ECE 468 & 573

Problem Set 4: Common sub-expression elimination and local register allocation **Solutions**

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

```
1.  A = 7;
2.  B = A + 2;
3.  C = A + B;
4.  D = C + B;
5.  B = C + B;
6.  A = A + B;
7.  E = C + D;
8.  F = C + D;
9.  G = A + B;
10. H = E + F;
```

1. Show the result of performing Common Subexpression Elimination (CSE) on the above code.

   Answer: See below. Expressions available after each statement are given in braces.

   ```
   1.  A = 7;         {nothing available}
   2.  B = A + 2;     {A + 2}
   3.  C = A + B;     {A + 2, A + B}
   4.  D = C + B;     {A + 2, A + B, C + B}
   5.  B = D;         {A + 2}
   6.  A = A + B;     {nothing available}
   7.  E = C + D;     {C + D}
   8.  F = E;         {C + D}
   9.  G = A + B;     {C + D, A + B}
   10. H = E + F;     {C + D, A + B, E + F}
   ```

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

   Answer: If E and C were aliased, then `C + D` would not be available after statement 7, and we would not be able to perform CSE on statement 8.

3. In class, we discussed how aliasing might reduce the number of common subexpressions that we can eliminate. How might aliasing *increase* the amount of redundancy in the code. (hint: consider what would happen if B and D were aliased).

   Answer: Aliasing might increase redundancy because an expression that is looks

different than an available expression may actually be redundant. For example, if statement 9 were `G = C + B`, then normally there is no redundancy. But if B and D were aliased, then `C + B` is the same as `C + D`, and we could replace statement 9 with `G = E`.

4. For each instruction, show which variables are live *immediately after the instruction.*

   Answer: See below. Live variables are shown in braces

```
1.  A = 7;          {A}
2.  B = A + 2;      {A, B}
3.  C = A + B;      {A, B, C}
4.  D = C + B;      {A, B, C, D}
5.  B = C + B;      {A, B, C, D}
6.  A = A + B;      {A, B, C, D}
7.  E = C + D;      {E, A, B, C, D}
8.  F = C + D;      {E, F, A, B}
9.  G = A + B;      {E, F}
10. H = E + F;      {}
```

5. How many registers would be needed to perform register allocation with no spilling?

   Answer: 5: there are 5 live variables after instruction 7.

6. Top down register allocation is inefficient for the above code, as there are some variables that could safely be assigned to the same register. What are they?

   Answer: F and D are never live at the same time, and hence they could share the same register.

7. Perform bottom-up register allocation on the code for a machine with three registers. Show what code would be generated for each 3AC instruction. When choosing registers to allocate, always allocate the lowest-numbered register available. When choosing registers to spill, choose the register holding a value that will be used farthest in the future (in case of a tie, choose the lowest-numbered register).

   Answer:

Stmt	R1	R2	R3	Code
1	A			`R1 = 7`
2	A	B		`R2 = R1 + 2`
3	A	B	C	`R3 = R1 + R2`
4	D	B	C	`ST R1, A; R1 = R3 + R2`
5	D	B	C	`R2 = R3 + R2`
6	A	B	C	`ST R1, D; LD A, R1; R1 = R1 + R2`
7	D	E	C	`ST R1, A; LD D, R1; ST R2, B; R2 = R3 + R1`
8	F	E		`ST R3, C; R1 = R3 + R1`
9		E		`LD A, R3; ST R1, F; LD B, R1; R1 = R3 + R1; ST R1, G`
10				`LD F, R1; ST R2, E; R1 = R2 + R1; ST R1, H`

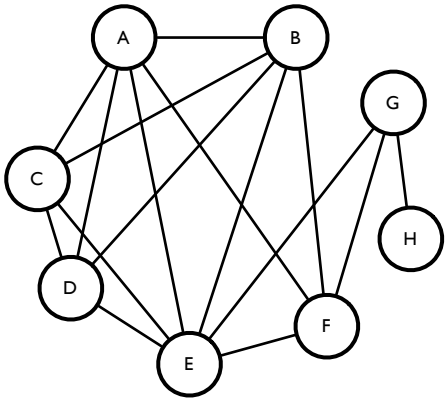
Note that at statement 8, D was not dirty, so no store was emitted. At Statements 9 and 10, G and H (respectively) were immediately dead, so they were immediately stored back and the registers were freed.

8. Draw the interference graph for the code.

   Answer: This can easily be derived from the liveness information in question 4.

9. (ECE 573 only) Perform register allocation via graph coloring for the code. If you need to spill, use the code-rewriting approach described in the notes.

   Answer: Note that if G and H are truly dead at the end of the basic block, there is no need to perform any of this code. We will assume that G and H are global variables, and hence live at the end of the basic block. I will be explicit about such constraints on any exam. The updated interference graph is below.



   In the first round of simplification, the stack is as follows (with $*$s marking variables that might be spilled). The rules that I'm using are: (i) remove nodes in reverse alphabetical order when possible; (ii) spill variables that are used the *least*, and when

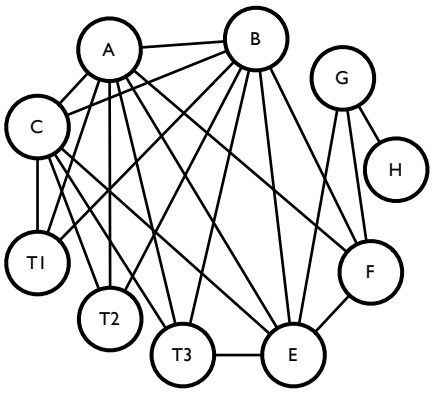
there's a tie, spill the variable with the most edges.

$$H, G, E*, F, D*, C, B, A.$$

When we try to recolor the graph, we find that D does, indeed, need to be spilled. This leads to the following rewritten code, with liveness information:

```
1.  A = 7;          {A}
2.  B = A + 2;      {A, B}
3.  C = A + B;      {A, B, C}
4.  T1 = C + B;     {A, B, C, T1}
4'. ST T1, D        {A, B, C}
5.  B = C + B;      {A, B, C}
6.  A = A + B;      {A, B, C}
6'. LD D, T2        {A, B, C, T2}
7.  E = C + T2;     {A, B, C, E}
7'. LD D, T3        {A, B, C, E, T3}
8.  F = C + T3;     {A, B, E, F}
9.  G = A + B;      {G, E, F}
10. H = E + F;      {G, H}
```

Which gives us the new interference graph:



In the second round of spilling, we remove the following. Note that spill temporaries cannot be spilled again:

$$H, G, E*, F, C*, T1, T2, T3, B, A$$

This time, C must be spilled, and the process continues as before.

Repeat steps 4, 7 and 8 for the following code (assume registers can hold either temporaries or variables):

```
1.  T1 = A + B;
2.  T2 = C + D;
3.  T3 = T1 - T2;
4.  T4 = C + T3;
5.  T5 = D + T3;
6.  T6 = T1 + T5;
7.  T7 = D + T6;
8.  B  = B + T7;
9.  T9 = A + T8;
10. A  = B + T9;
```