# CSE 401/M501 23au Final Exam 12/12/23 Sample Solution

The following question involves regular expressions. For this question you must restrict yourself to the basic regular expression operations covered in class and on homework assignments: *rs* , *r*|*s* , *r** , *r*+ , *r*?, character classes like [a-cxy] and [^aeiou], abbreviations *name=regexp* like as d=[0-9], and parenthesized regular expressions. No additional operations that might be found in the "regexp" packages in various Unix programs, scanner generators like JFlex, or programming language libraries are allowed.

**Question 1.** (12 points, 6 points each) The question that was dropped from the midterm! For each of the following, give a regular expression that generates the set of strings described. The set of strings generated by each regular expression should include the empty string as a possible result.

**There are many possible solutions. Here are a few possibilities for each part. Also, note that expressions like (b|c)* can also be written as [bc]*.**

(a) Strings over the alphabet {a,b,c} where the first a appears somewhere before the first b if the string contains at least one a and one b.

**(a|c)* | (b|c)* | c*a(a|c)*b(a|b|c)***

**(b|c)* | c*a(a|b|c)***

(b) Strings over the alphabet {a,b,c} that contain an even number of a's if they contain any a's.

**(b|c)* | ( (b|c)*a(b|c)*a(b|c)* )***

**( a(b|c)*a | b | c )***

**( (b|c)*a(b|c)*a )* (b|c)***

**Question 2.** (15 points) A bit of x86-64 coding. Here is a small C function that replaces a value in an integer array and returns the old value that was replaced.

```
// Store val in arr[idx] and return the value previously stored
// at that location.
int update(int arr[], int idx, int val) {
  int oldVal;
  oldVal = arr[idx];
  arr[idx] = val;
  return oldVal;
}
```

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 and using the local variable `oldVal`. 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). 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 2. (cont.)** Write your x86-64 translation of function `update` 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 `oldVal`. 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 update(int arr[], int idx, int val) {
  int oldVal;
  oldVal = arr[idx];
  arr[idx] = val;
  return oldVal;
}
```

**Argument registers:** `%rdi = arr, %rsi = idx, %rdx = val`

**There are obviously many possible solutions. Here are two. Clever x86-64 addressing mode version:**

```
update: pushq   %rbp                    # prologue – save rbp
        movq    %rsp,%rbp
        subq    $16,%rsp                # allocate stack
        movq    0(%rdi,%rsi,8),%rax     # load arr[idx]
        movq    %rax,-8(%rbp)           # store arr[idx] in oldVal*
        movq    %rdx,0(%rdi,%rsi,8)     # store val in arr[idx]
        movq    -8(%rbp),%rax           # return oldVal
        movq    %rbp,%rsp               # free stack & restore regs
        popq    %rbp                    #   (could also use leave)
        ret                             # return
```

**Simpler pointer addressing version:**

```
update: pushq   %rbp                # prologue – save rbp
        movq    %rsp,%rbp
        subq    $16,%rsp            # allocate stack
        shlq    %rsi,3              # rsi = idx*8
        addq    %rsi,%rdi           # rdi = address of arr[idx]
        movq    0(%rdi),%rax        # load arr[idx]
        movq    %rax,-8(%rbp)       # store arr[idx] in oldVal*
        movq    %rdx,0(%rdi)        # store val in arr[idx]
        movq    -8(%rbp),%rax       # return oldVal
        movq    %rbp,%rsp           # free stack & restore regs
        popq    %rbp                #   (could also use leave)
        ret                         # return
```

***Note: the load arr[idx] followed by the store in oldVal requires two instructions. x86-64 does not support two memory addresses in a single mov instruction, so it cannot be done with a single movq. The problem specification did require storing the original arr[idx] value in the local variable oldVal.**

**Question 3.** (24 points) Compiler hacking. As usually is the case once we've finished the MiniJava compiler project, one of our customers comes along with a request for a new feature or language addition. This time our customer has a lot of code that used the Java || conditional or operator, and we would like to add that to MiniJava for this problem. A conditional or expression is written as

*e1* || *e2*

The meaning of this expression is first evaluate *e1*. If *e1* evaluates to true, the entire *e1* || *e2* expression is true, and *e2* is *not* evaluated. If *e1* evaluates to false, then *e2* is evaluated, and the value of *e2* is the value of the *e1* || *e2* expression.

Answer the questions below about how this new operator 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 || operator to the original MiniJava language? Just describe any necessary changes and new token name(s) needed. 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.

**One new token needed: OR for conditional or || operator**

**(Note: this must be a single token, not a sequence of two | operators.)**

(continued on next page)

**Question 3. (cont.)** (b) (6 points) Complete the following new AST class to define an AST node type for this new `||` conditional or operator. 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 Or extends Exp {
  // add any needed instance variables below
```

```
Exp e1, e2;
```

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

  public Or ( Exp e1, Exp e2, Location pos ){
```

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

```
  }
}
```

(continued on next page)

**Question 3. (cont.)** (c) (5 points) Complete the CUP specification below to define a production for this new `||` conditional or operator, 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 the `||` operator 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 ::= ...
     | OrExp:e  {: RESULT = e; :}
    ...
     ;
OrExp ::=
```

```
Exp:e1 OR:e Exp:e2

   {: RESULT = new Or(e1, e2, exleft); :}


// Note: it would also be ok to use the Location of any
// component in the grammar rule for the Location of the
// expression
```

(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 operator 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 operator to verify it is used correctly.

**Verify that e1 and e2 have type Boolean**

**Designate the result type of the new `Or` expression node as Boolean (needed to check expressions that use this new operator)**

(continued on next page)

**Question 3. (cont.)** (e) (7 points) Describe the x86-64 code shape for this new || conditional or operator 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.

There are multiple ways to implement logical operators like this || conditional or, depending on whether it is being used to control execution in a loop or conditional statement or to produce a storable Boolean value. For this question your code shape should describe how to produce the binary value 1 (true) if *e1* || *e2* is true and produce the binary value 0 (false) if it is not. You may assume that the code shape for expressions *e1* and *e2* will similarly produce values 1 (true) or 0 (false) and do not have to do anything additional to make that happen.

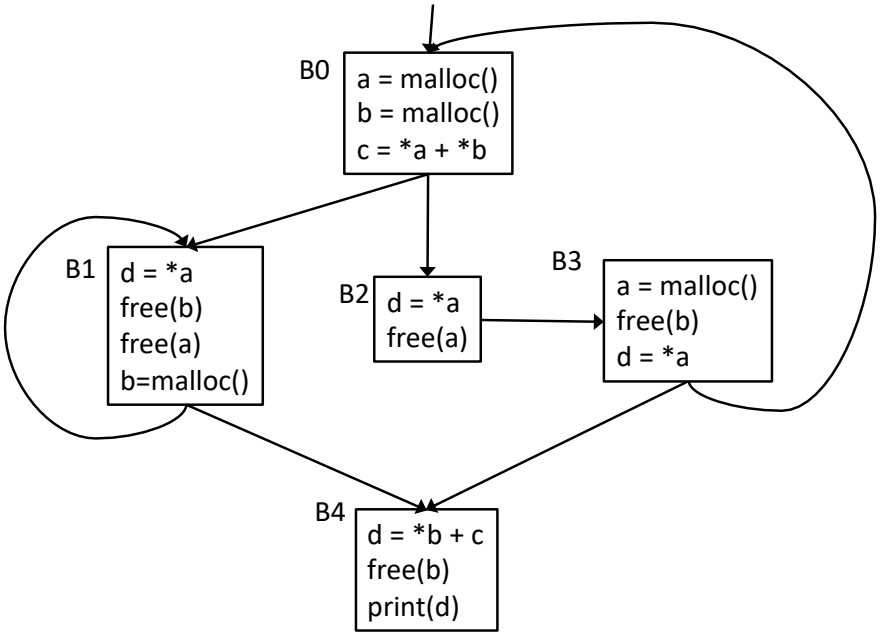
Use Linux/gcc x86-64 instructions and assembler syntax when needed. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them.

Hint: be sure that your code follows the described operation and semantics of the new || operator precisely, including evaluating *e2* only when needed.

**This is a very simple version and there are clearly other ways to do it. Note that the generated code for *e2* will only be executed if the generated code for *e1* produces 0 (false) during execution, as required.**

```
    Visit e1 to generate code.  Leaves 0/1 in %rax
    cmpq   $1,%rax          # see if result is 1 (true)
    jeq    done             # if result = 1, then finished
    Visit e2 to generate code.  Leaves result 0/1 in %rax, which is now the result of e1||e2
done:
```

The next two questions concern the following control flow graph for a strange program that manipulates C-style pointers and values:




The statements in this program include those that allocate and free memory. We would like to ensure that all allocated memory is free'd before the end of execution. We want to use a dataflow analysis to discover if there may be a memory leak by keeping track of which variables currently point to allocated heap space at various points in the program.

For this problem, assume that our language two types: pointers and scalars. The language has the following operations:

- x = malloc() – allocates a block of memory and assigns its location to x (x must be pointer type)
- free(x) – free allocated memory referenced by x (x must be pointer type)
- *x – dereference x to access the scalar value at the address whose value is contained in x (x must be pointer type)
- x = y – ordinary assignment (x and y must be scalars)
- x = y + z – compute sum y + z and assign to x (x, y, and z must be scalars)
- print(x) – print value of x (x must be a scalar)

This program clearly has other bugs, including reading uninitialized heap data and possible double-free errors. You should ignore these other potential memory errors and use the dataflow analysis to locate potential memory leaks only (heap memory allocated but never freed).

(continued on next page)

**Question 4.** (18 points) Dataflow analysis – memory leaks. We would like to use dataflow analysis to discover which variables point to allocated heap memory at different points in the program. To do that, we define the following sets for each basic block *b*:

IN(*b*) = the set of variables that are known to point to allocated heap memory (malloc) on entry to block *b*

OUT(*b*) = the set of variables that are known to point to allocated heap memory (malloc) on exit from block *b*

GEN(*b*) = the set of variables that are assigned a pointer to a newly allocated heap memory location (i.e., malloc) inside block *b* that is not later freed by the end of block *b*

KILL(*b*) = the set of all variables that are freed in block *b* and not later set to point to a newly allocated block of storage in block *b* by a later use of malloc

The GEN(*b*) and KILL(*b*) sets for each block can be computed once based on the static contents of the statements in block *b*.

The dataflow equation relating these sets are

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

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

(a) (15 points) Complete the following table using iterative dataflow analysis to identify the variables in the IN and OUT sets for each block in the above flowgraph that are known to point to allocated-but-not yet-freed heap data. You should first fill in the GEN and KILL sets for each block, then iteratively solve for IN and OUT. You can choose whichever direction you wish (forward or backward) to solve the equations. The table should only contain pointer variables (e.g., a and b).

| | GEN | KILL | IN | OUT | IN | OUT | IN | OUT |
|---|---|---|---|---|---|---|---|---|
| B0 | **a, b** | -- | -- | **a, b** | **a** | **a, b** | **a** | **a, b** |
| B1 | **b** | **a** | **a, b** | **b** | **a, b** | **b** | **a,** | **b** |
| B2 | -- | **a** | **a, b** | **b** | **a, b** | **b** | **a, b** | **b** |
| B3 | **a** | **b** | **b** | **a** | **b** | **a** | **b** | **a** |
| B4 | -- | **b** | **a, b** | **a** | **a, b** | **a** | **a, b** | **a** |

Same

(b) (3 points) Based on this analysis, is there heap-allocated memory that might not be freed by the end of the program? If so, which variable(s) reference this allocated memory and how do you know this data might not be properly freed?

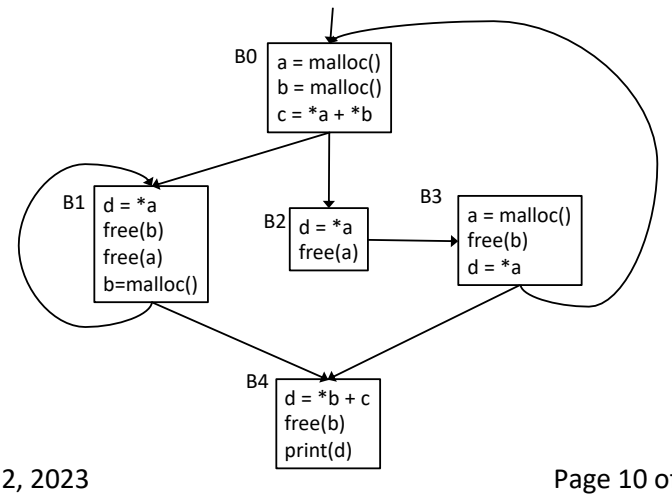
**Yes. Variable a is in the OUT set of block 4, which is the end of the flowgraph. That means a has been allocated but not freed previously when the program terminates.**

**Question 5.** (18 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

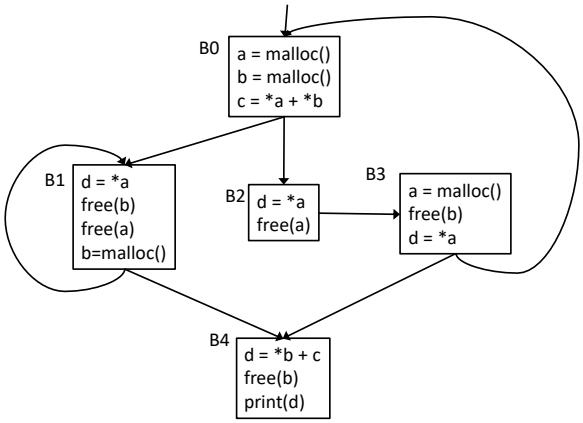
- Every control flow graph has a unique **start node** s.
- Node *x* **dominates** node *y* if every path from s to *y* must go through x.
  - A node *x* dominates itself.
- A node *x* **strictly dominates** node *y* if *x* dominates *y* and $x \neq y$.
- The **dominator set** of a node *x* is the set 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*

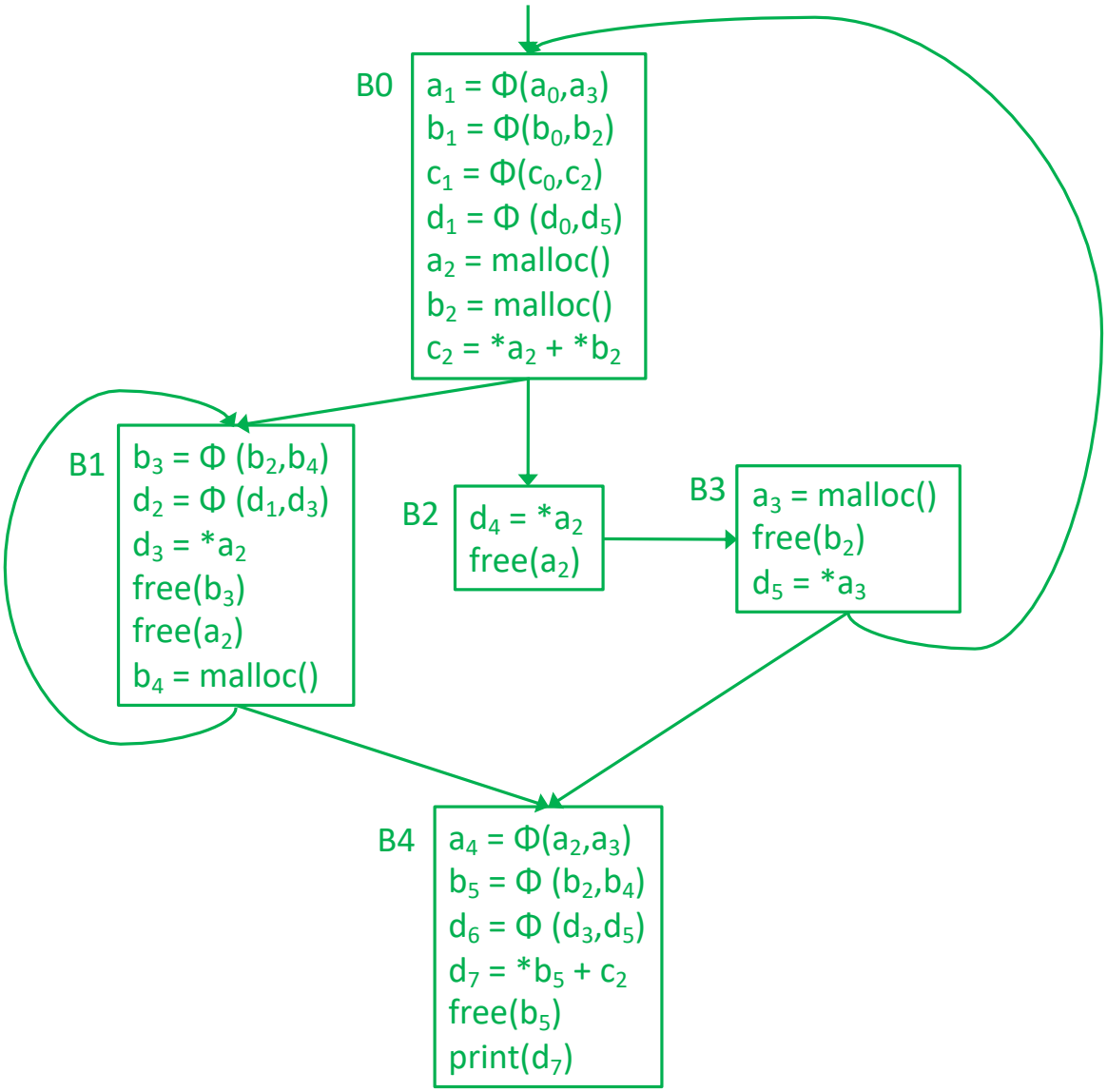
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it dominates, successor nodes to the dominated nodes, and the nodes that are in its dominance frontier (if any):

| 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** | **B0, B1, B2, B3, B4** | **B0** |
| B1 | **B1** | **B1, B4** | **B1, B4** |
| B2 | **B2, B3** | **B0, B3, B4** | **B0, B4** |
| B3 | **B3** | **B0, B4** | **B0, B4** |
| B4 | **B4** | **---** | **---** |



**Question 5 (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:

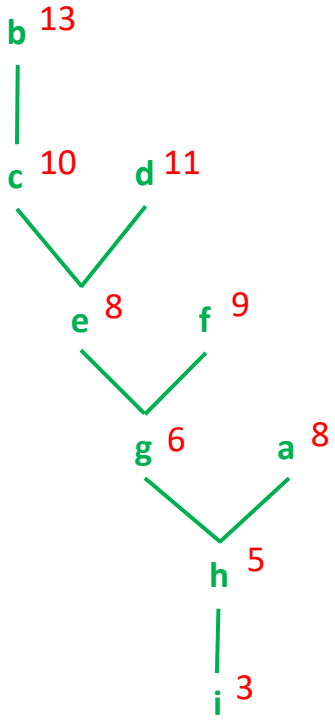
```
a.  LOAD   t1 <- x          // t1 = x
b.  LOAD   t2 <- y          // t2 = y
c.  MULT   t3 <- t2,5       // t3 = y * 5
d.  LOAD   t4 <- z          // t4 = z
e.  MULT   t5 <- t3,t4      // t5 = y * 5 * z
f.  LOAD   t6 <- w          // t6 = w
g.  ADD    t7 <- t5, t6     // t7 = (y * 5 * z) + w
h.  MULT   t8 <- t1,t7      // t8 = x * ((y * 5 * z) + w)
i.  STORE  n <- t8          // n = x * ((y * 5 * z) + w)
```

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-t8) 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 6.** (16 points) Forward list scheduling. (a) (8 points) Given the original sequence of instructions on the previous page, draw the precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with the letter identifying the instruction (a-i) and its latency – the number of cycles between the beginning of that instruction and the end of the graph on the shortest possible path that respects the dependencies.

| | | |
|---|---|---|
| a. | LOAD | t1 <- x |
| b. | LOAD | t2 <- y |
| c. | MULT | t3 <- t2,5 |
| d. | LOAD | t4 <- z |
| e. | MULT | t5 <- t3,t4 |
| f. | LOAD | t6 <- w |
| g. | ADD | t7 <- t5, t6 |
| h. | MULT | t8 <- t1,t7 |
| i. | STORE | n <- t8 |




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

(continued on next page)

**Question 6. (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.

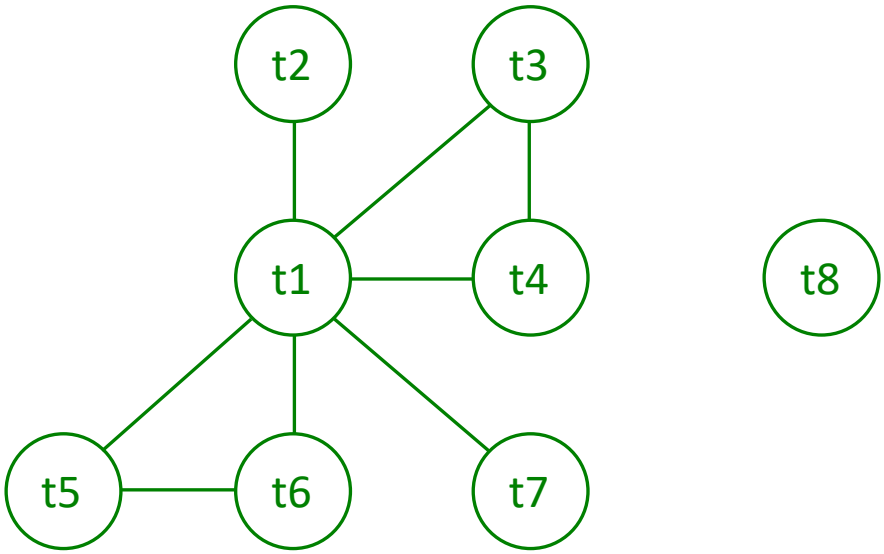
1. **b LOAD**
2. **d LOAD**
3. **f LOAD**
4. **c MULT**
5. **a LOAD**
6. **e MULT**
7. **---**
8. **g ADD**
9. **h MULT**
10. **---**
11. **i STORE**

**This is the unique correct solution. There are no ties to be broken arbitrarily at any step in the algorithm.**

**Question 7.** (15 points) Register allocation/graph coloring.

(a) (8 points) Draw the interference graph for the temporary variables (t1-t8) 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 <- x
b.  LOAD   t2 <- y
c.  MULT   t3 <- t2,5
d.  LOAD   t4 <- z
e.  MULT   t5 <- t3,t4
f.  LOAD   t6 <- w
g.  ADD    t7 <- t5, t6
h.  MULT   t8 <- t1,t7
i.  STORE  n <- t8
```




(b) (7 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.

**There are several possible assignments that require the minimum of 3 registers. Here is one.**

**R1: t1, t8**

**R2: t2, t3, t5, t7**

**R3: t4, t6**

**Question 8.** (2 free points – all answers get the free points)

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

**Here's one that was fairly popular:**

*Have a great break and best wishes for the holidays! We'll see you back here in the winter!*
The CSE 401/M501 staff