CSE 401/M501 19au Final Exam

December 10, 2019

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.

There are several full pages in the exam that contain problem descriptions and/or reference information. Those cannot be removed because the exam is printed on both sides of each page. However, a second copy of those reference pages is included at the end of the exam, along with a copy of the MiniJava grammar, and those extra copies can be removed for convenience during the exam.

A blank page is provided **at the end** of the exam if you need extra space for answers or scratch work. If you write any answers on that page, please be sure to indicate on the original question page that your answers are continued on that page, and label the answers on the additional page.

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	/ 25
4	/ 16
5	/ 20
6	/ 18
7	/ 15
8	/ 16
Total	/ 140

CSE 401/M501 19au Final Exam 12/10/19

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

_____ Eliminate re-calculation of the expression `a+b` if it is guaranteed to be available at some point in the program that calculates it again.

_____ Decide to use a `leaq` instruction to combine two addition operations into a single instruction

_____ Report the existence of a non-terminating ("infinite") loop in a program

_____ Report the error in the array element reference `a[x<y]` (assuming `a` is an array of `int` values and that `x` and `y` also are `int` variables)

_____ Report that `#` is not a legal operator in the expression `x#y`, where `x` and `y` are `int` variables

_____ Report that <=> is not a legal operator in a full Java program

_____ Ensure that code for a method that returns a reference (pointer) places its result in `%rax`

_____ Report an error because the size used to allocate a new array is a negative integer value

_____ Arrange the instructions in a basic block to do LOAD instructions early so their execution overlaps other computation that can be done while the LOADs are in progress

_____ In the loop `for (i=0; i<n; i++) { ... a[i] ... }`, rewrite the code to use pointers instead: `for (p=&a[0]; p<p+n; p++) { ... *p ... }`

Do not remove this page from the exam. However, there is an extra copy of this page at the end of the exam that you can remove for reference while you are working if that is helpful.

Question 2. (20 points) A bit of x86-64 coding. Here is a small C function that adds the elements of an integer array in a somewhat tedious manner:

```
// return the sum a[i]+...+a[n-1]
// i.e., asum(a,0,n) returns the sum of an n element array a[0..n-1]
int asum(int a[], int i, int n) {
  int result;
  if (i == n)
    result = 0;
  else
    result = a[i] + asum(a, i+1, n);
  return result;
}
```

On the next page, translate this function into x86-64 assembly language. You should use the standard 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 when a stack frame is allocated. 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):

- 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
- 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). You should allocate space for the local variable `result` and generate appropriate load and store instructions to implement the assignment and return statements that use it. (And, of course, you can allocate additional space in the stack frame as needed to store other data.). 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 `asum` into x86-64 assembly language below. Remember to read and follow the above ground rules carefully. Brief comments are appreciated. Original code repeated below for convenience:

```
int asum(int a[], int i, int n) {
  int result;
  if (i == n)
    result = 0;
  else
    result = a[i] + asum(a, i+1, n);
  return result;
}
```

Question 3. (25 points) Compiler hacking. As is often the case, now that our MiniJava compiler is done, one of our customers would like us to add a new "feature". The customer has a bunch of old code that includes a different kind of `if` statement, and they would like us to add this to MiniJava.

The simplest form of the new `if` statement has one condition and looks like this:

```
if x > y => temp = x; x = y; y = temp; fi
```

The statement begins with `if` and ends with `fi`. A => arrow is used to separate the condition from the statement(s) that are executed when it is true. If the condition is false, execution of everything between => and `fi` is skipped.

More interesting is that an `if` statement can contain several condition/statements pairs separated by `[]` (a box made up of adjacent left and right square brackets). For example, the following `if` sets `sign` to -1, 0, or +1, depending on whether `n` is negative, 0, or positive:

```
if n > 0 => sign = 1;
[] n < 0 => sign = 0-1;  // i.e., -1, but we don't have unary minus in MiniJava
[] true => sign = 0;
fi
```

If the `if-fi` statement contains more than one condition/statement sequence group separated by `[]` boxes, the conditions are evaluated in order from beginning to end. When a condition is found that evaluates to true, the statement(s) to the right of the corresponding => are executed, and then execution of the entire `if-fi` statement is done.

Fine print: there has to be at least one *condition* => *statements* pair between `if` and `fi`. The list of *statements* following the => arrow cannot be empty.

Answer the questions below about how this new `if` 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 at the end of the exam if you need to refer to it.

(a) (4 points) What new lexical tokens, if any, need to be added to the scanner and parser of our MiniJava compiler to add this new `if` 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.

(continued on next page)

Question 3. (cont.) (b) (6 points). Give an unambiguous context-free grammar rule or rules to add this new `if-fi` conditional statement to the MiniJava grammar for *Statement*. Your answer should include terminals and non-terminals as needed, including the new terminal symbols identified in your answer to part (a), and can include additional new grammar rules for existing or new non-terminals as needed. You only need to give the additions and changes to the MiniJava grammar. You do not need to write CUP specifications or other MiniJava code, but the context free grammar rules you write here should be directly usable as the basis for appropriate CUP rules that would parse the new `if-fi` statement.

Hint: it might be useful to think about how things like method parameter lists, which are comma-separated lists of expressions, are specified using CUP grammar productions.

(c) (5 points) Describe the changes or additions that need to be made to the MiniJava Abstract Syntax Tree (AST) classes or node definitions to add this new `if-fi` statement to the language. You should not include specific Java code or AST class definitions, but you should precisely describe the new or changed node types and their contents so that it is obvious how they would be implemented.

(continued on next page)

Question 3. (cont.) (d) (4 points) What additions or changes need to be made to the static semantics / type checking part of the compiler to verify that an `if-fi` statement is correct? Again, you don't need to provide specific visitor method code or anything like that – just describe what type or other information needs to be produced or checked for this new statement.

(e) (6 points) Describe the x86-64 code shape for this new `if-fi` 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. Your answer should show the code shape needed for a `if-fi` statement like the following one with two condition/statement pairs:

```
if  cond1 => stmts1
[]  cond2 => stmts2
fi
```

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 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.

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 like * 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 statement:
y[i] = a*x[i] + y[i]

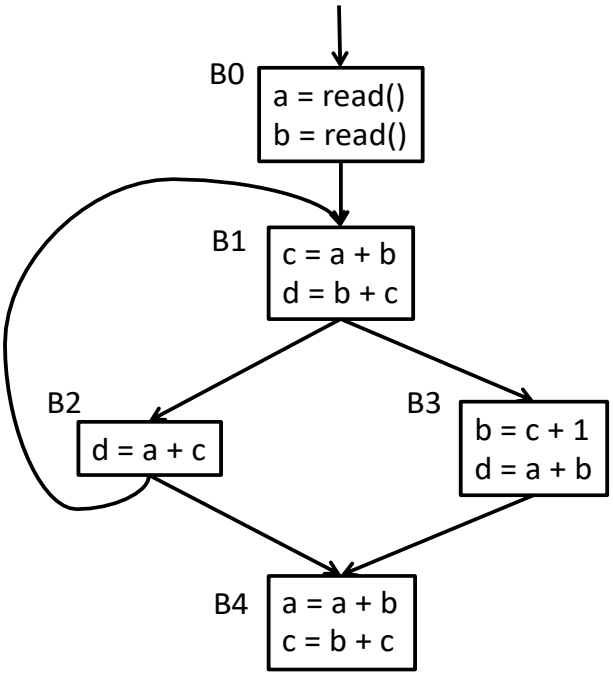
(a) Fill in the second column with the code from the first column after any changes due to constant propagation and folding, copy propagation, strength reduction, and common subexpression elimination, but before any dead code elimination. (Notes: memory reference addresses can use a register (t*i* or `fp`) and a constant offset only – they cannot be more complex. Also note that the arrays `x` and `y` are assumed to be local variables in the current stack frame.)

(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 prop./folding & copy prop., strength reduction, and CSE (copy original code if no change)	"X" if deleted as dead code
t1 = *(fp + aoffset) // a		
t2 = *(fp + ioffset) // i		
t3 = t2 * 8 // 8*i		
t4 = fp + t3		
t5 = *(t4 + xoffset) // x[i]		
t6 = t1 * t5 // a*x[i]		
t7 = *(fp + ioffset)		
t8 = t7 * 8		
t9 = fp + t8		
t10 = *(t9 + yoffset) // y[i]		
t11 = t6+t10 // a*x[i] + y[i]		
t12 = *(fp + ioffset)		
t13 = t12 * 8		
t14 = fp + t13		
*(t14 + yoffset) = t11 // y[i] = …		

Do not remove this page from the exam. However, there is an extra copy of this page at the end of the exam that you can remove for reference while you are working if that is helpful.

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 the next few questions.

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*.

Do not remove this page from the exam. However, there is an extra copy of this page at the end of the exam that you can remove for reference while you are working if that is helpful.

Question 5. (20 points) Dataflow analysis – available expressions.

Recall 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

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

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.

Once you have calculated the AVAIL sets, be sure to answer the question at the bottom about whether there are any redundant expression calculations that can be eliminated.

Hint: notice that there are only a limited number of expressions calculated in this flowgraph. These include a+b, b+c, a+c, and so forth. So all of the AVAIL, NKILL, and DEF sets for the different blocks will contain some, none, or all of these expressions. In your answer it is okay to write something like "all expressions that don't include *b*" rather than listing all of them – or you can list all of them if you prefer.

Question 5. (cont.) (a) (8 points) For each of the blocks B0, B1, B2, B3, and B4, write their DEF and NKILL sets in the table below.

Block	DEF	NKILL
B0		
B1		
B2		
B3		
B4		

(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	
B4	

(c) (2 points) Are there any *redundant expressions* in the flowgraph that can be eliminated? (i.e., expressions that do not need to be recomputed because they are available at that point in the flowgraph as discovered by this analysis.) If so, identify the expressions and the specific locations (blocks) where they are redundant.

Question 6. (18 points) Dominators and SSA. (a) (8 points) Using the same control flow graph from the previous problem, complete the following table. For each node, list the node(s) that it strictly dominates, and list the nodes in its dominance frontier (if any):

Node	Strictly Dominates:	Dominance Frontier
B0		
B1		
B2		
B3		
B4		

(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 insert all of the Φ-functions that are required by the dominance frontier criterion, but no others. 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.

Do not remove this page from the exam. However, there is an extra copy of this page at the end of the exam that you can remove for reference while you are working if that is helpful.

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
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[i] = a*x[i]; here's what the instruction selector generated:

```
a.  LOAD   t1 <- a              // t1 = a
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]
g.  MULT   t7 <- t1, t6         // t7 = a*x[i]
h.  LOAD   t8 <- y              // t8 = address of y[] array
i.  ADD    t9 <- t4, t8         // t9 = address of y[i]
j.  STORE  MEM[t9] <- t7        // store y[i]
```

(Note that this code assumes that variables x and y contain pointers to arrays, 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 points fairly (☺).

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

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

(a) (9 points) Draw the interference graph for the temporary variables (t1-t9) 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. (16 points) Forward list scheduling. (a) (8 points) Given the original sequence of instructions on the previous page for the assignment statement y[i] = a*x[i], 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.

(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 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.