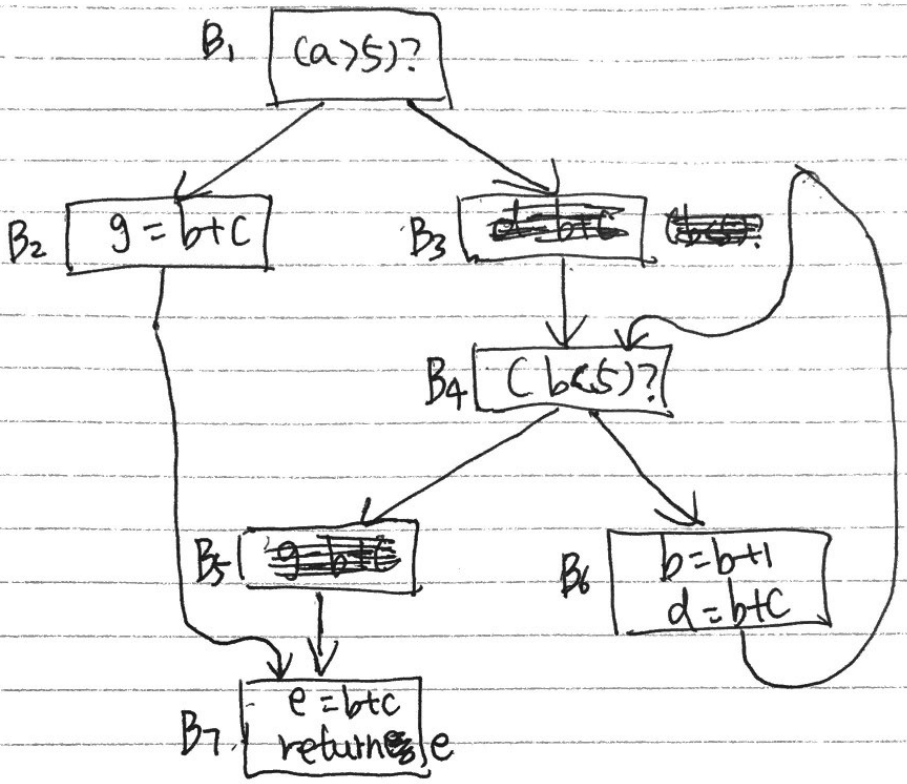
3.2

(1) Lazy Code Motion is the optimization used to remove fully or partially redundant expressions while postponing the placement of these solutions to avoid unnecessary registers' occupancy.

(2) CFG:



Remove the critical edges:



Anticipated Expressions:

B_1: {b+c, b<5, a>5}

B_2: {b+c}.

B_3: {b<5, ~~b+c~~}

B_4: {b<5, ~~b+c~~}

B_5: {b+c}

B_6: {b+1, b+c}

B_7: {b+c}

(3) Will-be-Available Expressions:

B_1: {b+c, b<5, a>5}

B_2: {b+c, b<5, a>5}

B_3: {b<5, ~~b+c~~, a>5}

B_4: {b<5, a>5}

B_5: {b+c, b<5, a>5}

B_6: {a>5}

B_7: {b+c, b<5, a>5}

Earliest Placement: b+1 and b+c at B_6.

(4) $B_1: \phi$, $B_2: \phi$, $B_3: \phi$, $B_4: \phi$, $B_5: \phi$, $B_7: \phi$.
$B_6: \{b+1, b+C\}$
Latest Placement: b+1 and b+C at B_6

(5) Nothing is changed in the original CFG ...?