ECE 468 & 573
Problem Set 3: Common sub-expression elimination, local register allocation.

For the following problems, consider the following piece of three-address code:

```
1.  READ(A)
2.  READ(B)
3.  C = A + B
4.  D = A + B
5.  E = C * D
6.  T1 = A + C
7.  T2 = T1 + E
8.  A = T2 + B
9.  F = A + B
10. G = C * D
11. T3 = F + G
12. WRITE(T3)
```

1. Show the result of performing Common Subexpression Elimination (CSE) on the above code. Rather than generating assembly, just give new 3AC. If you're reusing the results of an expression, just generate code that looks like X = Y, where Y is the variable/temporary that held the result of the original calculation.

   Answer:

   ```
   1.  READ(A)
   2.  READ(B)
   3.  C = A + B
   4.  D = C
   5.  E = C * D
   6.  T1 = A + C
   7.  T2 = T1 + E
   8.  A = T2 + B
   9.  F = A + B
   10. G = E
   11. T3 = F + G
   12. WRITE(T3)
   ```

2. Suppose A and C were aliased. How would that change the results of CSE?

   Answer: If A and C were aliased, the write to A in line 8 would kill any expression that uses C, including `C * D`. Thus, `C * D` would not be available in line 10, and we would have to recompute it, instead of using the old result.

3. For each instruction, show which variables are live *immediately after the instruction.* (Use the original code, not the CSE code)

   Answer: Live variables shown in brackets after instruction:

```
1.  READ(A) [A]
2.  READ(B) [A, B]
3.  C = A + B [A, B, C]
4.  D = A + B [A, B, C, D]
5.  E = C * D [A, B, C, D, E]
6.  T1 = A + C [B, C, D, E, T1]
7.  T2 = T1 + E [B, C, D, T2]
8.  A = T2 + B [A, B, C, D]
9.  F = A + B [C, D, F]
10. G = C * D [F, G]
11. T3 = F + G [T3]
12. WRITE(T3) [ ]
```

4. Assume that E is a global variable, and the given code is the code from a single function in a program with many functions. What changes in your liveness analysis?

   Answer: If E is a global variable, then we cannot safely assume that no variables are live at the end of the code (as we did above). Instead, we would have to assume that E is live at the end of the program, yielding the following live sets:

```
1.  READ(A) [A]
2.  READ(B) [A, B]
3.  C = A + B [A, B, C]
4.  D = A + B [A, B, C, D]
5.  E = C * D [A, B, C, D, E]
6.  T1 = A + C [B, C, D, E, T1]
7.  T2 = T1 + E [B, C, D, E, T2]
8.  A = T2 + B [A, B, C, D, E]
9.  F = A + B [C, D, E, F]
10. G = C * D [E, F, G]
11. T3 = F + G [E, T3]
12. WRITE(T3) [E]
```

5. (Go back to assuming the above code is the entire program) Perform bottom-up register allocation on the code for a machine with four registers. Show what code would be generated for each 3AC instruction. Use LOAD X Rx to load from a

variable/temporary into a register, STORE Rx X to store from a register into a variable/temporary, Rx = Ry + Rz for addition, and Rx = Ry * Rz for multiplication.

When choosing registers to allocate, always allocate the lowest-numbered register available. When choosing registers to spill, choose the non-dirty register that will be used farthest in the future. In case all registers are dirty, choose the register that will be used farthest in the future. In case of a tie, choose the lowest-numbered register.

Answer: For each 3AC instruction, I show the assembly generated, as well as the contents of each register at the end of the instruction. When registers are spilled or freed, I will annotate the code appropriately:

```
1.  READ(A) [A]
    READ R1
    //R1: A*

2.  READ(B) [A, B]
    READ R2
    //R1: A*, R2: B*

3.  C = A + B [A, B, C]
    R3 = R1 + R2
    //R1: A*, R2: B*, R3: C*

4.  D = A + B [A, B, C, D]
    R4 = R1 + R2
    //R1: A*, R2: B*, R3: C*, R4: D*

5.  E = C * D [A, B, C, D, E]
    //Need to find register for E. D will be used farthest
    ST R4, D //spill D to memory
    R4 = R3 * R4
    //R1: A*, R2: B*, R3: C*, R4: E*

6.  T1 = A + C [B, C, D, E, T1]
    //A is dead, so can free it. No need to store.
    R1 = R1 + R3
    //R1: T1*, R2: B*, R3: C*, R4: E*

7.  T2 = T1 + E [B, C, D, T2]
    //T1 & E are dead, so can be freed. No need to store.
    R1 = R1 + R4
```

```
    //R1: T2*, R2: B*, R3: C*

8.  A = T2 + B [A, B, C, D]
    //T2 is dead, so can be freed. No need to store.
    R1 = R1 + R2
    //R1: A*, R2: B*, R3: C*

9.  F = A + B [C, D, F]
    //A, B are dead, so can be freed. No need to store.
    R1 = R1 + R2
    //R1: F*, R3: C*

10. G = C * D [F, G]
    //Need to load D back into a register
    LD R2, D
    //C and D are dead, so can be freed. No need to store.
    R2 = R3 + R2
    //R1: F*, R2: G*

11. T3 = F + G [T3]
    //F and G are dead, so can be freed. No need to store.
    R1 = R1 + R2
    //R1: T3*

12. WRITE(T3) [ ]
    //T3 is dead, so can be freed. No need to store.
    WRITE R1
```

6. Repeat the process for *3* registers.

Answer:

```
1.  READ(A) [A]
    READ R1
    //R1: A*

2.  READ(B) [A, B]
    READ R2
    //R1: A*, R2: B*

3.  C = A + B [A, B, C]
```

```
    R3 = R1 + R2
    //R1: A*, R2: B*, R3: C*

4.  D = A + B [A, B, C, D]
    //Need to find register for D. B is used farthest
    ST R2, B //Spill B to memory
    R2 = R1 + R2
    //R1: A*, R2: D*, R3: C*

5.  E = C * D [A, B, C, D, E]
    //Need to find register for E. D is used farthest
    ST R2, D //Spill D to memory
    R2 = R3 * R2
    //R1: A*, R2: E*, R3: C*

6.  T1 = A + C [B, C, D, E, T1]
    //A is dead, can be freed. No need to store
    R1 = R1 + R3
    //R1: T1*, R2: E*, R3: C*

7.  T2 = T1 + E [B, C, D, T2]
    //T1, E are dead, can be freed. No need to store
    R1 = R1 + R2
    //R1: T2*, R3: C*

8.  A = T2 + B [A, B, C, D]
    //Load B back into register
    LOAD B, R2
    //T2 is dead, can be freed. No need to store
    R1 = R1 + R2
    //R1: A*, R2: B, R3: C*

9.  F = A + B [C, D, F]
    //A, B are dead, can be freed. No need to store
    R1 = R1 + R2
    //R1: F*, R3: C*

10. G = C * D [F, G]
    //Load D into register
    LOAD D, R2
    //C, D are dead, can be freed. No need to store
```

```
    R2 = R2 * R3
    //R1: F*, R2: G*

11. T3 = F + G [T3]
    //F, G are dead, can be freed. No need to store
    R1 = R1 + R2
    //R1: T3*

12. WRITE(T3) [ ]
    //T3 is dead, can be freed. No need to store
    WRITE R1
```

Pay close attention to what happened in instruction 10, where we loaded D into a register, but were then immediately able to free it for use by the destination operand of the instruction.

Note that when we had four registers, we only had to perform two spill operations (spilling D in instruction 5, loading it back in instruction 10), while when we had three registers, we had to perform 4 spill operations (spilling B in instruction 4, spilling D in instruction 5, loading B back in instruction 8, loading D back in instruction 10).