CSE 401 Final Exam

March 14, 2017
Happy π Day! (3/14)

Name ____

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	/ 15
2	/ 15
3	/ 14
4	/ 14
5	/ 16
6	/ 14
7	/ 14
8	/ 12
9	/ 6
Total	/ 120

Question 1. (15 points) Virtual madness. Consider the following Java program.

```
public class Vtables {
  public static void main(String[] args) {
    Derived d = new Derived();
    Base b = d;
    b.one();
    System.out.println("***");
    b.two();
    System.out.println("***");
    d.four();
  }
}

class Base {
  public void one()   {        System.out.println("Base.one"); }
  public void two()   { one(); System.out.println("Base.two"); }
  public void three() {        System.out.println("Base.three"); }
}

class Derived extends Base {
  public void one()  {        System.out.println("Derived.one"); }
  public void four() { two(); System.out.println("Derived.four"); }
}
```

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

offset	Object Layout field
+0	vtable pointer

Vtable layout

```
Base$$:  .quad 0              # no superclass
         .quad Base$one       # +8
         .quad Base$two       # +16
         .quad Base$three     # +24
```

Answer questions about this program on the next page. You may remove this page for reference while you are working.

Question 1. (cont.) (a) (4 points) Show the vtable layout for class `Derived` using the same format used on the previous page for class `Base`. Be sure to properly account for the methods inherited from class `Base`, including those that are overridden.

(b) (5 points) What output is produced when we execute the `main` method in class `Vtables`? (i.e., what happens when we execute the program?)

(continued on next page)

Question 1 (cont.) (c) (6 points) OK, now for the truly evil "trick" question. ☺

The new intern, I. M. Pedantic, has decided that things would be much easier to understand if vtables were neatly organized in alphabetical order, instead of the order previously used by the compiler. So the compiler has been changed to generate the following vtables for classes `Base` and `Derived`:

```
Base$$:  .quad 0               Derived$$:  .quad Base$$          # superclass
         .quad Base$one                    .quad Derived$four    +8
         .quad Base$three                  .quad Derived$one     +16
         .quad Base$two                    .quad Base$three      +24
                                           .quad Base$two        +32
```

As before, every object contains a pointer to the vtable for its class. Further, Pedantic did not change 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 question is, what does this program print if the vtables contents are changed this way?

Hint 1: remember that the code for methods in each class is compiled using the known vtable information for the class, and code generated for the main method will use the declared types of variables to decide which vtable offsets contain the appropriate pointers to methods.

Hint 2: Your answers may well be different from the expected output that would be produced by a correct Java compiler. But there is a single answer to this question.

Question 2. (15 points) Compiler hacking: a question of several parts. Clearly our MiniJava compiler is missing several useful Java features. One of our customers would like us to add the `?:` ternary operator that exists in Java, C, C++, and other languages. The meaning of *e1* ? *e2* : *e3* is that *e1* is first evaluated. If it is true, then *e2* is evaluated and that is the value of the expression. If *e1* is false, then *e3* is evaluated and that is the value of the overall expression. To add this to MiniJava, we need to add one new production to the MiniJava grammar:

Expression ::= Expression "?" Expression ":" Expression

(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; you don't need to give JFlex or CUP specifications or code. The full MiniJava grammar is attached as the last page of this exam if you need to refer to it.

(b) (3 points) What changes are needed to the MiniJava abstract syntax tree (AST) data structures to add this new expression to MiniJava? Again, you do not need to give any Java or CUP code, just describe the changes (what kinds of new or changed nodes, what children would they have, etc.).

(c) (4 points) What checks need to be performed to verify that there are no type compatibility or other semantic errors for this new expression? Hint: remember that MiniJava is statically typed, so this new expression must have a definite type like all other expressions.

(continued next page)

Question 2. (cont.) (d) (6 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. Also, if it matters, you can assume that the stack is aligned on a 16-byte boundary at the beginning of the code sequence, and, if you change the size of the stack, you need to be sure this alignment is preserved if a part of the `?:` expression could contain a method call.

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 $\texttt{J}_{\texttt{false}}$, 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 3. (14 points) x86-64 coding. Consider the following class:

```
class Foo {
  public int maxv(int x, int y) {
    if (x < y)
      return x;
    else if (y < x)
      return this.maxv(y, x);
    else
      return x;
  }
}
```

On the next page, translate method `maxv` 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, `maxv`, 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 may detach this page from the exam if that is convenient.)

Question 3. (cont.) Write your translation of method `maxv` into x86-64 assembly language below.

Question 4. (14 points) A little optimization. For this question we'd like to perform local constant propagation and folding, plus copy propagation (reuse values that are already present in another temporary t*i* when possible) and dead code elimination.

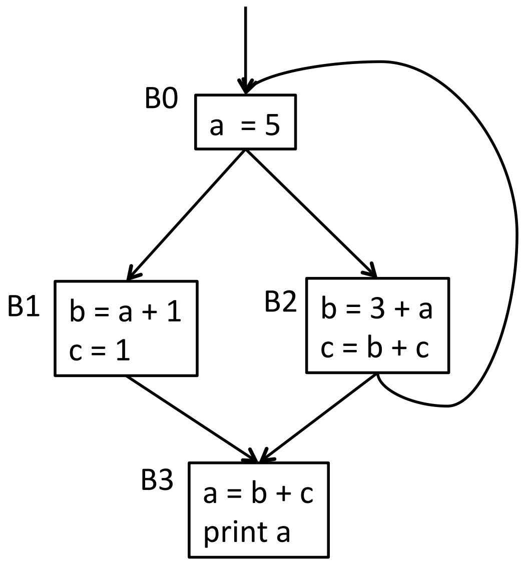
The first column of the table below gives the three-address code generated for the statements `x = 5; a[i] = a[i]+ x;` .

(a) Fill in the second column with the code from the first column after any changes due to constant propagation and folding (compile-time arithmetic), and copy propagation. (Note: memory accesses must involve a temporary t*i* and a memory address; they cannot load/store a constant like 5 directly.)

(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 and copy optimizations in part (a).

Original Code	After constant propagation & folding & copy propagation	"X" if deleted as dead code
t1 = 5		
*(fp + xoffset) = t1 // x = …		
t2 = *(fp + ioffset) // i		
t3 = t2 * 4		
t4 = fp + t3		
t5 = *(t4 + aoffset) // a[i]		
t6 = *(fp + xoffset) // x		
t7 = t5 + t6 // a[i] + x		
t8 = *(fp + ioffset) // i		
t9 = t8 * 4		
t10 = fp + t9		
*(t10 + aoffset) = t7 // a[i]=…		

The next two questions concern the following control flow graph:



Question 5. (16 points) Dataflow analysis. Recall from lecture that *live-variable* analysis determines for each point *p* in a program which variables are live at that point. A live variable *v* at point *p* is one where there exists a path from point *p* to another point *q* where *v* is used without *v* being redefined anywhere along that path. The sets for the live variable dataflow problem are:

use[*b*] = variables used in block *b* before any definition
def[*b*] = variables defined in block *b* and not later killed in *b*
in[*b*] = variables live on entry to block *b*
out[*b*] = variables live on exit from block *b*

The dataflow equations for live variables are

$$\text{in}[b] = \text{use}[b] \cup (\text{out}[b] - \text{def}[b])$$
$$\text{out}[b] = \cup_{s \in \text{succ}[b]} \text{in}[s]$$

On the next page, calculate the use and def sets for each block, then solve for the in and out sets of each block. A table is provided with room for the use and def sets for each block and up to three iterations of the main algorithm to solve for the in and out sets. If the algorithm does not converge after three iterations, use additional space below the table for additional iterations.

Then remember to answer the undefined variable question at the bottom of the page.

Hint: remember that live-variables is a backwards dataflow problem, so the algorithm should update the sets from the end of the flowgraph towards the beginning to reduce the total amount of work needed.

You may remove this page for reference while working these problems.

Question 5. (cont.) (a) (14 points) Write the results of calculations for live variables in the chart below. Use the rest of the page for extra space if needed, then remember to answer part (b) at the bottom!

Block	use	def	out (1)	in (1)	out (2)	in (2)	out (3)	in (3)
B3								
B2								
B1								
B0								

(b) (2 points) One use of live variable analysis is to detect potential use of uninitialized variables in a program. Are there any potentially uninitialized variables in this flowgraph, and if so which ones, where, and why? (i.e., justify your answer using the information from the analysis you have done above.)

Question 6. SSA. (14 points) Redraw the flowgraph used in the previous problem in SSA (static single-assignment) form. You need to insert appropriate Φ-functions where they are required and add appropriate version numbers to all variables. Do not insert Φ-functions at the beginning of a block if they clearly would not be appropriate there, but we will not penalize occasionally extra Φ-functions if they are inserted correctly. 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. (14 points) First things first. 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 `y = (a*x + b)*x + c`, our compiler's instruction selection phase initially emits the following sequence of instructions:

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

Answer the following questions on the next page. You can remove this page for convenience if you like.

(a) (6 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) (6 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 _____

Question 8. (12 points) Register allocation. Considering the following code:

```
a = read();
b = read();
if (a < b) {
   c = read();
   while (c < a) {
      c = c + 1;
   }
   print(c);
} else {
   d = a + b;
   print(d);
}
```

On the next page write your answer to the following questions. You can remove this page from the exam for convenience if you wish.

(a) (8 points) Draw the interference graph for the variables in this code. You are not required to draw the control flow graph, but it might be useful to sketch that to help find the solution and to leave clues in case we need to assign partial credit.

(b) (4 points) Give an assignment of variables to registers using the minimum number of registers possible, based on the information in the interference graph. You do not need to go through the steps of the graph coloring algorithm explicitly, although that may be helpful as a guide to assigning registers. If there is more than one possible answer that uses the minimum number of registers, any of them will be fine. Use R1, R2, R3, … for the register names.

write

your

answer

on

the

next

page ->

Question 8. (cont.) Draw the interference graph and write your register assignments below the graph.

Question 9. (6 points, 2 each) Almost done. Time take out the garbage at the end of the party. ☺

Two fundamental concepts in a garbage collector are the notions of which objects are *reachable* and which ones are *live*. Give a brief (one or two sentences please) definition of each of these terms:

(a) *Reachable*

(b) *Live*

(c) What is the relationship between these two concepts? In particular, are they the same, or, if not, what is the difference and does either concept imply the other? (Again, a brief sentence or two at most, please.)

Have a great spring break!
The CSE 401 staff