Name:_____

Final Examination
CS4430 Introduction to Compilers
University of Missouri
Spring 2017

Problem	Points Possible	Points Scored
1	25	
2	10	
3	25	
4	10	
5	10	
6	5	
7	10	
8	10	

1. **(25 total/12.5 each)** Design code translation schemes in the style we used in class for following expressions. For an example of what I mean by "translation scheme", see slide 21 of the "Semantic Analysis 2" slides.

   - Conditional expressions: **(e1 ? e2 : e3)**. This expression evaluates **e1** and, if it is **1**, it evaluates **e2** and, otherwise, it evaluates **e3**.

   - Parallel assignment: **(x1,x2,x3) = (e1,e2,e3)**. This expression performs the assignments **x1=e1**, **x2=e2**, and **x3=e3** in succession.

2. **(10 points)** Circle the best answer:

a. T or F Control may enter a basic block at more than one location.

b. T or F It is possible to implement graph-coloring register allocation in such a way that only one attempt at graph-coloring is needed.

c. T or F A compiler optimization always improves program performance.

d. T or F Register spilling can always be used to make an interference graph n-colorable, for arbitrary n.

e. Define *enabling* optimization.

3. **(25 points)**

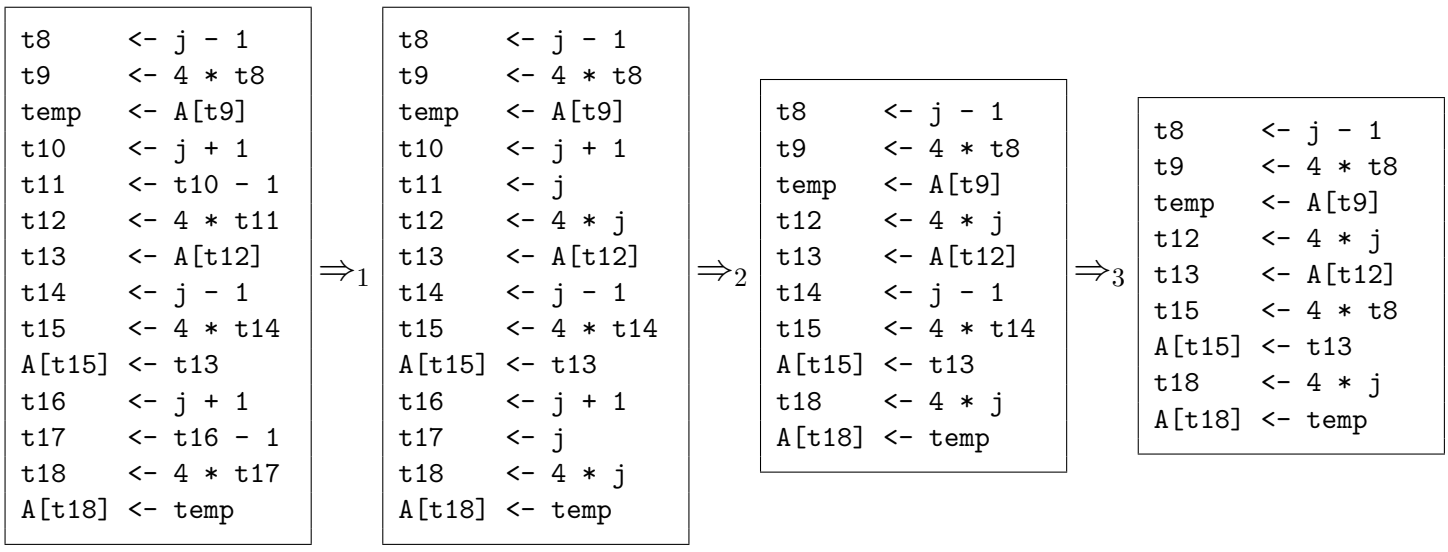
Here is a list of optimizations, which we discussed in class. We refer to this list in the following questions. Some of these optimizations are slightly different from those we discussed in class, in which case the difference is explained below.

- Algebraic simplification (AS). This is slightly more general that constant folding in that it can be applied to expressions that are not necessarily constant. E.g., "x + 0" can be algebraically simplified to "x", although x is not a constant. Note that every constant fold is an algebraic simplification.
- Common subexpression elimination (CSE). Given an expression that occurs more than once within a basic block, CSE stores that expression in a new temporary variable and replaces the expression with the temporary variable; e.g., CSE might replace the left with the right:

```
x <- foo(v * 3);                t <- v * 3;
if (v * 3 < 9) { ... }          x <- foo(t);
                                if (t < 9) { ... }
```

- Constant propagation (CP). Given an assignment of a constant to a variable "x = c" substitutes constant c for variable x wherever possible. Note that this transformation does not eliminate the instruction "x = c".
- Dead code elimination (DCE). *Dead code* is code that has no effect on the behavior of a program and, hence, may be safely eliminated. E.g., if `x` is never read, then the instruction `x <- 9` may be safely removed.
- Expression propagation (EP). Given an assignment a←e, we allow the entire expression e to be substituted for later uses of a, assuming neither a nor any of the temporaries occurring in e are assigned to again before the use. The code resulting from substituting e for a must be "algebraically simplifiable" to three address code. For e = "1 + y", "x←a + 1" may be transformed to "x←1 + y + 1" because it may be simplified to "x←y + 2", but "x←a + z" may not be transformed to "x←1 + y + z" because this is no longer three address code.

The graph below represents the evolution of a basic block through multiple optimizations. For each stage of the evolution (i.e., 1, 2 or 3) identify the optimizations applied. N.b., there may be more than one application per stage.



For stage 1, circle the answer that best describes the optimizations (1 point each):

AS was used:	CSE was used:	CP was used:	DCE was used:	EP was used:
(a) 0 times	(a) 0 times	(a) 0 times	(a) 0 times	(a) 0 times
(b) 1 times	(b) 1 times	(b) 1 times	(b) 1 times	(b) 1 times
(c) 2 times	(c) 2 times	(c) 2 times	(c) 2 times	(c) 2 times
(d) 3 times	(d) 3 times	(d) 3 times	(d) 3 times	(d) 3 times
(e) 4 times	(e) 4 times	(e) 4 times	(e) 4 times	(e) 4 times

For stage 2, circle the answer that best describes the optimizations (1 point each):

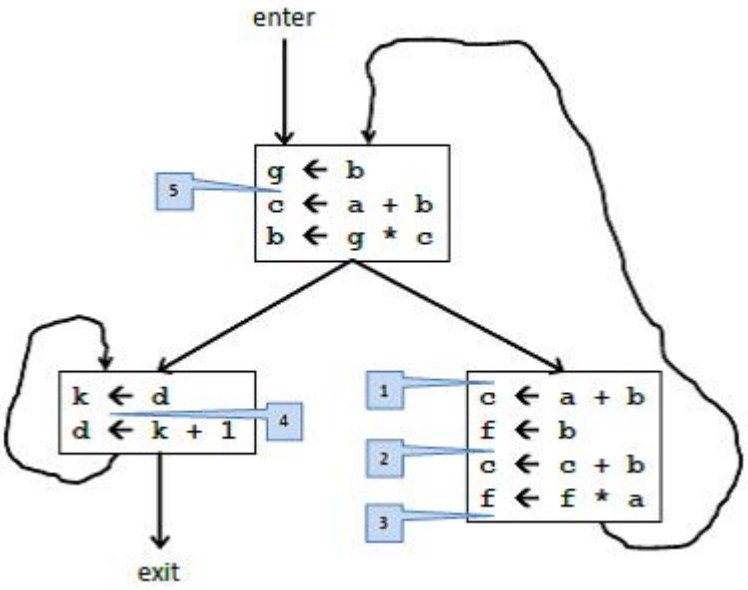
AS was used:	CSE was used:	CP was used:	DCE was used:	EP was used:
(a) 0 times	(a) 0 times	(a) 0 times	(a) 0 times	(a) 0 times
(b) 1 times	(b) 1 times	(b) 1 times	(b) 1 times	(b) 1 times
(c) 2 times	(c) 2 times	(c) 2 times	(c) 2 times	(c) 2 times
(d) 3 times	(d) 3 times	(d) 3 times	(d) 3 times	(d) 3 times
(e) 4 times	(e) 4 times	(e) 4 times	(e) 4 times	(e) 4 times

For stage 3, circle the answer that best describes the optimizations (1 point each):

AS was used:	CSE was used:	CP was used:	DCE was used:	EP was used:
(a) 0 times	(a) 0 times	(a) 0 times	(a) 0 times	(a) 0 times
(b) 1 times	(b) 1 times	(b) 1 times	(b) 1 times	(b) 1 times
(c) 2 times	(c) 2 times	(c) 2 times	(c) 2 times	(c) 2 times
(d) 3 times	(d) 3 times	(d) 3 times	(d) 3 times	(d) 3 times
(e) 4 times	(e) 4 times	(e) 4 times	(e) 4 times	(e) 4 times

4. **(10 points total)**.

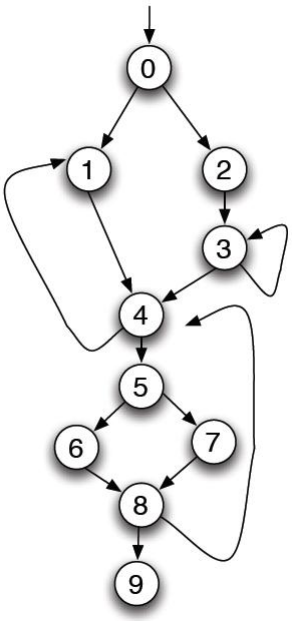
There are five program points, labeled 1-5, in the graph below. Assume that there are no live variables on exit.



Fill in the following table writing "L" for live and "D" for dead for each program point/variable combination.

Program Point	a	b	c	d	e	f	g	k
1								
2								
3								
4								
5								

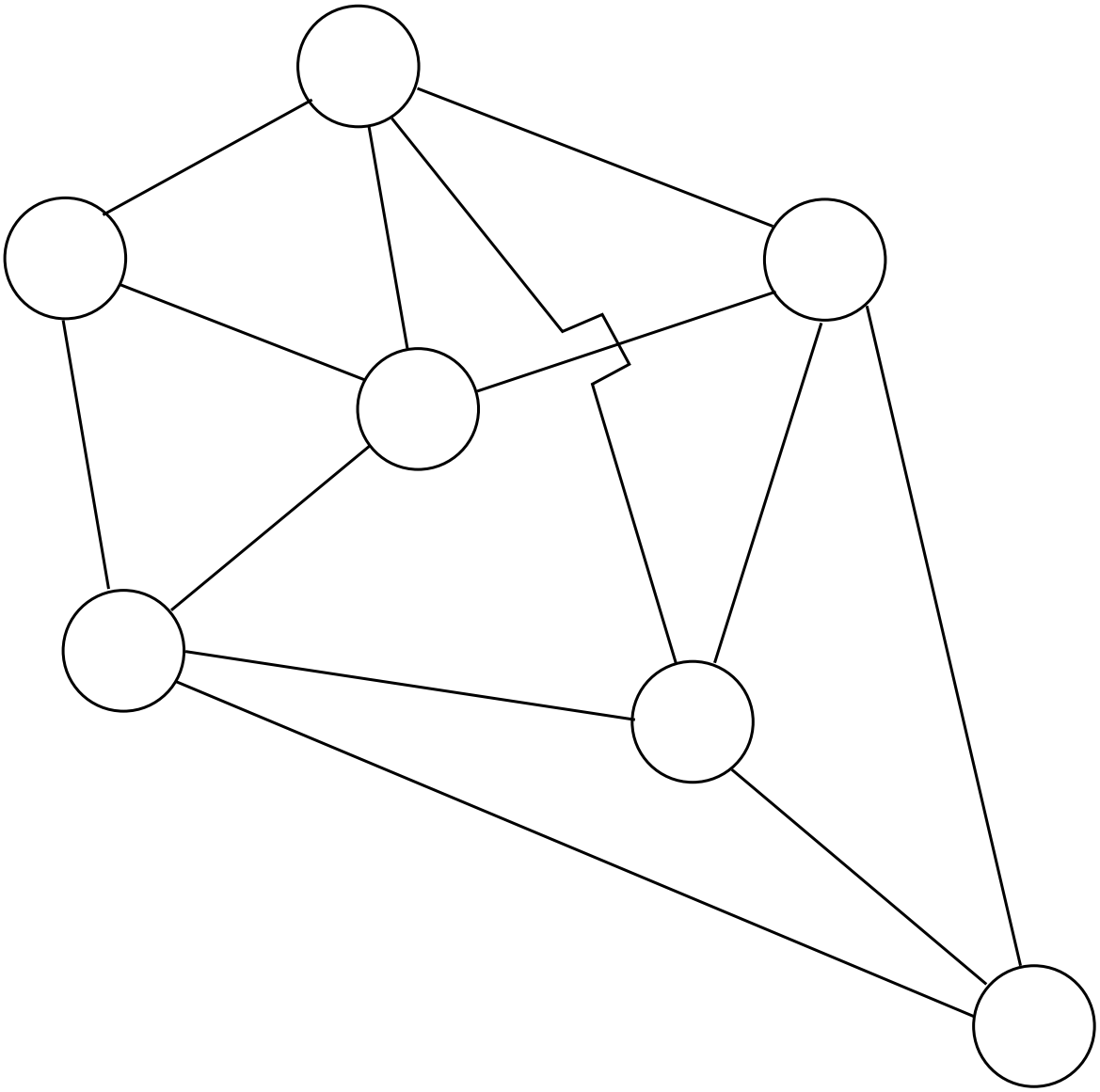
5. **(10 points)** Consider the following control flow graph. A node d dominates a node n if every path from the start node to n must go through d. By definition, every node dominates itself. The dominator set of a node is the set of nodes it dominates in the control flow graph.



Fill in the following table.

Node	Dominator Set
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

6. **(5 points)**. If possible, 4-color the following interference graph.



7. **10 points total**.

Translate the following loop into three-address code.

```
for i=j to 2*n step m+k do
    s := s+i
```

8. **10 points total**.

The following is a context-free grammar:

```
L -> ident
L -> ( L L )
L -> ( lambda ( ident ) L )
```

Answer the following questions:

a. True or False: the above grammar is ambiguous (*1 point*).

b. (*1 point*) List the non-terminal symbols.

c. (*1 point*) List the terminal symbols.

d. (*7 points*) Show a derivation of: `( (lambda(x) x) (u v) )`