CSE 401 / M501 18au Final Exam

December 13, 2018

Name _____

There are 8 questions worth a total of 140 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

The exam will be scanned for grading. **Write on the front side of each page only.** The back sides will not be scanned. **Two blank pages** are provided **at the end** of the exam if you need extra space for answers or scratch work. If you write any answers on these pages, please be sure to indicate on the original question page that your answers are continued on these extra pages, and label the answers on the additional pages if you use them.

There are several pages in the exam that are intended to be removed for reference during the exam. **Do not write on those pages.** They will not be scanned for grading.

This exam is closed book, closed notes, closed electronics, closed neighbors, open mind, ...

Please wait to turn the page until everyone has their exam and you have been told to begin.

If you have questions during the exam, raise your hand and someone will come to you.

Legibility is a plus, as is showing your work. We can't read your mind, but we'll try to make sense of what you write.

1	/ 10
2	/ 20
3	/ 24
4	/ 16
5	/ 20
6	/ 20
7	/ 15
8	/ 15
Total	/ 140

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 (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

_____ In an array reference a[k], discover that the subscript k is out of bounds

_____ Report a variable x has been declared twice in the same scope (duplicate declaration)

_____ Insert extra store and load instructions in the program when there are not enough registers available to hold all active values

_____ In an assignment statement var=exp, discover that the type of exp is not assignment compatible with the type of var

_____ Discover code that computes the same value each time a loop executes and move it so the value is computed just once outside the loop

_____ Discover that === is not a legal relational operator in Java

_____ Decide whether a method declaration overrides or overloads a method in some superclass

_____ In the assignment statement p=(t)q that contains a cast involving object references, verify that the object referenced by q actually has type t or some subtype of t

_____ Determine that some variable used in the program might not be initialized at the point where it is used

_____ Decide if a program is stuck in an infinite loop or will eventually terminate after it starts executing

Question 2. (20 points) x86-64 coding. Consider the following tiny MiniJava class:

```
class Test {
  int min(int i, int j) {
    int ans;
    if (i < j)
      ans = i;
    else
      ans = j;
    return ans;
  }

  int min3(int x, int y, int z) {
    return this.min(x, this.min(y,z));
  }

}  // end of class Test
```

On the next page, translate the methods in this class into x86-64 assembly language. You should use the standard runtime conventions for parameter passing (including the `this` pointer), register usage, and so forth that we used in the MiniJava project, including using `%rbp` as a stack frame pointer when a stack frame is allocated. You should assume that the vtable for class `Test` contains an appropriate pointer to `min` at offset +8 and a pointer to `min3` at offset +16.

`call` instruction hints: Recall that if `%rax` contains a pointer to (i.e., the memory address of) the first instruction in a method, then you can call the method by executing `call *%rax`. If `%rax` contains the address of a vtable, we can call a method whose pointer is at offset *d* in that vtable by executing `call *`*d*`(%rax)`.

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

- 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
- Pointers and `int`s are 64 bits (8 bytes) each, as in MiniJava
- Your x86-64 code must implement all of the statements in the original methods. You may *not* rewrite the method into a different form that produces equivalent results (i.e., restructuring or reordering the code or eliminating method 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.
  - *Exception*: in `min`, your code can simply place the result in `%rax` and does not need to store/load a separate variable `ans` if you do not want to do that. You also do not need to allocate a stack frame for `min` if you don't need it.
- 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.)

You should **remove this page from the exam** and use it while answering this question. **Do not write on this page** – it will not be scanned for grading.

Question 2. (cont.) Write your x86-64 translations of methods `min` and `min3` into x86-64 assembly language below. (Remember to read the above ground rules carefully. You must use regular vtables and dynamic dispatch for method calls, for example, but you do not need to allocate a stack frame or use local variables in `min` if not needed. Be sure to include any necessary assembly language labels in your code, and brief comments are appreciated.)

Question 3. (24 points) Compiler hacking. Now that we've written a MiniJava compiler, we would like to take this opportunity to add a new feature to the language. This time we would like to add a simultaneous assignment statement that looks like this: x, y = a, b;. The meaning of this assignment is that the expressions are evaluated from left to right, then, after the expressions are evaluated, the value of a is assigned to x and the value of b is assigned to y. Some examples:

```
a, b = 0, 1;         // set a = 0 and b = 1
n, p = 17, true;     // set n = 17 and p = true
p, q = q, p;         // swap the values of p and q
x, y = x+1, x;       // increment x and assign the old value of x to y
```

Notice in the last two examples, in particular, it is important that the values are evaluated from left to right and then the assignments are performed after that. So, in the last example, the value of the expression assigned to y is not affected by the new value assigned to x, since the assignments happen left to right after the expressions are evaluated.

To add this statement to MiniJava, we will add the following production to the MiniJava grammar:

Statement ::= Identifier "," Identifier "=" Expression "," Expression ";"

Answer the questions on the following pages about how this statement 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 as the last page of this 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 expression 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.

(continued on next page)

Question 3. (cont.) (b) (6 points) Complete the following new AST class to define an AST node type for the simultaneous assignment statement. You only need to define instance variables and the constructor. Assume that all needed 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 (supplied below), and the `super(pos);` statement at the beginning of the constructor initializes the superclass with this information.)

```
public class SimulAssign extends Statement {
  // add instance variables below
```

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

  public SimulAssign( ____________________________________________

       ____________________________________________, Location pos){
```

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

```
  }
}
```

(continued on next page)

Question 3. (cont.) (c) (5 points) Complete the CUP specification below to define a production for the new simultaneous assignment operator including associated semantic action(s) needed to parse this statement and create an appropriate `SimulAssign` AST node (as defined in part (b) above). You should use any new lexical tokens defined in your answer to part (a) as needed. We have added the necessary additional code to the parser rule for `Statement` to get started.

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

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

```
SimulAssign::=
```

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a simultaneous assignment statement 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 and any type information that needs to be produced for this statement.

Question 3. (cont.) (e) (7 points) Describe the x86-64 code shape for this new simultaneous assignment 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 expression, so using things like `Jfalse`, 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.

Question 4. (16 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 with cheaper ones when possible), common subexpression elimination, and dead code elimination.

The first column of the table below gives the three-address code generated for this sequence of assignment statements: z = 5; x = z*3; x = 5*3+z*3;.

(a) Fill in the second column with the code from the first column after any changes due to constant propagation and folding, copy propagation, and strength reduction. (Note: memory accesses must involve a register (possibly `fp`) and a constant offset only – they cannot be more complex.)

(b) In the third column, check the box "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).

Original Code	After constant & copy prop., folding & strength reduction	"X" if deleted as dead code
t1 = 5		
*(fp + z_offset) = t1		
t2 = *(fp + z_offset)		
t3 = 3		
t4 = t2 * t3		
*(fp + x_offset) = t4		
t5 = 5		
t6 = 3		
t7 = t5 * t6		
t8 = *(fp + z_offset)		
t9 = 3		
t10 = t8 * t9		
t11 = t7 + t10		
*(fp + x_offset) = t11		

Question 5. (20 points) Dataflow analysis – security. Suppose we have a language that allows us to write data to output and also to encrypt data. We would like to ensure that only encrypted data is written to output. We would like to use a dataflow analysis to discover if this is the case by keeping track of which variables contain encrypted data and which ones contain data that is not encrypted.

For this problem, assume that we have a language with the following operations:

x = 17	store unencrypted constant in x
x = y + z	compute sum y+z (decrypting the values read from y and z if necessary). The result is **not** encrypted. The encryption status of variables y and z is not changed.
x = y	ordinary assignment (decrypting the value read from y if necessary). Variable x is **not** encrypted after this assignment. The encryption status of y is not changed.
x = encrypt(y)	encrypt the value of y and assign to x. x **is** encrypted after this assignment
write(x)	write the value of x (encrypted or not) to output

To use a dataflow framework to analyze which variables are encrypted at various points in the program we will define the following sets:

IN(b)	The set of variables that are known to be encrypted on entry to block b
OUT(b)	The set of variables that are known to be encrypted on exit from block b
GEN(b)	The set of all variables that are assigned an encrypted value in block b and not later decrypted by a further assignment before exiting block b
KILL(b)	The set of all variables that are assigned a non-encrypted value in block b and not later assigned an encrypted value before exiting block b

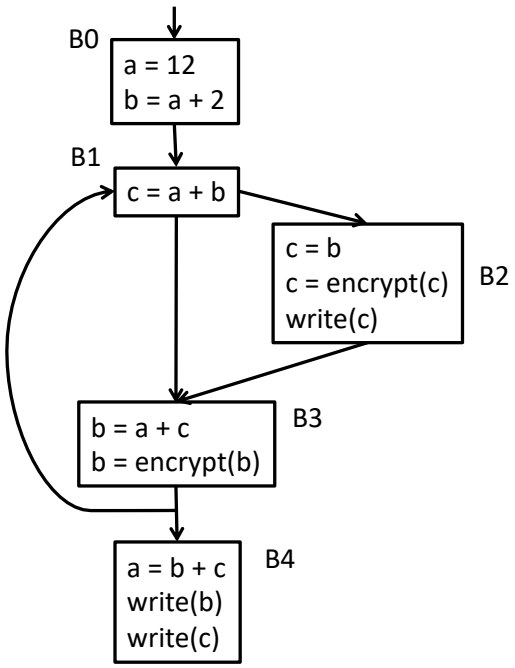
The GEN(b) and KILL(b) sets can be computed once based on the static contents of each block b. The IN(b) and OUT(b) sets need to be computed iteratively during the dataflow analysis.

(a) (6 points) Give appropriate dataflow equations for the IN and OUT sets for a block b in terms of the IN, OUT, GEN, and KILL sets for b as defined above. As with all dataflow problems, these equations will involve some combination of local information about block b itself as well as information about the block's predecessors and successors in the flow graph.

IN(b) =

OUT(b) =

Question 5. (cont.) Now consider the following flow graph and answer parts (b) and (c) below.

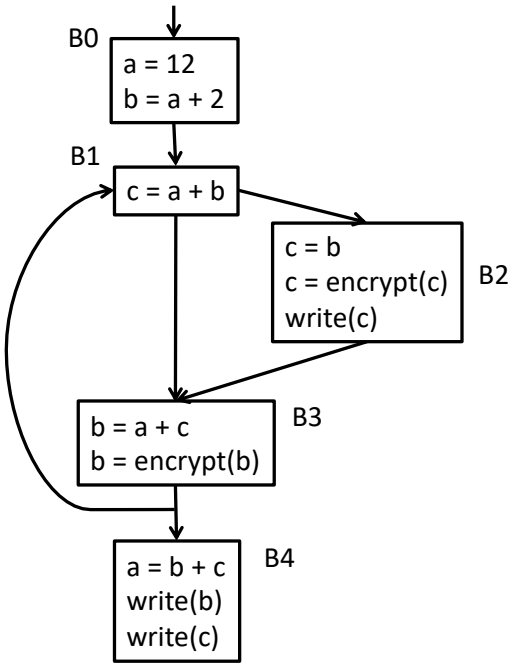


(b) (12 points) Complete the following table using iterative dataflow analysis to identify the encrypted variables in the IN and OUT sets for each block in the above flow graph. You should first fill in the GEN and KILL sets for each block, then iteratively solve for IN and OUT. You can choose whichever direction you wish (forward or backward) to solve the equations. You should assume there are no encrypted variables in the IN set for block B0. (If you run out of room, continue on the blank page provided at the end of the exam and indicate here that your answer is continued there.)

Block	GEN	KILL	IN (1)	OUT (1)	IN (2)	OUT (2)	IN (3)	OUT (3)
B0								
B1								
B2								
B3								
B4								

(c) (2 points) Based on your analysis, can this program write unencrypted data to output? If so, which variables and where in the flowgraph?

The questions on the next page concern the same flowgraph from the previous problem, repeated here for convenience.



The rest of this page contains reference material and definitions that might be useful. You should **remove this page from the exam** and use it while answering the following questions. **Do not write on this page** – it will not be scanned for grading.

Reference Material

Every control flow graph has a unique **start node** s0. (B0 in the above control flow graph)

Node *x* **dominates** node *y* if every path from s0 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 *y* such that

- *x* dominates a predecessor of *y*, but
- *x* does not strictly dominate *y*

Dominance frontier criteria for inserting Φ-functions in SSA graphs: If node x contains the definition of a variable a, then every node in the dominance frontier of x needs a Φ-function for a.

You should **remove this page from the exam** and use it while answering the following questions. **Do not write on this page** – it will not be scanned for grading.

Question 6. (20 points) Dominators and SSA. (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
B0		
B1		
B2		
B3		
B4		

(b) (12 points) Now redraw the flowgraph in SSA (static single-assignment) form. You need to insert the Φ-functions that are required by the dominance frontier criteria (see previous page), 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.

The last 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

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 s = s + a*b + b*c; here's what the instruction selector generated:

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

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 points fairly (☺).

Answer the questions about this sequence of code on the next two pages. You should **remove this page from the exam** and use it while answering this question. **Do not write on this page** – it will not be scanned for grading.

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

(a) (9 points) Draw the interference graph for the temporary variables (t1-t8) in the code on the previous page. You should assume that the code is executed in the sequence given and not rearranged before assigning registers.

(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.

Question 8. (15 points) Forward list scheduling. (a) (7 points) Given the original sequence of instructions on the previous page for the assignment statement s = s + a*b + b*c; , 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.

(b) (8 points) Rewrite the instructions in the order they would be chosen by forward list scheduling (i.e., choosing on each cycle an instruction that is not dependent on any other instruction that has not yet been issued or is still executing). 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 code (LOAD, ADD, etc.) from the original sequence above and the cycle number on which it begins execution. 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.

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

Additional Space for answers, if needed. Please identify the question you are answering here, and be sure to indicate on the question page that the answers are continued here.

Additional space for answers or scratch work.