CSE 401/M501 21au Final Exam Sample Solution

Question 1. (20 points) A bit of x86-64 coding. Here is a small C function that has two integer parameters and returns a value computed from those parameters. Function `rand()` is an external function that returns some random integer.

```
extern int rand();

// return abs(a*some_random_int + b)
int rando(int a, int b) {
  int ans;
  ans = a*rand()+b;
  if (ans < 0) {
    ans = -ans;
  }
  return ans;
}
```

On the next page, translate this function into x86-64 assembly language. You should use the standard x86-64 runtime conventions for parameter passing, register usage, and so forth that we used in the MiniJava project, including using `%rbp` as a stack frame pointer in the function prologue code. Note that this is simple C code, not a Java method, so there is no `this` pointer or method vtable involved.

Reference and ground rules for x86-64 code, (same as for the MiniJava project and other x86-64 code):

- All values, including pointers and `int`s, are 64 bits (8 bytes) each, as in MiniJava
- You must use the Linux/gcc assembly language, and must follow the x86-64 function call, register, and stack frame conventions:
  - Argument registers: `%rdi, %rsi, %rdx, %rcx, %r8, %r9` in that order
  - Called function must save and restore `%rbx, %rbp`, and `%r12-%r15` if these are used in the function
  - Function result returned in `%rax`
  - `%rsp` must be aligned on a 16-byte boundary when a `call` instruction is executed
  - `%rbp` must be used as the base pointer (frame pointer) register for this question
- The full form of a memory address is *constant*(*%rbase*,*%rindex*,*scalefactor*), which references memory address *%rbase*+*%rindex***scalefactor*+*constant*. *scalefactor* must be 0, 2, 4, or 8.
- Your x86-64 code must implement all of the statements in the original function. You may *not* rewrite the code into a different form that produces equivalent results (i.e., restructuring or reordering the code or eliminating function calls). You should allocate space for the local variable `ans` and generate appropriate load and store instructions when needed so that the copy in memory always has the correct current value of that variable at the end of the x86-64 code for each statement in the original C function. (And, of course, you can allocate additional space in the stack frame as needed to store other data.). Other than that, you can use any reasonable x86-64 code that follows the standard function call and register conventions – you do not need to mimic the code produced by your MiniJava compiler.
- Please include *brief* comments in your code to help us understand what the code is supposed to be doing (which will help us assign partial credit if it doesn't do exactly what you intended.)

Question 1. (cont.) Write your x86-64 translation of function `rando` below. Remember to read and follow the above ground rules carefully. Brief comments are appreciated. Original code repeated below for convenience:

```
// return abs(a*some_random_int + b)
int rando(int a, int b) {
  int ans;
  ans = a*rand()+b;
  if (ans < 0) {
    ans = -ans;
  }
  return ans;
}
```

There are obviously many possible solutions. Here is a fairly straightforward one.

```
rando:  pushq   %rbp              ; function prologue
        movq    %rsp,%rbp
        subq    $32,%rsp          ; alloc space for ans + parameters
        movq    %rdi,-8(%rbp)     ; save a
        movq    %rsi,-16(%rbp)    ; save b
        call    rand              ; rand() result in %rax
        movq    -8(%rbp),%rdi     ; reload a, b
        movq    -16(%rbp),$rsi
        imulq   %rdi,%rax         ; a*rand()
        addq    %rsi,%rax         ; a*rand()+b (sets cond codes)
        movq    %rax,-24(%rbp)    ; store in ans
        jge     nonneg            ; jump if cond codes show ans >= 0
        negq    %rax              ; %rax = -%rax
        movq    %rax,-24(%rbp)    ; update ans
nonneg: movq    %rbp,%rsp         ; return with result in %rax
        popq    %rbp              ; (leave ok instead of movq/popq
        ret
```

Notes:

- **Must save argument registers in memory and reload after calling `rand` since any function call may potentially clobber argument and other not-guaranteed-to-be-preserved registers.**
- **This solution reloads the saved argument values back into the registers `%rdi` and `%rsi` after the function call, but the values could have been re-loaded into different registers or even used directly from memory for arithmetic.**
- **This solution updates `ans` in memory as required but does not reload it when it is already present in a register. Solutions that reloaded it from memory as needed are also fine.**
- **The `addq` instruction sets the condition codes so it is not necessary to re-test `ans` before the conditional jump, but most solutions had a test or compare instruction to check whether the value is 0, which is fine.**
- **This solution computes `-ans` using a `negq` instruction, but solutions that subtract the original value from 0 or compute the value correctly in other ways are also fine.**

Question 2. (20 points) Compiler hacking. As is often the case, now that our MiniJava compiler is done, one of our customers would like us to add a new "feature". This customer learned Ruby as a youth and really likes Ruby's ability to turn any statement into a conditional (if) statement. In Ruby, a statement *S* can often be followed by "`if` *condition*", in which case the condition is evaluated first and the statement *S* is only executed if the condition is true.

Our customer would like to add a similar `do-if` statement to MiniJava. An example of the new statement is the following:

```
do max = x; if x < max
```

This new statement has exactly the same meaning and semantics as this ordinary Java `if` statement with no else part: `if (x < max) max = x;` . Notice that the **;** (semicolon) in the code examples here is part of the assignment statement grammar rule, not part of either the new `do-if` statement or the traditional Java `if` statement. Also, even though the keyword `do` is used here, this is *not* a loop, just an unusual conditional statement.

Answer the questions below about how this new `do-if` statement would be added to a MiniJava compiler as a new statement (i.e., a new regular statement added to the language along with the previously existing statements in MiniJava). There is likely way more space than you will need for some of the answers. The full MiniJava grammar is attached at the end of the exam if you need to refer to it.

(a) (2 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add this new `do-if` statement to the original MiniJava language? Just describe any new token name(s) needed. If any other changes to the lexical tokens in MiniJava are needed, you should also describe those changes here. You don't need to give JFlex or CUP specifications or code.

Add a new lexical token DO for the new do keyword

(continued on next page)

Question 2. (cont.) (b) (4 points). Give the context-free grammar rule or rules needed to add this new `do-if` statement to the MiniJava grammar. You only need to give the additions and changes needed to the MiniJava grammar. You do not need to write CUP specifications or other MiniJava code, and do not need to worry about any possible shift-reduce or reduce-reduce conflicts these changes might introduce into the existing MiniJava grammar. (Recall that there is a copy of the MiniJava grammar at the end of the exam that you can refer to.)

Add a new production to the grammar for this new statement type:

```
Statement :: = …
             | "do" Statement "if" Expression
```

(c) (4 points) Describe the changes or additions that need to be made to the MiniJava Abstract Syntax Tree (AST) classes (i.e., AST nodes) to add this new `do-if` statement to the compiler. You should not include specific Java code or AST class definitions, but you should precisely describe the new or changed node types and their contents so that it is obvious how they would be implemented.

Create a new `DoIf` node type (a subclass of the AST `Statement` class) with two fields: an `Exp e` for the condition expression subtree of the do-if statement, and a `Statement s` field for the statement that is to be conditionally executed.

Note: this should be done by creating a new kind of AST node rather than doing something like using the existing `If` node and setting the second Statement field in that node to `null`. The rest of the compiler logic assumes that `If` nodes have two non-null `Statement` children based on the MiniJava language specification, so if we were to use existing `If` nodes for this new statement it would require modifications elsewhere in the compiler to handle those nodes in different ways, and it would also obscure the fact that this really is a new, different kind of MiniJava statement.

(continued on next page)

Question 2. (cont.) (d) (4 points) What additions or changes need to be made to the static semantics / type checking part of the compiler to verify that a new `do-if` statement is correct? Again, you don't need to provide specific visitor method code or anything like that – just describe what type or other information needs to be produced or checked for this new statement.

Verify that the `Exp` subtree in the new `DoIf` AST node has type `Boolean`.

(e) (6 points) Describe the x86-64 code shape for this new `do-if` statement that would be generated by a MiniJava compiler. Your answer should be similar in format to the descriptions we used in class for other language constructs.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. However, remember that the question is asking for the code shape for this statement, so using things like J_{false}, for example, to indicate control flow, instead of pure x86-64 machine instructions, is fine as long as the meaning is clear. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them.

```
        <code evaluating expression>
        Jfalse skip
        <code for statement>
skip:
```

Question 3. (15 points) A little optimization. For this question we'd like to perform local constant propagation and folding (compile-time arithmetic), plus copy propagation (reuse values that are already present in another temporary t*i* when possible), strength reduction (replace expensive operations like * with cheaper ones when possible), common subexpression elimination, and dead code elimination.

The left column of the table below gives the three-address code generated for this statement:
a[i] = a[i]+a[i-1]

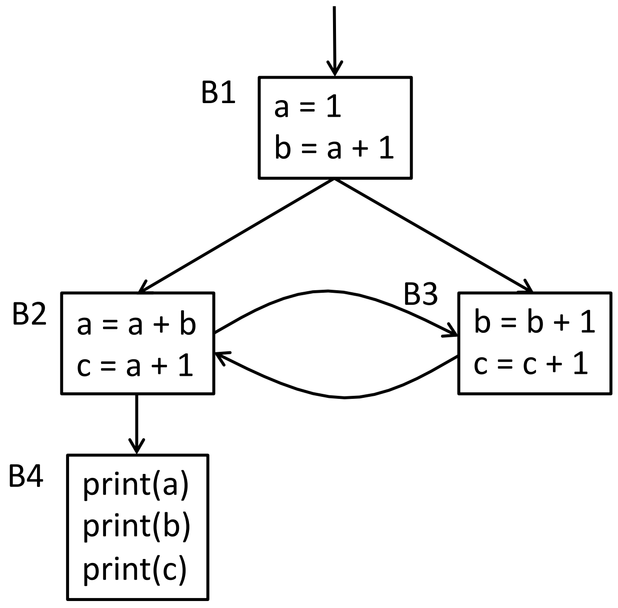
(a) Fill in the middle column with the code from the left column after any changes due to constant propagation and folding, copy propagation, strength reduction, and common subexpression elimination, but before any dead code elimination. (Notes: memory reference addresses can use a register (t*i* or `fp`) and a constant offset only – they cannot be more complex. Also note that the array `a` is assumed to be a local value stored in the current stack frame, as is possible in C or C++ code, instead of being a pointer to an array allocated elsewhere in memory as in Java.)

(b) In the last column, put an X under "deleted" if the statement would be deleted by dead code elimination after performing the constant propagation/folding, copy, and strength reduction optimizations in part (a).

Some solutions included more aggressive optimizations based on a bit of algebra. The most common one was to replace h with t8=t3-8, in which case g then becomes dead code. Solutions that did this were fine, but there was no deduction for solutions that only did the requested transformations.

	Original Code	After constant prop./folding & copy prop., strength reduction, and CSE (copy original code if no change)	"X" if deleted as dead code
a	t1 = *(fp + ioffset) // i	t1 = *(fp + ioffset) // i	
b	t2 = t1 * 8 // i*8	**t2 = t1 << 3 // i*8**	
c	t3 = fp + t2	t3 = fp + t2	
d	t4 = *(t3 + aoffset) // a[i]	t4 = *(t3 + aoffset) // a[i]	
e	t5 = *(fp + ioffset) // i	**t5 = t1 // i**	**X**
f	t6 = t5 * 8 // i*8	**t6 = t2 // i*8**	**X**
g	t7 = t6 - 8 // (i-1)*8	**t7 = t2 - 8 // (i-1)*8**	
h	t8 = fp + t7	t8 = fp + t7	
i	t9 = *(t8 + aoffset) // a[i-1]	t9 = *(t8 + aoffset) // a[i-1]	
j	t10 = t4 + t9 // a[i] + a[i-1]	t10 = t4 + t9 // a[i] + a[i-1]	
k	t11 = *(fp + ioffset)	**t11 = t1**	**X**
l	t12 = t11 * 8	**t12 = t2**	**X**
m	t13 = fp + t12	**t13 = t3**	**X**
n	*(t13 + aoffset) = t10 // a[i] = …	***(t3 + aoffset) = t10 // a[i] = …**	

The next two questions concern the following rather unusual control flow graph:



Question 4. (20 points) Dataflow – live variables. Recall from lecture that *live-variable* analysis determines for each point *p* in a program which variables are live at that point. A live variable *v* at point *p* is one where there exists a path from point *p* to another point *q* where *v* is used without *v* being redefined anywhere along that path. The sets for the live variable dataflow problem are:

use[*b*] = variables used in block *b* before any definition
def[*b*] = variables defined in block *b* and not later killed in *b*
in[*b*] = variables live on entry to block *b*
out[*b*] = variables live on exit from block *b*

The dataflow equations for live variables are

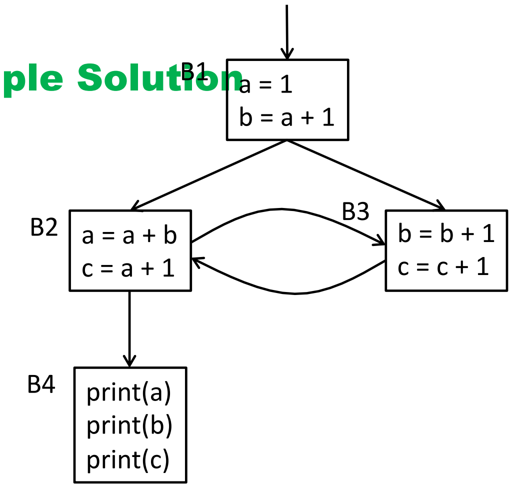
in[*b*] = use[*b*] ∪ (out[*b*] – def[*b*])
out[*b*] = $\cup_{s \in \text{succ}[b]}$ in[*s*]

On the next page, calculate the use and def sets for each block, then solve for the in and out sets of each block. A table is provided with room for the use and def sets for each block and up to three iterations of the main algorithm to solve for the in and out sets. If the algorithm does not converge after three iterations, use additional space below the table for additional iterations.

Then remember to answer the undefined variable question at the bottom of the page.

Hint: remember that live-variables is a backwards dataflow problem, so the algorithm should update the sets from the end of the flowgraph towards the beginning to reduce the total amount of work needed.

CSE 401/M501 21au Final Exam Sample Solution



Question 4. (cont). Graph and definitions repeated from previous page:

use[*b*] = variables used in block *b* before any definition
def[*b*] = variables defined in block *b* and not later killed in *b*
in[*b*] = variables live on entry to block *b*
out[*b*] = variables live on exit from block *b*

in[*b*] = use[*b*] ∪ (out[*b*] – def[*b*])
out[*b*] = ∪ $_{s \in \text{succ}[b]}$ in[*s*]

(a) (18 points) Write the results of calculations for live variables in the chart below. Use the rest of the page for extra space if needed, then remember to answer part (b) at the bottom! (Note: you should write down the details of the individual passes so if there is an error in the final result we will be better able to evaluate what happened.)

Block	use	def	out (1)	in (1)	out (2)	in (2)	out (3)	in (3)
B4	**a, b, c**	**---**	**---**	**a, b, c**	**---**	**a, b, c**		
B3	**b, c**	**b, c**	**---**	**b, c**	**a, b**	**a, b, c**	**no changes**	
B2	**a, b**	**a, c**	**a, b, c**	**a, b**	**a, b, c**	**a, b**		
B1	**---**	**a, b**	**a, b, c**	**c**	**a, b, c**	**c**		

Note: solutions that processed the blocks in different orders (for example B2 before B3 in the first iteration) received full credit as long as the final result was correct.

(b) (2 points) One use of live variable analysis is to detect potential use of uninitialized variables in a program. Are there any potentially uninitialized variables in this flowgraph, and if so which ones, where, and why? (i.e., justify your answer using the information from the analysis you have done above.)

c is used as an uninitialized variable somewhere in the flowgraph because it is live-in to B1. (The live variable analysis is not more precise than that but the actual uninitialized use is in B3 if B3 is executed before executing B2.)

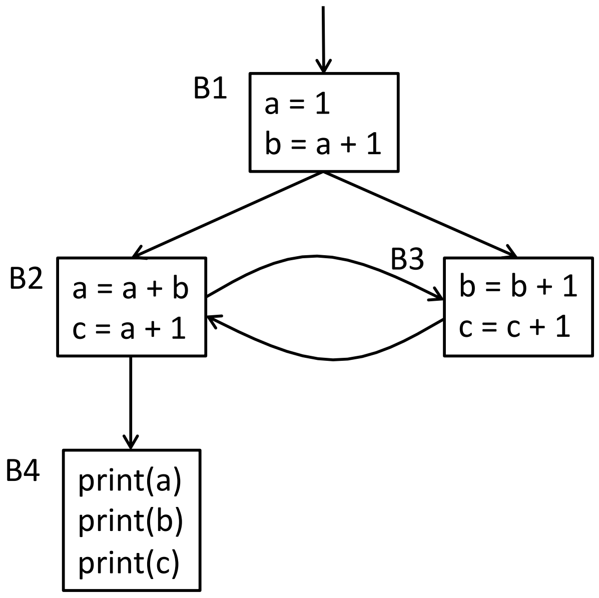
Question 5. (20 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

- Every control flow graph has a unique **start node** s.
- Node *x* **dominates** node *y* if every path from s to *y* must go through x.
    - A node *x* dominates itself.
- A node *x* **strictly dominates** node *y* if *x* dominates *y* and *x* ≠ *y*.
- The **dominator set** of a node *y* is the set of all nodes *x* that dominate *y*.
- An **immediate dominator** of a node *y*, idom(*y*), has the following properties:
    - idom(*y*) strictly dominates *y* (i.e., dominates *y* but is different from *y*)
    - idom(*y*) does not dominate any other strict dominator of *y*

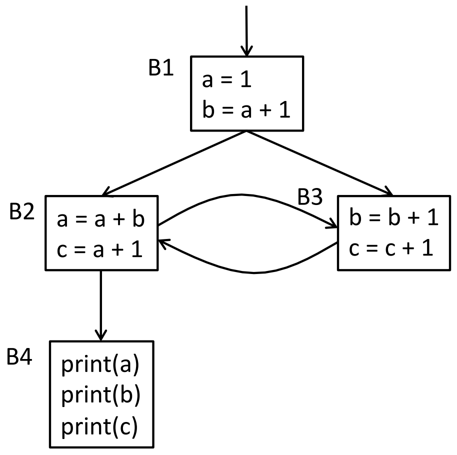
  A node might not have an immediate dominator. A node has at most one immediate dominator.
- The **dominator tree** of a control flow graph is a tree where there is an edge from every node *x* to its immediate dominator idom(*x*).
- The **dominance frontier** of a node *x* is the set of all nodes *w* such that
    - *x* dominates a predecessor of *w*, but
    - *x* does not strictly dominate *w*

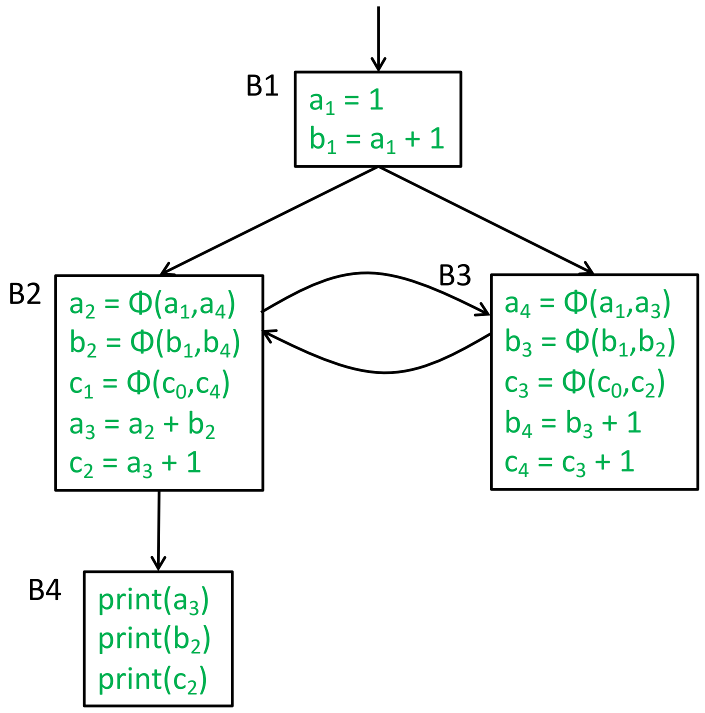
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it strictly dominates and the nodes that are in its dominance frontier (if any):

Node	Strictly dominates	Dominance Frontier
B1	**B2, B3, B4**	**---**
B2	**B4**	**B3**
B3	**---**	**B2**
B4	**---**	**---**



(b) (12 points) Now redraw the flowgraph in SSA (static single-assignment) form. You need to insert all Φ-functions that are required by the dominance frontier criteria, even if some of the variables created by those functions are not used later. Once that is done, add appropriate version numbers to all variables that are assigned in the flowgraph. You do not need to trace the steps of any particular algorithm to place the Φ-functions as long as you add them to the flowgraph in appropriate places. Answers that have a couple of extraneous Φ-functions will receive appropriate partial credit, but answers that, for example, use a maximal-SSA strategy of placing Φ-functions for all variables at the beginning of every block will not be looked on with favor.





The last question concerns register allocation. Assume that we're using the same hypothetical machine that was presented in lecture and in some textbook examples.

Our instruction selection algorithm has been modified so it does not re-use registers, but instead just creates temporaries and leaves register selection for later. Given the statement ans=a*x*x + b*x +c; here's what the instruction selector generated:

```
a.  LOAD    t1 <- a            // t1 = a
b.  LOAD    t2 <- x            // t2 = x
c.  MULT    t3 <- t1, t2       // t3 = a*x
d.  MULT    t4 <- t2, t3       // t4 = a*x*x
e.  LOAD    t5 <- b            // t5 = b
f.  MULT    t6 <- t5, t2       // t6 = b * x
g.  ADD     t7 <- t4, t6       // t7 = a*x*x + b *x
h.  LOAD    t8 <- c            // t8 = c
i.  ADD     t9 <- t7, t8       // t9 = a*x*x + b*x +c
j.  STORE   ans <- t9          // store ans
```

In a real compiler we would first use list scheduling to pick a (possibly) better order for the instructions, then use graph coloring to assign temporaries (t1-t9) to actual registers. But for this question we're only concerned with register allocation so we'll assume that this is the fixed, final order of the instructions.

Answer the questions

about this sequence of code

on

the

next

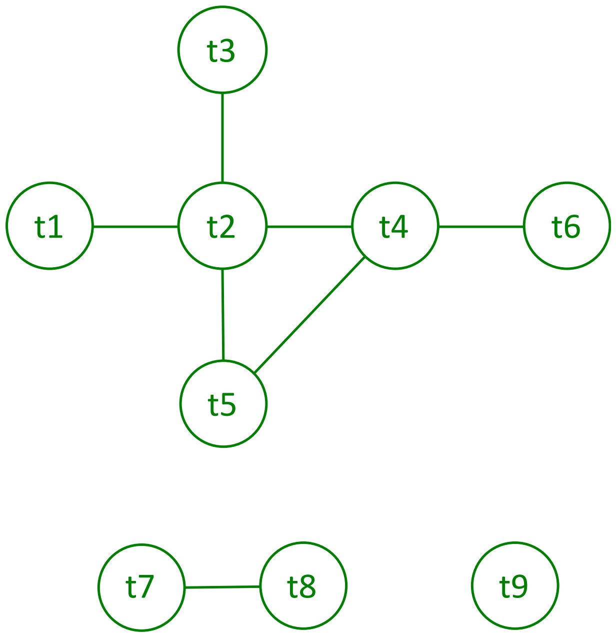
page…..

CSE 401/M501 21au Final Exam **Sample Solu**

```
a.  LOAD    t1 <- a
b.  LOAD    t2 <- x
c.  MULT    t3 <- t1, t2
d.  MULT    t4 <- t2, t3
e.  LOAD    t5 <- b
f.  MULT    t6 <- t5, t2
g.  ADD     t7 <- t4, t6
h.  LOAD    t8 <- c
i.  ADD     t9 <- t7, t8
j.  STORE   ans <- t9
```

Question 6. (15 points) Register allocation/graph coloring.

(a) (10 points) Draw the interference graph for the temporary variables (t1-t9) in this code. You should assume that the code is executed in the sequence given and not rearranged before assigning registers.



Note that in a statement like MULT t3<-t1,t2, the operand values are read from t1 and t2 before the result is stored in t3. This statement does not in itself produce a conflict between t1 and t3 or t2 and t3. Of course, if the lifetimes of t1 and t3 or t2 and t3 overlap for other reasons then there would be edges in the graph to reflect this, as is the case here between t2 and t3.

(b) (5 points) Give an assignment of groups of temporary variables to registers that uses the minimum number of registers possible based on the information in the interference graph. Use R1, R2, R3, … for the register names.

Three registers are needed. Here is one possible assignment:

- **R1: t2, t6, t7, t9**
- **R2: t1, t3, t4, t8**
- **R3: t5**

Have a great holiday break and best wishes for the new year!
The CSE 401/M501/P501 staff