CSE 401/M501 25sp Final Exam

June 10, 2025

Name _____ **UW netid** _____@uw.edu
(print legibly)

There are 8 questions worth a total of 112 points. Please budget your time so you get to all of the questions. Keep your answers brief and to the point.

Two extra **blank pages** are provided **at the end** of the exam if you need extra space for answers or for scratch work. If you write any answers on those pages, please be sure to indicate on the original question page(s) that your answers are written or continued on an extra page, and label the answers on the extra page.

A copy of the MiniJava grammar is provided separately for reference if you need it. Please return that page for recycling/reuse when you are done.

This exam is closed book, closed notes, closed electronics, closed neighbors, open mind, ..., however you may have two 5x8 notecards with whatever hand-written information you wish written on both sides.

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 do our best to figure out the meaning of what you write.

1	/ 10
2	/ 14
3	/ 14
4	/ 24
5	/ 16
6	/ 18
7	/ 14
8	/ 2
Total	/ 112

Question 1. (10 points) Compiler Passes & Optimization – An array of questions.

(a) (6 points) Consider the following code snippets, each of which defines and assigns an array. For each snippet, identify *which compiler pass/phase* will detect an error (if any) **Assume that all code is MiniJava!** (not standard Java, nor some other language)

Please use the following abbreviations: (n.b. not all of these will be used)

scan – scanner
parse – parser
check – checker
mid – compiler middle (translating from ASTs to 3AC, SSA, optimization passes, dataflow)
back – compiler backend (instruction selection, scheduling & register allocation)
run – runtime (i.e., when the compiled code is executed)
none – there is no error

_____ `int[] arr;  arr = new int[false];`

_____ `int[] arr;  arr = new int [1 % 5];`

_____ `int[] arr;  arr = new int [1];`

_____ `int[] arr;  arr = new boolean[1];`

_____ `int[] arr;  arr = new int[0-1];`

_____ `int[] arr;  arr = new int[1.5];`

(b) (4 points) Consider the following MiniJava class in isolation.

```
class Foo {
    int[] arr;
    public int get_init() {
        arr = new int[1];
        arr = new int[2];
        return arr;
    }
}
```

Which optimization can a compiler safely make to this code? And which analysis enables that optimization?

Question 2. (14 points) Scoping, and VTables — I Object! It's not as easy as one, two, …

(a) (6 points) Scope & Dispatch. Here is a peculiar **Java** (not MiniJava) program consisting of a main class and two subclasses.

```
public class one {
    int one;
    int two;
    void init() {
        one = 1;
        two = 2;
    }
    int one(int one) {
        one = two;
        two = one;
        return 1;
    }
    int two(int two) {
        this.one = this.two;
        this.one(two);
        return one;
    }
}
public class two extends one {
    int two;
    int one(int one) {
        this.two = one;
        this.one = one + two;
        return one;
    }
    int three(int three) {
        return one + two + three;
    }
}
```

```
public class Main {
  static void main(String[] args) {
    one one = new one();
    one.init();
    System.out.println(one.one(1));
    System.out.println(one.two(2));

    two two = new two();

    one = two;
    one.init();
    System.out.println(one.one(1));
    System.out.println(one.two(2));
    System.out.println(two.three(3));
  }
}
```

What output does this program produce when we compile it and then execute the main method in class Main? (The program does compile and execute without errors.) (If you wish, you may supply additional information about the state of the objects after each println call. We may award partial credit on that basis, and it may help you keep everything straight.)

Question 2. (cont.)

(b) (4 points) VTables. When the class `one` was compiled, the compiler picked the following vtable layout for the class.

```
                vtable layout       offset
one$$:          .quad 0             # no superclass
                .quad one$one       # +8
                .quad one$two       # +16
```

Below, show an appropriate Vtable layout for class `two`, in the same format used above for class `one`. Be sure to properly account for inherited methods in the Vtable layout.

(c) (4 points) Object Instance Layout. Assuming we have the same memory management scheme as MiniJava, how many bytes of heap memory would each instance of a `two` take up?

Question 3. (14 points) A bit of x86-64 coding – Perfectly Average.
The C function below calculates the signed difference between the mean and median of an integer array. For simplicity of implementation, we assume the input array is of odd length and is sorted.

```
/* Assume arr is sorted and odd-length */
double med_mean_difference(int[] arr, int length) {
    int middle = length / 2;
    int median = arr[middle];

    return median - mean(arr, length);
}

int mean(int[] arr, length) {
    // returns the closest int to the mean of all numbers in arr
}
```

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

- All values, including pointers and ints, 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.
- Please assume that the length of the array is odd, and its contents are sorted.
- This is simple C code, not a Java method, so there is no this pointer.
- The `mean()` function and set up a stack frame consistent with local variables declared in the `med_mean_difference()` function.
- Rather than trying to remember the sign division stuff from class, please use **`divq a,b`** for your answer, where **`divq a,b`** computes and places a divided by b in `%rax` with the remainder placed in `%rdx`.
- 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 3. (cont.) Write your x86-64 translation of function `med_mean_difference()` below.

Remember to read and follow the above ground rules carefully, including managing registers properly and using the correct argument registers for function calls, and creating a local stack frame to hold the local variables correctly while calling `mean()`. Your code should include a translation of all of the code in the original function. Brief comments are appreciated. Original code repeated below for convenience:

```
/* Assume arr is sorted and odd-length */
double med_mean_difference(int[] arr, int length) {
    int middle = length / 2;
    int median = arr[middle];

    return median - mean(arr, length);
}

int mean(int[] arr, length) {
    // returns the closest int to the mean of all numbers in arr
}
```

Question 4. (24 points) Add a Feature – I am the Walrus, Coo coo ca choo!
You have just launched the first version of your MiniJava compiler, and it was a huge success! Users are already wanting more features. One user, who is an avid Python programmer, is requesting an "assignment expression" construct using the walrus operator (`:=`).

The semantics of this operator are as follows for an expression of the form

<variable> := <expression>

1. The *<expression>* on the right hand side is evaluated.

2. The value of the *<expression>* is assigned to the *<variable>* on the left hand side.

3. The overall assignment expression returns the value of the RHS *<expression>*.

Your client is only requesting the assignment expression to offer the same level of support as standard assignment, so array assignment doesn't need to be handled.

Answer the questions below about how this new expression 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 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 assignment expressions to the original MiniJava language? Just describe any necessary changes and new token(s) needed and their name(s). You don't need to give JFlex or CUP specifications or code in this part of the question, 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 4. (cont.) (b) (6 points) Complete the following new AST class to define an AST node type for this new expression assignment. 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 constructor below is used to properly initialize the superclass with this information.)

```
public class AssignExp extends Exp {
    // add any needed instance variables below
```

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

    public AssignExp (_____________________________________________){
```

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

```
    }
}
```

(continued on next page)

Question 4. (cont.) (c) (5 points) Complete the CUP specification below to define a production for this new assignment expression, including associated semantic action(s) needed to parse the new expression 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 that already would exist in the compiler scanner and parser if you need them. We have added additional code to the parser rule for `Expression` below so the CUP specification for the new expression assignment can be written as an independent grammar rule with separate parser actions.

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

```
Expression ::= ...
     | AssignExp:e  {: RESULT = e; :}
    ...
     ;
AssignExp ::=
```

(d) (4 points) Describe the checks that would be needed in the checking part of the compiler to verify that a program containing this new assignment expression 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 statement to verify it is used correctly.

(continued on next page)

Question 4. (cont.) (e) (7 points) Describe the x86-64 code shape for this new assignment expression, as it 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 in our MiniJava project.

For example, you can use `<...>` as placeholders for code generated by a child expression. If needed, you should assume that the code generated for a child expression will leave the resulting value of that expression in `%rax`, as in our MiniJava project.

You may assume that the variable you are assigning to is a method local variable.

Use Linux/gcc x86-64 instructions and assembler syntax when needed. If you need to make any additional assumptions about code generated by the rest of the compiler you should state them.

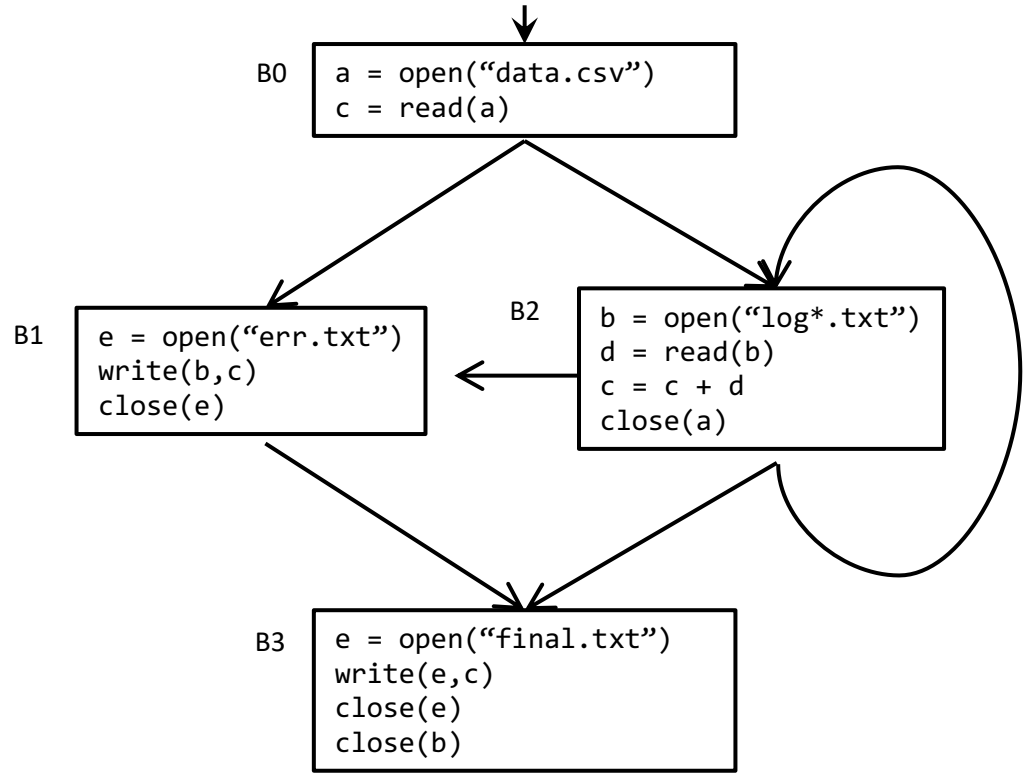
While we studied program analysis in order to justify the safety of compiler optimizations, analyses are also very useful for detecting and warning users about likely bugs. For the next question we will use dataflow to try to detect unclosed files.

The statements in the program include those that open files and those that close files. We would like to ensure that all files are closed before the end of execution. We want to use dataflow analysis to discover if there are any unclosed files by keeping track of which files may be open at various points in the program.

For the next two problems, assume that our language has two types: file objects and strings. The language has the following operations:

- `f = open(path):` opens the given file and assigns it to `f` (`f` must be file object type)
- `close(f):` closes the given file (`f` must be file object type)
- `x = read(f):` converts the contents of file `f` into a string and assigns it to `x` (`f` must be a file object type and `x` must be a string type)
- `write(f, x):` writes the string `x` onto at the end of file `f` (`f` must be a file object type and `x` must be a string type)
- `x = y + z:` concatenate strings y and z and store in x (x, y, and z must be string type)

The following two problems refer to this dataflow graph, which uses the above operations:



Question 5. (16 points) Dataflow analysis – files open/closed. We would like to ensure that every file which is opened will eventually be closed. We can use a dataflow framework to analyze which variables refer to open or closed files by defining the following sets for each basic block *b*:

- IN(b) – the set of file variables that are known to be open on entry to block b
- OUT(b) – the set of file variables that are known to be open on exit from block b
- GEN(b) – the set of all file variables that are opened in block b and not later closed in block b before exit from that block
- KILL(b) – the set of all file variables that are closed in block b and not later opened in block b before exit from that block.

The following dataflow equations describe the relationships between these sets:

$IN(b) = \bigcup_{x \in pred(b)} OUT(x)$

$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$

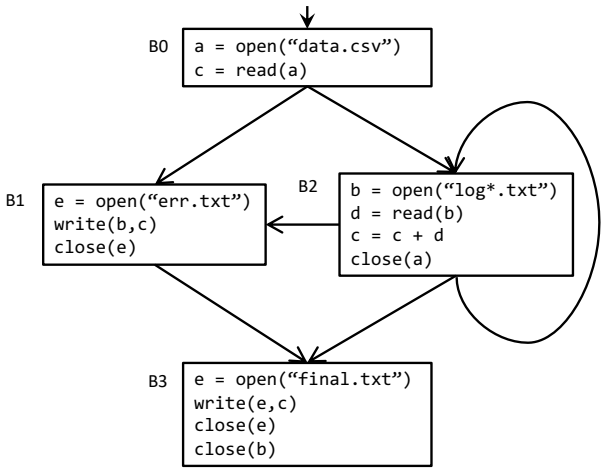
(a) (14 points) Complete the following table using iterative dataflow analysis to identify the open file variables in the IN and OUT sets for each block in the above flow graph. You should first fill in the GEN and KILL sets for each block (which do not depend on other blocks) and then iteratively solve for IN and OUT.

	GEN	KILL	IN	OUT	IN	OUT	IN	OUT
B0								
B1								
B2								
B3								

(b) (2 points) Is it possible that there is an unclosed file at the end of execution? If so, which file(s) might be open? Answer using the control flow diagram and the information about the IN and OUT sets calculated above.

Question 6. (18 points) Dominators and SSA. Here are the basic definitions of dominators and related concepts we have seen previously in class:

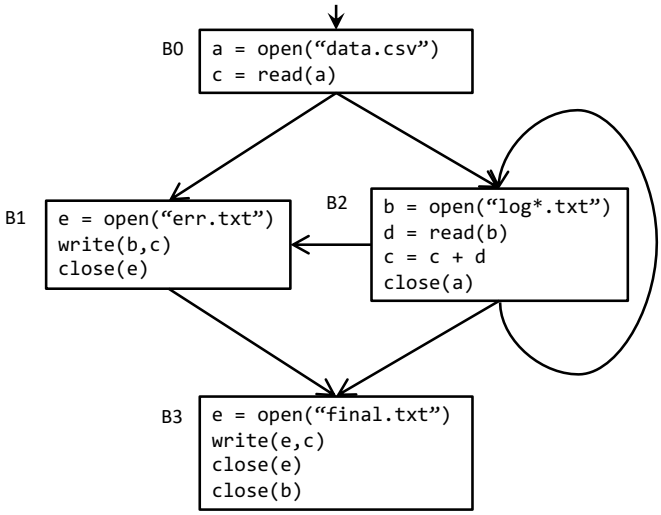
- Every control flow graph has a unique **start node** s.
- Node *x* **dominates** node *y* if every path from s to *y* must go through x.
  - A node *x* dominates itself.
- A node *x* **strictly dominates** node *y* if *x* dominates *y* and $x \neq y$.
- The **dominator set** of a node *x* is the set of nodes *dominated by x*.
  - $|\text{Dom}(x)| \geq 1$
  - (note: sometimes the definition of Dom(x) is given as the set of all nodes that dominate *x*. For SSA it is more convenient to keep track of the set of nodes that *x* dominates.)
- 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*
- 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			
B1			
B2			
B3			

Question 6. (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.



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

(a) (8 points) Draw the interference graph for the temporary variables (t1-t10) in the following code. You should assume that all temporaries are dead at the conclusion of this snippet of code

```
   // code for z = *((v+w) * (w*x)) * (w*x)^2 + y
a. LOAD    t1 <- v       //  t1 = v
b. LOAD    t2 <- w       //  t2 = w
c. ADD     t3 <- t1, t2  //  t3 = v+w
d. MULT    t4 <- t2, x   //  t4 = w*x
e. MULT    t5 <- t3, t4  //  t5 = (v+w) * (w*x)
f. LOAD    t6 <- y       //  t6 = y
g. LOAD    t7 <- MEM[t5] //  t7 = *((v+w) * (w*x))
h. SHIFT   t8 <- t4, $2  //  t8 = (w*x)^2
i. MULT    t9 <- t7, t8  //  t9 = *((v+w) * (w*x)) * (w*x)^2
j. ADD    t10 <- t9, t6  // t10 = *((v+w) * (w*x)) * (w*x)^2 + y
k. STORE    z <- t10     // store z
```

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

Draw a picture of something you are planning to during summer break!
– or –
Draw a picture of something you think one or more of your TAs will do during summer break!

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

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

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