CSE 401/M501 23sp Final Exam 6/6/23 Sample Solution

Question 1. (14 points) A bit of x86-64 coding. Here is a small C function that has three integer parameters and returns the largest value of the three. We assume that there is a 2-argument `max` function that returns the largest of its arguments, and it is used by the `max3` function.

```
int max(int x, int y);  // defined elsewhere

int max3(int a, int b, int c) {
  return max(a, max(b, c));
}
```

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. Also note that we assume the function `max` exists and we can call it, but you do not have to define that function or provide its code.

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). 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 `max3` below. Remember to read and follow the above ground rules carefully, including managing registers properly and using the correct argument registers for function calls. Brief comments are appreciated. Original code repeated below for convenience:

```
int max(int x, int y);  // defined elsewhere

int max3(int a, int b, int c) {
  return max(a, max(b, c));
}
```

```
# input argument registers: %rdi a, %rsi b, %rdx c

max3:  pushq   %rbp                 # function prologue
       movq    %rsp,%rbp
       subq    $16,%rsp             # allocate space to save a
                                    # (size must be a multiple of 16)
       movq    %rdi,-8(%rbp)        # save a in stack frame
                                    # call max(b, c)
       movq    %rsi,%rdi            # first argument b
       movq    %rdx,%rsi            # second argument c
       call    max                  # max(b,c), result in %rax
                                    # call max(a, result)
       movq    -8(%rbp),%rdi        # first argument a
       movq    %rax,%rsi            # second argument result of first call
       call    max                  # compute final result in %rax
       movq    %rbp,%rsp            # function epilog (could use leave instead)
       popq    %rbp
       ret
```

Note: The main thing to be careful about here is to be sure the arguments are in the correct registers, and also to be sure to save a in memory during the first call to max, since all argument registers are potentially clobbered on a function call, and we cannot depend on any of them retaining their original values.

Question 2. (24 points) Compiler hacking. One of our big customers has gotten tired of writing `while` loops to iterate through integer values. They have asked us to add a "counting loop" that executes a statement repeatedly with an integer variable taking on successive integer values (i.e., not a general `for` loop as found in Java, C, C++, and many other languages). The new loop adds the following production to the MiniJava grammar:

Statement ::= "for" Identifier "from" Expression "to" Expression "do" Statement

(A copy of the original MiniJava grammar is included at the end of the exam. You should remove it for reference while working on this problem.)

The idea is that the Statement in the loop body is executed repeatedly with the Identifier assigned successive integer values starting with the value of the first Expression and increasing by 1 each time the loop repeats until the final iteration where the Identifier has the value of the second Expression. For example, the following code stores the value 1 + 2 + … + 10 in variable sum:

```
sum = 0;
for i from 1 to 10 do sum = sum + i;
```

The Identifier in the for statement must have been declared previously and must have type `int`. The two Expressions are only evaluated once, before the Identifier is assigned its initial value and before the loop body executes. The Expressions are not re-evaluated again as the loop executes. So, for example, the following code has exactly the same effect as the previous example:

```
sum = 0; i = 0;
for i from i+1 to i+10 do sum = sum + i;
```

In other words, the loop bounds i+1 and i+10 are evaluated before the loop begins execution and before the initial assignment to i, and are not reevaluated again. The Expressions are evaluated in order from left to right. If the value of the first Expression is greater than the value of the second Expression, then the body of the loop is not executed. The value in the loop variable is not defined after the loop terminates – it might be equal to the value of one of the Expressions, or it could have any other value depending on what the implementation does.

Answer the questions below about how this new loop 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 kind of statement 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, but you will need to use any token name(s) you write here in a later part of this question.

We need tokens for the four new keywords: FOR, FROM, TO, and DO

(continued on next page)

Question 2. (cont.) (b) (5 points) Complete the following new AST class to define an AST node type for the new loop. 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 `For` constructor below is used to properly initialize the superclass with this information.)

```
public class For extends Statement {
  // add any needed instance variables below
```

```
  public Identifier id;

  public Exp e1, e2;

  public Statement s;
```

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

  public For (Identifier id,Exp e1,Exp e2,Statement s,Location pos){
```

```
    super(pos);   // initialize location information in superclass

    this.id = id;

    this.e1 = e1;

    this.e2 = e2;

    this.s = s;
```

```
f
  }
}
```

Note: Constructor parameters could be in a different order as long as that order matches the AST node creation code in the CUP semantic action for the new comparison operator, below.

(continued on next page)

Question 2. (cont.) (c) (5 points) Complete the CUP specification below to define a production for the new loop statement including associated semantic action(s) needed to parse the new statement 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 needed that already would exist in the compiler scanner and parser if you need them. We have added additional code to the parser rule for `Statement` below so the CUP specification for the loop 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`.

```
Statement ::= ...
     | ForStatement:s  {: RESULT = s; :}
    ...
     ;
ForStatement ::=
```

```
FOR Identifier:i FROM Exp:e1 TO EXP:e2 DO

   Statement: s

{: RESULT = new For(i,e1,e2,s,ixleft); :}
```

(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 loop 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 loop.

- **Verify that the `for` variable has been declared and has type `int`**
- **Verify that expressions `e1` and `e2` have type `int`**

(continued on next page)

Question 2. (cont.) (e) (8 points) Describe the x86-64 code shape for this new `for` 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. If needed, you should assume that the code generated for an expression will leave the value of that expression in `%rax`, as we did in the MiniJava project.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. However, remember that the question is asking for the code shape for this new 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.

Hint: be sure that your code follows the described operation and semantics of the new `for` statement precisely.

This answer uses a strategy similar to the code shape outlined for our MiniJava compilers. Values are pushed on the stack to save them if they will be needed later. The code arranges for the value of the second expression (the limit) to be stored on the top of the stack when the statement that makes up the loop body is executed. We are also careful to evaluate both loop expressions before assigning to the loop variable. We also assume the loop variable is a local variable in the stack frame. The code could be generalized to allow the loop variable to be elsewhere but this shows the key ideas. Other reasonable pseudo-code was fine if done correctly.

```
            <code to evaluate first expression e1 and leave value in %rax>
            pushq   %rax                    # save initial expression value
            <code to evaluate second expression e2 and leave value in %rax>
            popq    %rdx                    # reload initial expression value
            movq    %rdx,offset_var(%rbp)   # store initial expression in loop variable
            pushq   %rax                    # save second expression on top of stack
test:
            movq    offset_var(%rbp),%rax   # load variable into %rax
            popq    %rdx                    # load limit into %rdx
            cmpq    %rdx,%rax               # set cond codes with var - limit
            jg      done                    # exit if var-limit > 0, i.e., var > limit
            pushq   %rdx                    # push limit back on stack
            <code for loop body statement>
            movq    offset_var(%rbp),%rax   # increment variable by 1
            addq    $1,%rax
            movq    %rax,offset_var(%rbp)
            jmp     test                    # repeat loop
done:                                       # end of loop; second (limit) expression already
                                            # popped from stack just before jump to done:
```

Question 3. (10 points) vtables. Suppose we have the following four classes in a MiniJava program.

class Shape {
  double area() { return 0.0; }
  String describe() { return "Shape"; }
}
	class Polygon extends Shape {
	  int nSides() { return 4; }
	  void rotate(double degrees) { }
	}

class Triangle extends Polygon {
  String describe() { return "Triangle"; }
  double area() { return 17.0; }
  int nSides() { return 3; }
}
	class Circle extends Shape {
	  String describe() { return "Circle"; }
	  void expand(double factor) { }
	  double area() { return 3.14; }
	}

(Obviously, the values returned by the methods are of no significance – they were just included above to be sure the code was syntactically correct.)

When class `Shape` was compiled, the compiler picked the following vtable layout for the class:

	Vtable layout	offset
`Shape$$:`	`.quad 0`	# no superclass
	`.quad Shape$area`	# +8
	`.quad Shape$describe`	# +16

Below, show appropriate vtable layouts for the remaining classes, in the same format used above for class `Shape`. Be sure to properly account for inherited methods in the vtable layouts. There is at least one correct solution. If there are different possible solutions, any one is acceptable.

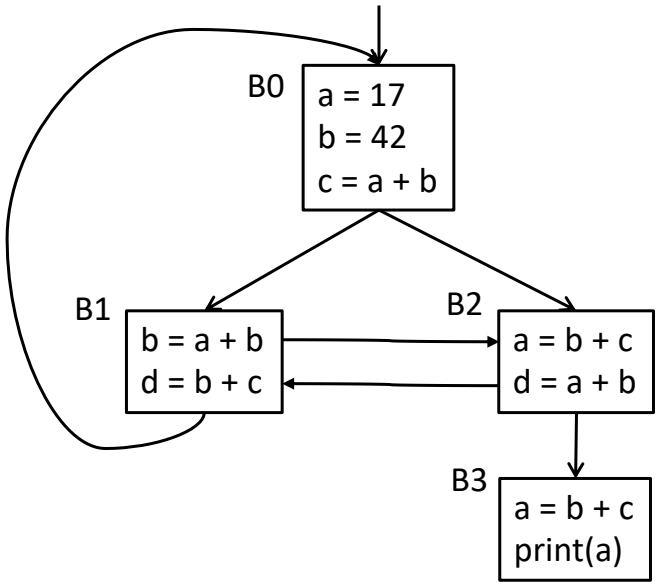
	Vtable layout	offset
`Polygon$$:`	`.quad Shape$$`	# superclass
	`.quad Shape$area`	# +8
	`.quad Shape$describe`	# +16
	`.quad Polygon$nsides`	# +24
	`.quad Polygon$rotate`	# +32

	Vtable layout	offset
`Triangle$$:`	`.quad Polygon$$`	# superclass
	`.quad Triangle$area`	# +8
	`.quad Triangle$describe`	# +16
	`.quad Triangle$nsides`	# +24
	`.quad Polygon$rotate`	# +32

	Vtable layout	offset
`Circle$$:`	`.quad Shape$$`	# superclass
	`.quad Circle$area`	# +8
	`.quad Circle$describe`	# +16
	`.quad Circle$expand`	# +24

Notes: the first two slots in all vtables must match `Shape` (`area`, `describe`). The first four slots in Triangle must match the method order in `Polygon`.

The next two questions concern the following control flow graph:



Question 4. (18 points) Dataflow analysis – available expressions. Recall from lecture that an expression *e* is *available* at a program point *p* if every path leading to point *p* contains a prior definition of expression *e* and *e* is not killed along a path from a prior definition by having one of its operands re-defined on that path.

We would like to compute the set of available expressions at the beginning of each basic block in the flowgraph shown above. For each basic block *b* we define the following sets:

AVAIL(*b*) = the set of expressions available on entry to block *b*

NKILL(*b*) = the set of expressions *not killed* in *b* (i.e., all expressions defined somewhere in the flowgraph except for those killed in *b*)

DEF(*b*) = the set of all expressions defined in *b* and not subsequently killed in *b*

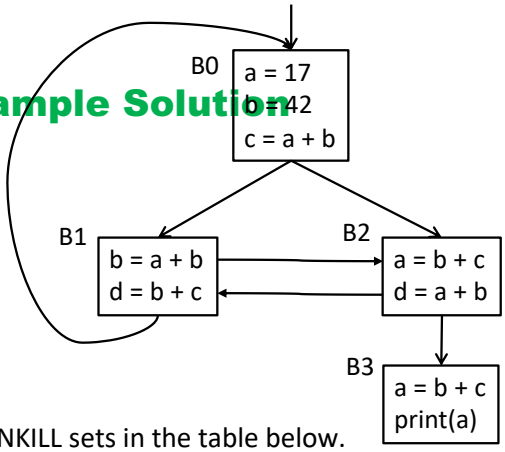
The dataflow equation relating these sets is

$$AVAIL(b) = \cap_{x \in preds(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$

i.e., the expressions available on entry to block *b* are the intersection of the sets of expressions available on exit from all of its predecessor blocks *x* in the flow graph.

On the next page, calculate the DEF and NKILL sets for each block, then use that information to calculate the AVAIL sets for each block. You will only need to calculate the DEF and NKILL sets once for each block. You may need to re-calculate some of the AVAIL sets more than once as information about predecessor blocks changes.

Hint: notice that there are only two expressions calculated in this flowgraph: a+b, and b+c. So all of the AVAIL, NKILL, and DEF sets for the different blocks will contain some, none, or both of these expressions.



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

AVAIL(*b*) = expressions available on entry to block *b*
NKILL(*b*) = expressions *not killed* in *b*
DEF(*b*) = expressions defined in *b* and not subsequently killed in *b*

AVAIL(*b*) = $\cap_{x \in preds(b)}$ (DEF(*x*) $\cup$ (AVAIL(*x*) $\cap$ NKILL(*x*)))

(a) (8 points) For each of the blocks B0, B1, B2, and B3, write their DEF and NKILL sets in the table below.

Block	DEF	NKILL
B0	{ a + b }	Ø
B1	{ b + c }	Ø
B2	{ a + b, b + c }	{ b + c }
B3	{ b + c }	{ b + c }

(b) (8 points) Now, in the table below, give the AVAIL sets showing the expressions available on entry to each block. If you need to update this information as you calculate the sets, be sure to cross out previous information so it is clear what your final answer is.

Block	AVAIL
B0	Ø
B1	{ a + b }
B2	Ø
B3	{ a + b, b + c }

(c) (2 points) One use of available expression analysis is to detect when an expression is available entering a basic block and does not need to be recalculated when it is used in that block. Are there any expressions that don't need to be recalculated because they are available on entry to some block? Identify each expression and list the block(s) where it does not need to be recalculated, or write none if there are no redundant calculations.

The expresson a+b is available on entry to B1 and does not need to be recalculated there.
The expression b+c is available on entry to B3 and does not need to be recalculated there.

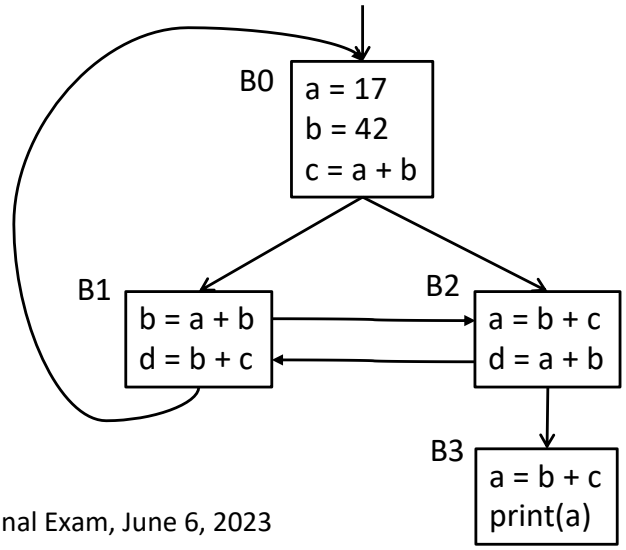
Question 5. (18 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

- 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 *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*

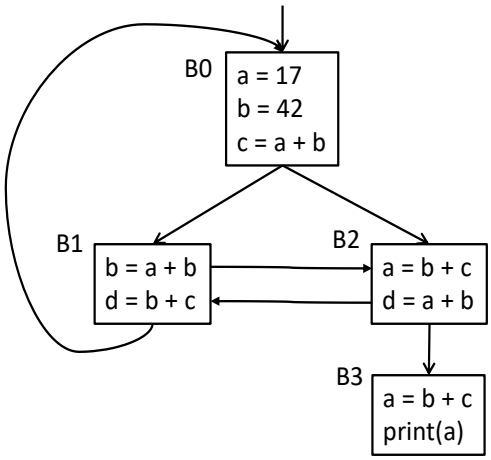
  A node might not have an immediate dominator. A node has at most one immediate dominator.
- The **dominator tree** of a control flow graph is a tree where there is an edge from every node *x* to its immediate dominator idom(*x*).
- The **dominance frontier** of a node *x* is the set of all nodes *w*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

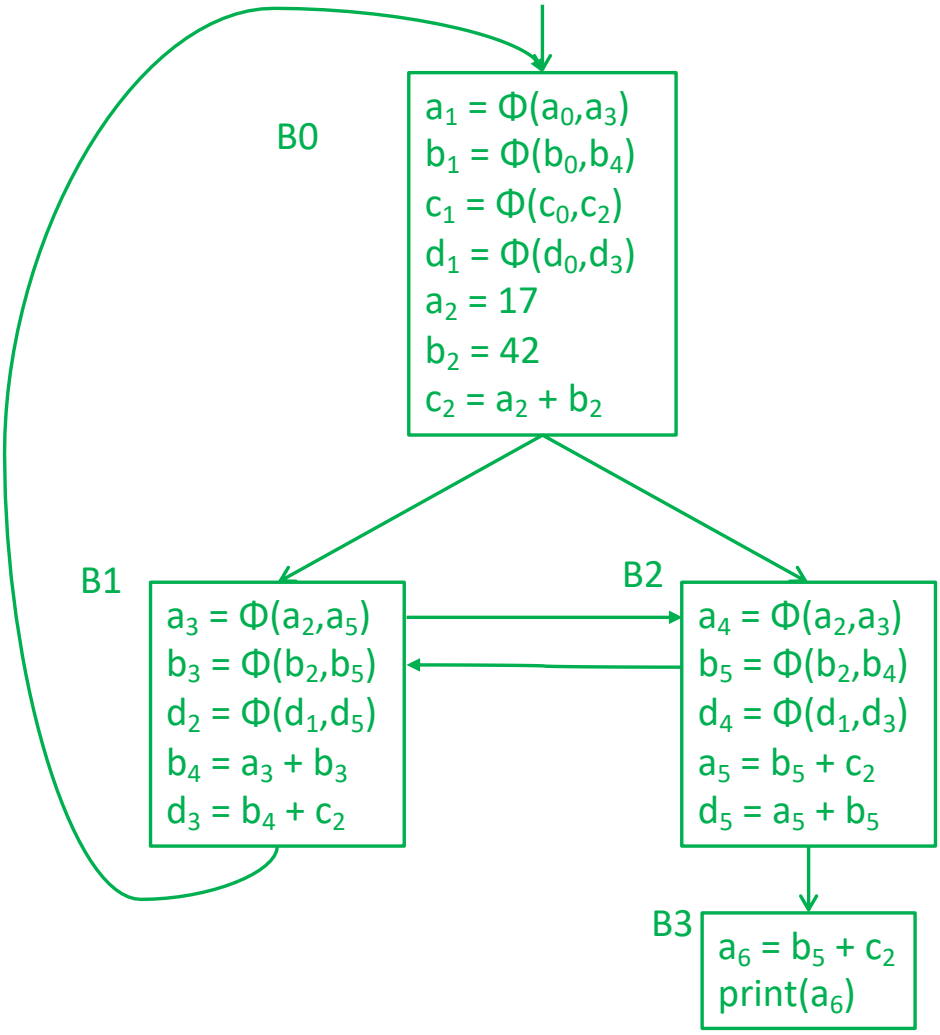
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it dominates, successor nodes to the dominated nodes, and the nodes that are in its dominance frontier (if any):

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



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. Given the statement y = m*x[i]+b; here's what the instruction selector generated:

```
a.  LOAD   t1 <- m          // t1 = m
b.  LOAD   t2 <- x          // t2 = address of x[] array
c.  LOAD   t3 <- i          // t3 = i
d.  SHIFT  t4 <- t3, 3      // t4 = i*8 (shift t3 left 3)
e.  ADD    t5 <- t2, t4     // t5 = address of x[i]
f.  LOAD   t6 <- MEM[t5]    // t6 = x[i] – i.e., load from computed address, not a named var
g.  MULT   t7 <- t1, t6     // t7 = m*x[i]
h.  LOAD   t8 <- b          // t8 = b
i.  ADD    t9 <- t7+t8      // t9 = m*x[i]+b
j.  STORE  y <- t9          // store y
```

(Note that this code assumes that array variable x contains a pointer to an array, as in Java.)

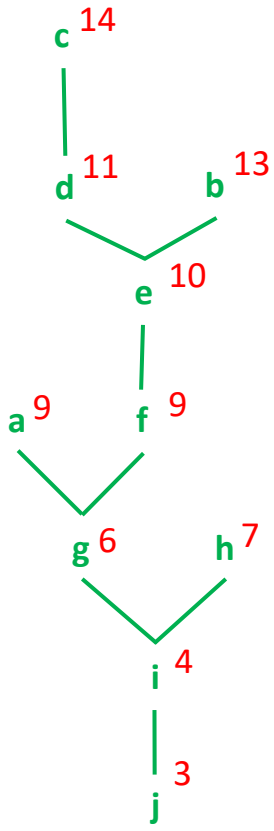
In a real compiler we would first use list scheduling to pick a (possibly) better order for the instructions, then use graph coloring to assign temporaries (t1-t9) to actual registers. But for this exam we're going to ask those two questions separately so the answers don't depend on each other, which will make it much easier to assign credit fairly (☺).

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

Question 6. (15 points) Forward list scheduling. (a) (8 points) Given the original sequence of instructions on the previous page for the assignment statement y = m*x[i]+b, draw the precedence graph showing the dependencies between these instructions. Label each node (instruction) in the graph with the letter identifying the instruction (a-j) 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 <- m
b.  LOAD   t2 <- x
c.  LOAD   t3 <- i
d.  SHIFT  t4 <- t3, 3
e.  ADD    t5 <- t2, t4
f.  LOAD   t6 <- MEM[t5]
g.  MULT   t7 <- t1, t6
h.  LOAD   t8 <- b
i.  ADD    t9 <- t7+t8
j.  STORE  y <- t9
```

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



(continued on next page)

Question 6. (cont) (b) (7 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.

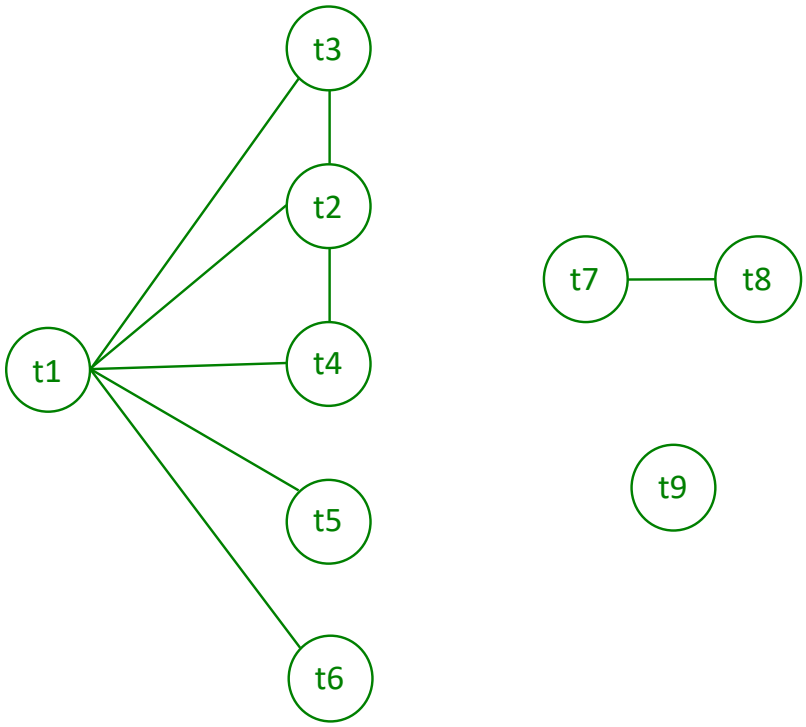
1. **c LOAD**
2. **b LOAD**
3. **a LOAD**
4. **d SHIFT**
5. **e ADD**
6. **f LOAD**
7. **h LOAD**
8. **---**
9. **g MULT**
10. **---**
11. **I ADD**
12. **j STORE**

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

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

(a) (8 points) Draw the interference graph for the temporary variables (t1-t9) 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 <- m
b.  LOAD    t2 <- x
c.  LOAD    t3 <- i
d.  SHIFT   t4 <- t3, 3
e.  ADD     t5 <- t2, t4
f.  LOAD    t6 <- MEM[t5]
g.  MULT    t7 <- t1, t6
h.  LOAD    t8 <- b
i.  ADD     t9 <- t7+t8
j.  STORE   y <- t9
```



(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.

R1: t1, t7, t9

R2: t2, t5, t6, t8

R3: t3, t4

There are, of course, some other solutions that assign temporaries to three registers. Any correct solution received full credit.

To wrap up, a short-answer question.

Question 8. (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 (if it matters). 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

scan Detect that a comment beginning with /* does not have a matching */ before reaching the end of the input file.

reg Add extra instructions to temporarily store a value in memory when there aren't enough registers available to keep all live values in registers at a given point in the program.

run Detect that in an array reference `a[k]`, the subscript `k` is out-of-bounds

sem Warn the programmer that an instance variable declaration in a subclass uses the same name as an instance variable name in a superclass, which can hide (shadow) the superclass variable

sem Report that the `*` operation cannot be used to combine expressions of type `String`.

can't Report that a particular pointer (reference) variable `p` will be `null` when used.
(Note that this is asking to determine that some use of `p` will always be `null`, which is undecidable, as opposed to detecting whether `p` is `null` at a particular use during execution)

instr Use the x86-64 addressing modes to calculate x*5 using `leaq (%rax,%rax,4),%rax`

parse Report that there is an extra "`else` *stmt*" clause that is not associated with any previous `if`.

sem In a method call `x.f(a,b,c)`, verify that the type of `x` actually does have a visible method `f` with the correct number and types of parameters.

opt Remove an instruction from the generated code that stores a value in a variable if that variable is never subsequently used.

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

Draw a picture of something you are planning to do this summer!
– or –
Draw a picture of one or more of your TAs!

TBD

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