CSE 401/M501 24au Final Exam 12/10/24 Sample Solution

A warmup question to get started…

Question 1. (10 points) Compiler phases. For each of the following situations, indicate where the situation would normally be discovered or handled in a production compiler. Assume that the compiler is a conventional one that generates native code for a single target machine (say, x86-64), and assume that the source language is standard Java (unless stated otherwise). Use the following abbreviations for the stages:

scan – scanner
parse – parser
sem – semantics/type check
opt – optimization (dataflow/ssa analysis; code transformations)
instr – instruction selection & scheduling
reg – register allocation
run – runtime (i.e., when the compiled code is executed)
can't – can't always be done during either compilation or execution

**parse** Report that `+=` is not a valid MiniJava operator

**scan** Report that `/` is not a valid MiniJava operator

**run** Select the correct version of a method to run based on whether the method is inherited or overridden in the source program **(This is runtime dynamic dispatch / vtables)**

**reg** Add extra instructions to the program to save register contents temporarily to memory and reload later if there are not enough registers to hold all active values

**opt** Replace a call to a `setX` method that updates an instance variable `x` with a copy of the `setX` method body, avoiding the overhead of a method call to change the value of `x`

**sem** Verify that the numbers and types of the parameters in a method call are compatible with the argument list in at least one overloaded version of the method that is visible at the call location

**sem** Verify and enforce that `private` members of a class are inaccessible outside of that class

**run** Report that execution of the expression `new int[n]` fails because the value of variable `n` is a negative number

**instr** Given several possible target machine instructions that set the value of a variable to 0, pick the one that minimizes the size in bytes of the generated code

**sem** Report that a variable name is incorrectly declared twice as a local variable in the same method

Question 2. (8 points) Declarations and scope. Here is a peculiar Java program consisting of a main class named `Mystery` and two subclasses.

```
public class Mystery {
    public static void main(String[] args) {
        Subtractor s = new Subtractor();
        Adder a = s;

        a.setX(2);
        a.setY(1);
        System.out.println(a.add());
        s.setX(5);
        System.out.println(s.update(3));
        System.out.println(s.add());
        System.out.println(s.sub());
    }
}

class Adder {
    int x;
    int y;

    int add() { return x + y; }
    void setX(int i) { x = i; }
    void setY(int i) { y = i; }
}

class Subtractor extends Adder {
    int x;

    int sub() { return x-y; }

    int update(int i) {
        x = i;
        return x;
    }
}
```

What output does this program produce when we compile it and then execute the `main` method in class `Mystery`? (The program does compile and execute without errors.)

3
3
6
2

Question 3. (15 points) A bit of x86-64 coding. Here is a small C function that calculates a value and returns 1 or 0 depending on whether the value is greater than 100 or not.

```
int isWinner(int points, int mult) {
  int score;
  score = 20 + points * mult;
  if (score > 100) {
    return 1;
  }
  return 0;
}
```

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 including allocating space for and using the local variable `score`, specifically updating the memory location allocated for `score` every time `score` is updated in the original code. 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, if any). 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 3. (cont.) Write your x86-64 translation of function `isWinner` below. Remember to read and follow the above ground rules carefully, including managing registers properly and using the correct argument registers for function calls, and creating a local stack frame to hold the variable `score`, properly updated as it is changed. Your code should include a translation of all of the code in the original function. Brief comments are appreciated. Original code repeated below for convenience:

```
int isWinner(int points, int mult) {
  int score;
  score = 20 + points * mult;
  if (score > 100) {
    return 1;
  }
  return 0;
}
```

```
isWinner:
   pushq %rbp             # prologue
   movq  %rsp, %rbp
   subq  $16, %rsp        # allocate stack frame (16-byte aligned)
   imulq %rsi, %rdi       # %rdi = points*mult
   addq  $20, %rdi        # %rdi = 20 + points*mult
   movq  %rdi, -8(%rbp)   # store value in local variable score
   cmpq  $100, %rdi       # compare score to $100 (compute score-100)
   jle   false            # jump if not greater
   movq  $1, %rax         # set function result to 1
   jmp   exit
false:
   movq  $0, %rax         # set function result to 0
exit:
   movq  %rbp, %rsp       # release stack frame
   popq  %rbp             # restore caller %rbp
   ret                    # return to caller
```

There are, of course, many other ways to write the function, including having separate code for the two return statements instead of jumping to a single set of code after storing the correct return value in `%rax`.

Question 4. (24 points) Compiler hacking. Recent programming languages have been adding features to decrease the chances of various runtime errors occurring during execution, such as memory management problems or bugs due to null pointers. One of our customers would like to add a null-safe method call to MiniJava as found in the Kotlin programming language. This is basically the same as an ordinary method call, except that if a null object pointer is used for the call, then the call evaluates to `null`, rather than causing a `NullPointerException` or some other error. To experiment with this in MiniJava, we want to add a 1-argument null-safe method call to the language. Since MiniJava does not include `null` as a keyword, we will use 0 as the "null value" in our prototype implementation.

The new null-safe method call works like this. The meaning of the expression `e1?.m(e2)` is to evaluate `e1` first. If `e1` evaluates to null (0), then the value of the entire `e1?.m(e2)` expression is 0. Otherwise, the value of `e1?.m(e2)` is the ordinary value of the regular method call `e1.m(e2)`. For the purposes of type-checking, the type of the null-safe method call `e1?.m(e2)` is assumed to be the same as the type of the ordinary method call `e1.m(e2)`.

Answer the questions below about how this new null-safe method call would be added to a MiniJava compiler. 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 null-safe method call to the original MiniJava language? Just describe any necessary changes and new token(s) needed and their name(s). You don't need to give JFlex or CUP specifications or code in this part of the question, but you will need to use any token name(s) you write here in a later part of this question.

Need a new token SAFECALL for the `?.` operator.

(Note that this should be a separate 2-character token since it is a single operator, although we did not deduct for answers that just specified adding a token for `?`, since the description was not entirely clear about whether `?.` should be treated as a single token or not.)

(continued on next page)

Question 4. (cont.) (b) (6 points) Complete the following new AST class to define an AST node type for this new null-safe method call. You only need to define instance variables and the constructor. Assume that all appropriate package and import declarations are supplied, and don't worry about visitor code.

(Hint: recall that the AST package in MiniJava contains the following key classes: `ASTNode`, `Exp extends ASTNode`, and `Statement extends ASTNode`. Also remember that each AST node constructor has a `Location` parameter, and the supplied `super(pos);` statement at the beginning of the constructor below is used to properly initialize the superclass with this information.)

```
public class SafeCall extends Exp {
  // add any needed instance variables below
```

```
  public Exp e1;
  public Identifier i;
  public Exp e2;
```

```
  // constructor - add parameters and method body below

  public SafeCall ( Exp e1, Identifier i, Exp e2, Location pos ){
```

```
    super(pos);   // initialize location information in superclass
    this.e1 = e1;
    this.i = i;
    this.e2 = e2;
```

```
  }
}
```

(continued on next page)

Question 4. (cont.) (c) (5 points) Complete the CUP specification below to define a production for this new null-safe method call, including associated semantic action(s) needed to parse the new expression and create an appropriate AST node (as defined in part (b) above). You should use any new lexical tokens defined in your answer to part (a) as needed. Use reasonable names for any other lexical tokens that already would exist in the compiler scanner and parser if you need them. We have added additional code to the parser rule for `Expression` below so the CUP specification for new null-safe method call can be written as an independent grammar rule with separate semantic actions.

Hint: recall that the `Location` of an item `foo` in a CUP grammar production can be referenced as `fooxleft`.

```
Expression ::= ...
     | SafeCall:e  {: RESULT = e; :}
    ...
     ;
```

```
SafeCall ::=
```

```
Expression:e1 SAFECALL Identifier:i LPAREN

     Expression:e2  RPAREN

{: RESULT = new SafeCall(e1, i, e2, e1xleft); :}
```

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a program containing this new null-safe method call expression is legal. You do not need to give code for a visitor method or anything like that – just describe what language rules (if any) need to be checked for this new statement to verify it is used correctly.

In the safecall e1?.m(e2), verify:

- **e1 is a reference type**
- **The type of e1 has a single-argument method m**
- **The type of e2 is assignment-compatible with the argument type of m**

The type of the entire expression is the declared result type of method m.

(continued on next page)

Question 4. (cont.) (e) (7 points) Describe the x86-64 code shape for this new null-safe method call expression 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. If needed, you should assume that the code generated for an expression will leave the value of that expression in `%rax`, as in our MiniJava project.

Hint: remember that an ordinary method call `e1.m(e2)` has the following codeshape:

Load `this` pointer into %rdi and value of argument e2 into %rsi
movq 0(%rdi),%rax # load vtable pointer
call *$\text{offset}_{\text{method}}$(%rax) # call via vtable pointer

You can use $\text{offset}_{\text{method}}$ to reference the correct vtable pointer offset for method `m` and do not need to know its exact numeric value.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them. Reminder: The argument registers for a function call are `%rdi,%rsi,%rdx,%rcx,%r8,` and `%r9` in that order.

Hint: be sure that your code follows the described operation and semantics of the new null-safe method call precisely, including calling the method only when `e1` is not null (i.e., 0).

```
    <generate code to evaluate e1 and leave the result in %rax>
    cmpq  %rax,$0
    je    done                # if e1 evaluates to 0, that is our result
    pushq %rax                # save e1 on stack
    <generate code to evaluate e2 and leave the result in %rax>
    movq  %rax, %rsi          # e2 is second argument
    popq  %rdi                # e1 is the "this" pointer to the object
    movq  0(%rdi), %rax       # call method m via vtable
    call  *offset_m(%rax)     #   (result will be in %rax)
done:                         # end of safe method call
```

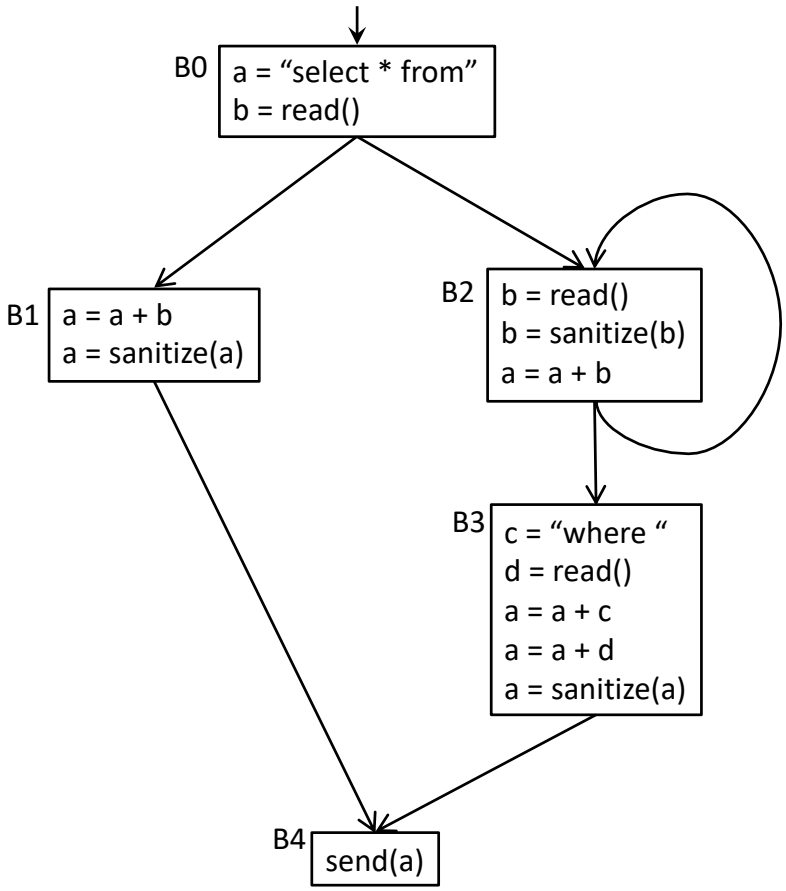
Several of the program analysis techniques pioneered in programming language compilers have been found useful for analyzing similar problems in other areas. For the next question we will use dataflow to analyze a security property known as tainted variables. The idea is to keep track of user input and verify that it is safe before using it to reference items in a database. When we read input, the data is assumed to be "tainted", i.e., it might not be safe to use. Once we verify or sanitize the data, it is no longer tainted. The goal is to ensure that no tainted data, either read directly or computed from tainted data, is used as input to database operations where it could lead to security problems.

For the next two problems, assume that we have a language with String values and variables. The language has the following String operations:

- x = "hello" – assignment with a String literal. String literals are not tainted, so x is not tainted after this assignment.
- x = y + z – string concatenation. x is assumed to be tainted regardless of the tainted status of y and z. The tainted status of y and z are not changed unless one of y or z (or both) are the same variable as x.
- x = read() – read user input as a string. x is now tainted.
- x = sanitize(y) – set x to the value of y after that value is sanitized. x is not tainted after this assignment. The tainted status of y does not change unless x and y are the same variable.
- send(x) – send the value of x to our database

The goal of our analysis will be to verify that no tainted values are sent to the database.

The following two problems refer to this dataflow graph, which uses the above operations:



Question 5. (18 points) Dataflow analysis – tainted variables. To defend against possible errors and malicious inputs, we would like to ensure that no "tainted" variables are sent to a database. We can use a dataflow framework to analyze which variable are tainted by defining the following sets for each basic block *b*:

- IN(b) – the set of variables that are known to be tainted on entry to block b
- OUT(b) – the set of variables that are known to be tainted on exit from block b
- GEN(b) – the set of all variables that are set to tainted values in block b and not later sanitized (i.e., assigned untainted values) in block b before exit from that block
- KILL(b) – the set of all variables that have been sanitized in block b (i.e., have been assigned untainted values) and not later tainted in block b before exit from that block.

The following dataflow equations describe the relationships between these sets:

$$\text{IN(b)} = \bigcup_{x \in pred(b)} \text{OUT(x)}$$

$$\text{OUT(b)} = \text{GEN(b)} \cup (\text{IN(b)} - \text{KILL(b)})$$

(a) (14 points) Complete the following table using iterative dataflow analysis to identify the tainted variables in the IN and OUT sets for each block in the above flow graph. You should first fill in the GEN and KILL sets for each block (which do not depend on other blocks) and the iteratively solve for IN and OUT. You can choose which ever direction (forward or backward) you wish to solve the equations.

	GEN	KILL	IN	OUT	IN	OUT	IN	OUT
B0	**b**	**a**		**b**		**b**		
B1		**a**	**b**	**b**	**b**	**b**	**No changes**	
B2	**a**	**b**	**b**	**a**	**a, b**	**a**		
B3	**d**	**a, c**	**a**	**d**	**a**	**d**		
B4			**b, d**	**b, d**	**b, d**	**b, d**		

(b) (2 points) Are any tainted variables sent to the database? Justify your answer using the control flow diagram and the information about the IN and OUT sets calculated above.

No. The only variable sent to the database is `a` in block 4, and `a` is not in the IN set of block 4, so therefore is not tainted when it is sent.

(c) (2 points) Do any tainted variables exist at the end of the program? List the variables, if any, and justify your answer using information about the IN and OUT sets calculated above.

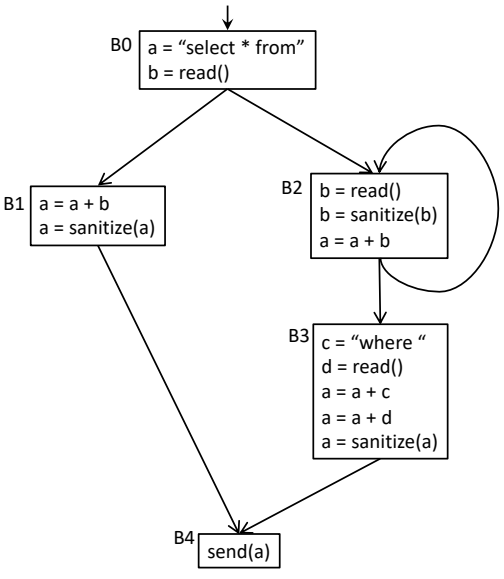
Yes. The OUT set for B4 includes `b` and `d`, so they are tainted variables that exist on exit from the program.

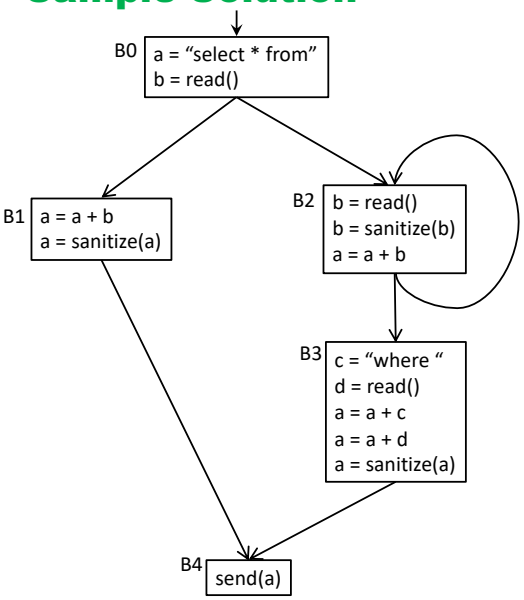
Question 6. (18 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

- 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 $x \neq y$.
- The **dominator set** of a node *x* is the set of nodes *dominated by x*.
  - | Dom(x) | ≥ 1
  - (note: sometimes the definition of Dom(x) is given as the set of all nodes that dominate *x*. For SSA it is more convenient to keep track of the set of nodes that *x* dominates.)
- 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*
- 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*

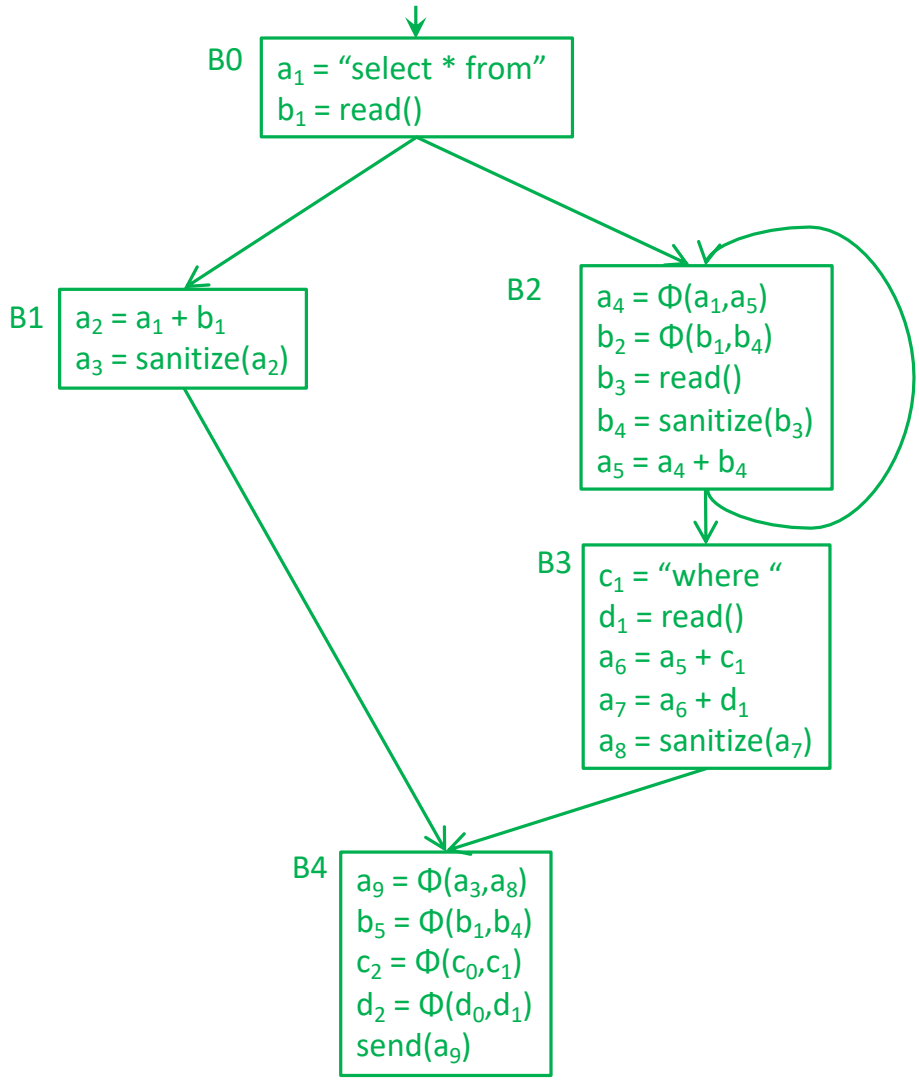
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it dominates, successor nodes to the dominated nodes, and the nodes that are in its dominance frontier (if any):

Node	Nodes dominated by this node	Successor(s) to nodes dominated by this node	Dominance Frontier of this node
B0	**B0, B1, B2, B3, B4**	**B1, B2, B3, B4**	
B1	**B1**	**B4**	**B4**
B2	**B2, B3**	**B2, B3, B4**	**B2, B4**
B3	**B3**	**B4**	**B4**
B4	**B4**		





Question 6. (cont.) (b) (10 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 next two questions concern register allocation and instructions scheduling. For both of these questions, assume that we're using the same hypothetical machine that was presented in lecture and in the textbook examples for list scheduling.

The instructions on this example machine are assumed to take the following numbers of cycles each:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1

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. Here is a sequence of instructions produced by the code generator for the statement `x = A[i] + i*3`. We assume that the elements of `A` each occupy 8 bytes. We also assume that one of the operands of a MULT or ADD instruction can be an integer constant.

```
a.  LOAD   t1 <- A          // t1 = A (array address)
b.  LOAD   t2 <- i          // t2 = i
c.  SHIFT  t3 <- t2, 3      // t3 = i*8
d.  ADD    t4 <- t1, t3     // t4 = address of (pointer to) A[i]
e.  LOAD   t5 <- MEM[t4]    // t5 = load A[i]
f.  MULT   t6 <- t2, 3      // t6 = i*3
g.  ADD    t7 <- t5, t6     // t7 = A[i] + i*3
h.  STORE  x <- t7          // x = A[i] + i*3
```

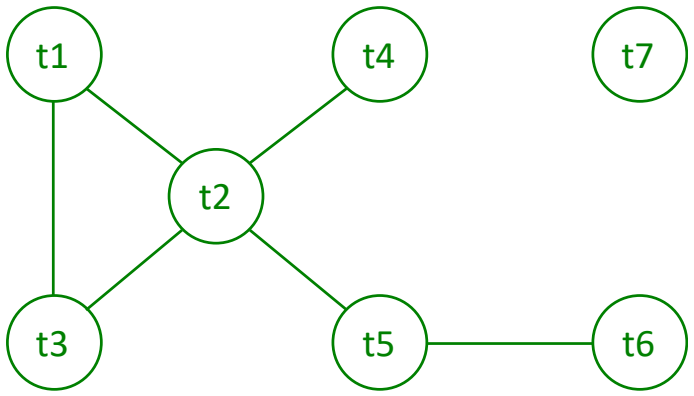
In a real compiler we would first use list scheduling to pick a (possibly) better order for the instructions, then use graph coloring to assign temporaries (t1-t7) to actual registers. But for this exam we're going to ask those two questions separately so the answers don't depend on each other, which will make it much easier to assign credit fairly (☺).

Answer the questions about this sequence of code on the next pages.

Question 7. (14 points) Register allocation/graph coloring.

(a) (8 points) Draw the interference graph for the temporary variables (t1-t7) in the code on the previous page (repeated here for convenience). You should assume that the code is executed in the sequence given and not rearranged before assigning registers.

```
a.  LOAD   t1 <- A
b.  LOAD   t2 <- i
c.  SHIFT  t3 <- t2, 3
d.  ADD    t4 <- t1, t3
e.  LOAD   t5 <- MEM[t4]
f.  MULT   t6 <- t2, 3
g.  ADD    t7 <- t5, t6
h.  STORE  x <- t7
```



(b) (6 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.

As usual, there are many possible answers. Here is one:

R1: t2, t6, t7

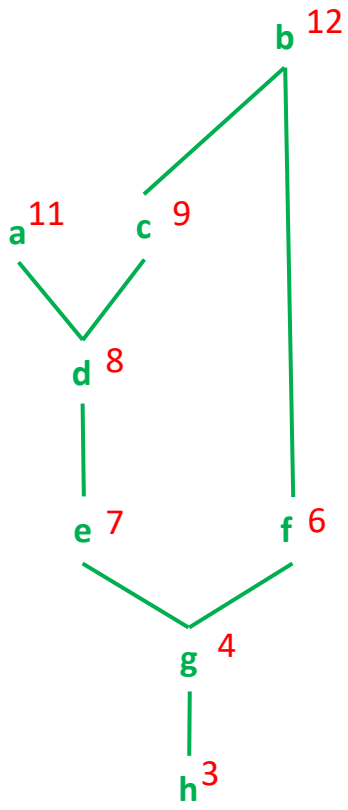
R2: t1, t4, t5

R3: t3

Question 8. (16 points) Forward list scheduling. (a) (8 points) Given the original sequence of instructions shown below, draw the precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with the letter identifying the instruction (a-h) and its latency – the number of cycles between the beginning of that instruction and the end of the graph on the correct path that respects the dependencies.

```
a.  LOAD   t1 <- A
b.  LOAD   t2 <- i
c.  SHIFT  t3 <- t2, 3
d.  ADD    t4 <- t1, t3
e.  LOAD   t5 <- MEM[t4]
f.  MULT   t6 <- t2, 3
g.  ADD    t7 <- t5, t6
h.  STORE  x <- t7
```

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1



(continued on next page)

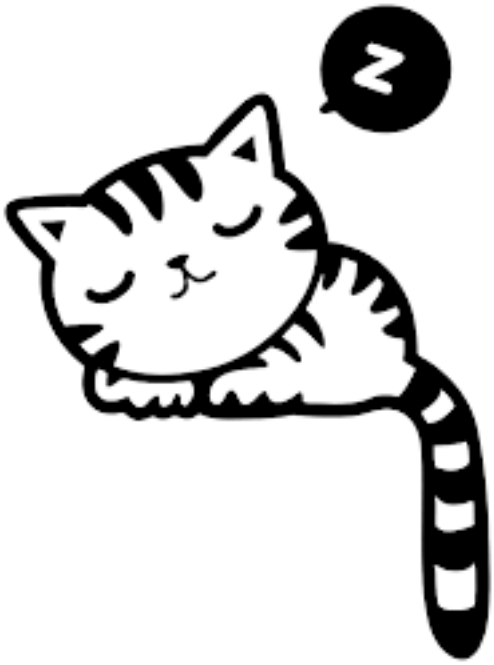
Question 8. (cont) (b) (8 points) Rewrite the instructions in the order they would be chosen by forward list scheduling. If there is a tie at any step when picking the best instruction to schedule next, pick one of them arbitrarily. Label each instruction with its letter and instruction operation (LOAD, ADD, etc.) from the original sequence above and the cycle number on which it begins execution (you do not need to write all of the instruction operands, just the letter and opcode). The first instruction begins on cycle 1. You do not need to show your bookkeeping or trace the algorithm as done in class, although if you leave these clues about what you did, it could be helpful if we need to figure out how to assign partial credit.

Cycle	Instr	Operation
1	b	LOAD
2	a	LOAD
3	---	
4	c	SHIFT
5	d	ADD
6	e	LOAD
7	f	MULT
8	---	
9	g	ADD
10	h	STORE

Question 9. (2 free points – all answers get the free points)

Draw a picture of something you are planning to during winter break!
– or –
Draw a picture of something you think one or more of your TAs will do during winter break!

A very common answer….



Have a great winter holiday and best wishes for the future!
The CSE 401/M501 staff