CSE 401 / M501 18sp Final Exam

June 7, 2018

Name ____

There are 7 questions worth a total of 130 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	/ 20
2	/ 18
3	/ 24
4	/ 16
5	/ 18
6	/ 18
7	/ 16
Total	/ 130

Question 1. (20 points) Virtual madness. Consider the following Java classes.

```
class Uno {
  public void foo() {        System.out.println("Uno.foo"); }
  public void bar() { baz(); System.out.println("Uno.bar"); }
  public void baz() { foo(); System.out.println("Uno.baz"); }
}

class Duo extends Uno {
  public void baz() {        System.out.println("Duo.baz"); }
  public void zap() { bar(); System.out.println("Duo.zap"); }
}
```

When class `Uno` was compiled, the compiler picked the following trivial layout for objects of type `Uno` (since only a vtable pointer is needed), and generated the following vtable for that class:

Object Layout

offset	field
+0	vtable pointer

Vtable layout

```
Uno$$:   .quad 0          # no superclass
         .quad Uno$bar    # +8
         .quad Uno$baz    # +16
         .quad Uno$foo    # +24
```

Notice that the method pointers in the vtable are in alphabetical order, not in the order that the method declarations appear in class `Uno`. This is simply an arbitrary choice made by the compiler and doesn't affect the operation of the program.

(a) (4 points) Show the vtable layout for class `Duo` using the same format used above for class `Uno`. Be sure to properly account for the methods inherited from class `Uno`, including those that are overridden.

Question 1. (cont.) (b) (8 points) Now, suppose we have a program that uses these two classes. For each of the following, write the output that is produced when these sequences of statements are executed. If there is some sort of error, describe (briefly) the problem. Each part of this question is independent of the others.

(i)
```
Uno u = new Uno();
u.baz();
```

(ii)
```
Duo d = new Duo();
Uno x = d;
x.baz();
```

(iii)
```
Duo d = new Duo();
d.zap();
```

(iv)
```
Duo d = new Duo();
Uno x = d;
x.zap();
```

(continued on next page)

Question 1 (cont.) (c) (8 points) Now for the truly evil, and becoming somewhat traditional, "trick" question. ☺ What would happen if we changed the compiler so the vtables included all of the methods in the class hierarchy, in the order declared, using this layout:

```
Uno$$:   .quad 0              Duo$$:     .quad Uno$$          # superclass
         .quad Uno$foo                   .quad Uno$foo        # +8
         .quad Uno$bar                   .quad Uno$bar        # +16
         .quad Uno$baz                   .quad Uno$baz        # +24
                                         .quad Duo$baz        # +32
                                         .quad Duo$zap        # +40
```

As before, every object contains a pointer to the vtable for its class. Further, there are no changes to the dynamic dispatch code generated by the compiler to call methods, which continues to use the pointer in each object to find the object's vtable and the pointers found there to call methods. The compiled code does, of course, use the new offsets of method pointers in vtables in appropriate ways.

What output is produced when the statements from the previous part of the question are executed with these vtable layouts? Hints: Remember that the code for methods in each class is compiled using the known vtable information for the class, and calling code will use the declared types of variables to decide which vtable offsets contain the appropriate pointers to methods. Your answers may well be different from the expected output that would be produced by a correct Java compiler

(i)
```
Uno u = new Uno();
u.baz();
```

(iii)
```
Duo d = new Duo();
d.zap();
```

(ii)
```
Duo d = new Duo();
Uno x = d;
x.baz();
```

(iv)
```
Duo d = new Duo();
Uno x = d;
x.zap();
```

Question 2. (18 points) x86-64 coding. Consider the following class:

```
class Foo {
  public int fn(int a, int b) {
    if (a == 0)
      return 0;
    else
      return b + this.fn(a-1, b);
  }
}
```

On the next page, translate method `fn` 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. Since class `Foo` has only one method, `fn`, you should assume that the vtable layout for the class has a single pointer to this method at offset +8.

`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, even though this is not strictly required by the x86-64 specification.
- 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 method. You may *not* rewrite the method into a different form that produces equivalent results (i.e., restructuring or reordering the code). 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.)

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 translation of method `fn` into x86-64 assembly language below.

Question 3. (24 points) Compiler hacking. Several programming languages, including PERL, Ruby, and PHP include a three-way comparison operator <=>. This operator evaluates to either -1, 0, or +1 depending on the values of its operands. If *e1* < e2, then *e1* <=> *e2* evaluates to -1; if *e1* == *e2*, then then the value is 0; and if *e1* > *e2*, then *e1* <=> *e2* has the value +1.

We would like to add this operator to our MiniJava compiler and language to compare integer (`int`) values. To do this, we will add the following production to the MiniJava grammar:

Expression ::= Expression "<=>" Expression

Answer the questions on the following pages about how this 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 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) (5 points) Complete the following new AST class to define an AST node type for the new `<=>` 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 `Comp3` constructor below is used to properly initialize the superclass with this information.)

```
public class Comp3 extends Exp {
  // add instance variables below

  // constructor – add parameters and method body below

  public Comp3 ( ______________________________________________ ){
   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 <=> operator including associated semantic action(s) needed to parse an expression containing <=> and create an appropriate `Comp3` 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 `Exp` to get started.

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

```
Exp ::= ...
     | Comp3:e  {: RESULT = e; :}
    ...
     ;
```

```
Comp3 ::=
```

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

Question 3. (cont.) (e) (8 points) Describe the x86-64 code shape for this added <=> 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 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 $\mathtt{J}_{\mathtt{false}}$, 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.

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), and dead code elimination.

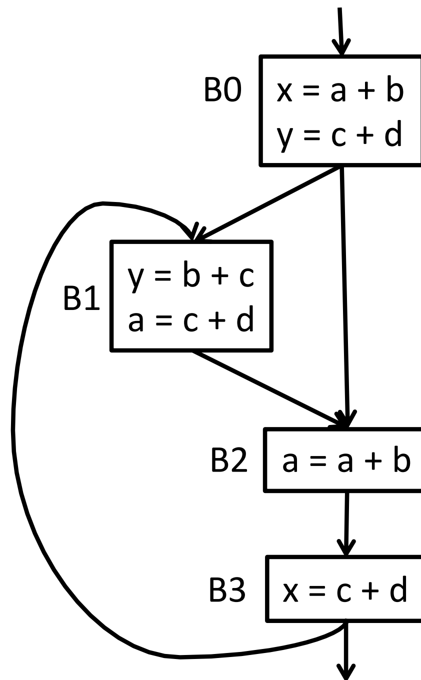
The first column of the table below gives the three-address code generated for the assignment statement `s = s + a[i]*b[i];`. This code assumes array elements occupy 8 bytes each.

(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 = *(fp + soffset) // s		
t2 = *(fp + ioffset) // i		
t3 = t2 * 8		
t4 = fp + t3		
t5 = *(t4 + aoffset) // a[i]		
t6 = *(fp + ioffset) // i		
t7 = t6 * 8		
t8 = fp + t7		
t9 = *(t8 + boffset) // b[i]		
t10 = t5 * t9 // a[i]*b[i]		
t11 = t1 + t10 // s + …		
*(fp + soffset) = t11 // s = …		

The next two questions concern the following control flow graph.



The rest of this page contains reference material and definitions that might be useful when answering some of the remaining questions.

You should **remove this page from the exam** and use it while answering the remaining 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.

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* ≠ *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*

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

Question 5. (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 on the previous page.

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

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

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 change.

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

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 5. (cont.) (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		
B1		
B2		
B3		

(b) (10 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	
B1	
B2	
B3	

Question 6. (18 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 nodes that dominate it, the node that is its immediate dominator (if any), and the nodes that are in its dominance frontier (if any):

Node	Dominators	IDOM	Dominance Frontier
B0			
B1			
B2			
B3			

(b) (10 points) Now redraw the flowgraph in SSA (static single-assignment) form. You need to insert appropriate Φ-functions where they are required and, once that is done, add appropriate version numbers to all variables that are assigned in the flowgraph. You should not insert extra Φ-functions at the beginning of a block if they clearly would not be appropriate there, but we will not penalize a few extraneous Φ-functions if they are correct, but possibly not needed. 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.

Question 7. (16 points) First things first (or last). We'd like to use forward list scheduling to pick a good order for executing a sequence of instructions. For this problem, assume that we're using the same hypothetical machine that was presented in lecture and in the textbook examples. Instructions are assumed to take the following number of cycles:

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2

Given the assignment statement `z = (x+y) * (a*b);` , our compiler's instruction selection phase initially emits the following sequence of instructions:

```
a. LOAD   r1 <- x
b. LOAD   r2 <- y
c. ADD    r3 <- r1, r2     // x + y
d. LOAD   r4 <- a
e. LOAD   r5 <- b
f. MULT   r6 <- r4,r5      // a * b
g. MULT   r7 <- r3, r6     // (x + y) * (a * b)
h. STORE  z <- r7
```

Answer the following questions on the next page. 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.

(a) (7 points) 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 it's 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) (7 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.

(c) (2 points) At the bottom of the next page, write down the number of cycles needed to completely execute the instructions in the original order and the number of cycles needed by the new schedule.

Question 7. (cont.) (a) and (b) Draw the precedence diagram and write the new instruction schedule (sequence) below. Then fill in part (c) at the bottom of the page.

(c) Fill in: Number of cycles needed to completely execute all instructions in the original schedule _____

Number of cycles needed to completely execute all instructions in the new schedule _____

Have a great summer and best wishes for the future!
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.