CSE 401/M501 25au Final Exam

December 9, 2025

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

There are 9 questions worth a total of 135 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 attached at the very end of the exam for reference if you need it, and you should **remove** that page with the grammar from the exam and return it for recycling when you're 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	/ 12
2	/ 15
3	/ 15
4	/ 25
5	/ 18
6	/ 18
7	/ 14
8	/ 16
9	/ 2
Total	/ 135

CSE 401/M501 25au Final Exam 12/9/25

A warmup question to get started…

Question 1. (12 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 (unless stated otherwise). Use the following abbreviations for the stages:

scan – scanner
parse – parser
sem – static 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

______ Report division by 0 error in an expression `x/y`

______ Verify that the condition in `while (condition) statement` has type Boolean

______ Report that `#` is not a legal Java operator

______ In a program with multiple threads, report that the program will never deadlock during execution

______ Move loop-invariant code (code that always does the same thing) outside of a loop to improve execution time

______ Report if a class fails to include or inherit all of the methods contained in a Java `interface` that the class implements.

______ Replace a multiplication of a value by a constant power of 2 with a shift operation

______ Report that a comment starting with `/*` is not terminated by `*/` before the end of the input file

______ Ensure that a function result value is stored in register `%rax` when the function returns

______ Report an error if a language keyword is used as a variable name

______ Insert load/store (move) instructions to save a value in memory and later reload it if there are not enough registers available at a program point to hold all live values currently in use

______ Report that in the array reference `a[i]`, the value of `i` exceeds the length of array `a`.

Question 2. (15 points) Declarations, inheritance and scope. Here is a strange Java program consisting of a main class and two classes A and B related by inheritance:

```
class A {
  int x = 11;
  int y = 12;
  void m1(int a) {
    x = a;
    System.out.println("A.m1 x = " + x + ", y = " + y);
    m2(401);
    System.out.println("A.m1 x = " + x + ", y = " + y);
  }
  void m2(int c) {
    y = c;
    System.out.println("A.m2 x = " + x + ", y = " + y);
  }
}

class B extends A {
  int x = 21;
  int z = 22;
  void m2(int b) {
    x = y;
    z = b;
    System.out.println("B.m2 x = " + x + ", y = " + y + ", z = " + z);
  }
}

class Main {
  public static void main(String[] args) {
    A widget = new A();
    widget.m2(333);
    System.out.println("---");
    widget.m1(351);
    System.out.println("---");
    A thing = new B();
    thing.m1(42);
    System.out.println("---");
    thing.m2(17);
  }
}
```

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.)

Question 3. (15 points) A bit of x86-64 coding. Here is a function that checks one of its arguments to see if it is a valid user id, and, depending on the result, returns either a given value or 0. Assume we have the following library function that we can use without having to further declare or implement it.

```
// return 1 if uid is valid, otherwise return 0
int valid(int uid) { ... }
```

For this problem, translate the following function to x86-64 assembly language, and write your answer on the next page.

```
int gettoken(int token, int uid) {
  if (valid(uid)) {
      return token;
    } else {
      return 0;
    }
}
```

On the next page, translate this function into x86-64 assembly language. You should use the standard x86-64 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 in the function prologue code. 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):

- All values, including pointers and `int`s, 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.
- Your x86-64 code must be a straightforward translation of the given code. You may not rewrite or rearrange the code even if it produces equivalent results. However, 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 a MiniJava compiler. In particular, these are C functions, not Java methods. Ordinary C calling conventions without object vtables should be used.
- 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 `gettoken` below. Remember to read and follow the above ground rules carefully, including managing registers properly and using the correct argument registers for function calls. Hint: be careful about what might happen to registers when function `valid` is called.

```
int gettoken(int token, int uid) {
  if (valid(uid)) {
      return token;
    } else {
      return 0;
    }
}
```

Question 4. (25 points) Compiler hacking. One of our big customers has a lot of code that uses the `instanceof` Java operator heavily. They would like us to add this operator to MiniJava before they will consider licensing our compiler.

`instanceof` is a Boolean operator. The expression `obj instanceof T` returns true if `obj` has type `T` and false otherwise. The `instanceof` operator is only defined on reference (object) types; it cannot be applied to primitive types like `int`.

Remember that a value (object) that has a reference type can actually have multiple types, since it not only has a basic type but it also has all of the types of its superclasses. So, for example, suppose we have three related classes: class `SuperThing {...}`, `class Thing extends SuperThing {...}`, and `class SubThing extends Thing {...}`. Then if we create a new object of type `Thing` (i.e., `new Thing()`), the resulting object has type `Thing`, but it also has type `SuperThing`, because any `Thing` object also can be used as if it were a `SuperThing` object.

The `instanceof` operation follows these rules. Suppose we have the following code in a method: `SuperThing st = new Thing();` (which is a legal assignment). Then the expressions `st instanceof Thing` and `st instanceof SuperThing` both evaluate to `true`, but `st instanceof SubThing` is `false`.

Answer the questions below about how we would add this new `instanceof` expression operator 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 this new `instanceof` operator 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 `instanceof` 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 constructor below is used to properly initialize the superclass with this information.)

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




  // constructor - add parameters and method body below


  public Instanceof ( _____________________________________________ )
    super(pos);    // initialize location information in superclass




  }
}
```

(continued on next page)

Question 4. (cont.) (c) (5 points) Complete the CUP specification below to define a production for this new `instanceof` operator, 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 new `Instanceof` operator can be written as an independent grammar rule with separate semantic actions.

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

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

(d) (4 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a program containing this new `instanceof` 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 for this new statement to verify it is used correctly.

(continued on next page)

Question 4. (cont.) (e) (8 points) Describe the x86-64 code shape for the code that would be generated to evaluate the expression *expr* `instanceof Foo` and leave the result (0=false, 1=true) in `%rax`. (Use the specific type `Foo` in your answer.) Your answer should be similar in format to the descriptions we used in class for other language constructs.

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.

Be sure that your code follows the described operation and semantics of the `instanceof` operator precisely, including handling subtypes and supertypes as needed.

When the compiler is producing code for *expr* `instanceof Foo` we can assume that the code generated for *expr* will leave a pointer to the *expr* object in `%rax`. The first 8 bytes of every object contain a vtable pointer. For an object that is created by `new Foo()`, the first 8 bytes of that object in memory will contain the memory address of (pointer to) the `Foo$$` vtable. If the object was created as an instance of some other class, its first 8 bytes will contain the address of some other vtable.

You should assume that the first 8 bytes of the vtable for a class are either 0x0 (null) if the class does not have a superclass, or are a pointer to the superclass vtable if it does. So if class `Bar extends Foo`, then the first 8 bytes in the vtable at `Bar$$` in memory will contain the address of (pointer to) the `Foo$$` vtable.

To load the address of the vtable for class `Foo` into `%rax`, you can use `leaq Foo$$(%rip),%rax`.

Write the codeshape for *expr* `instanceof Foo` below

CSE 401/M501 25au Final Exam 12/9/25

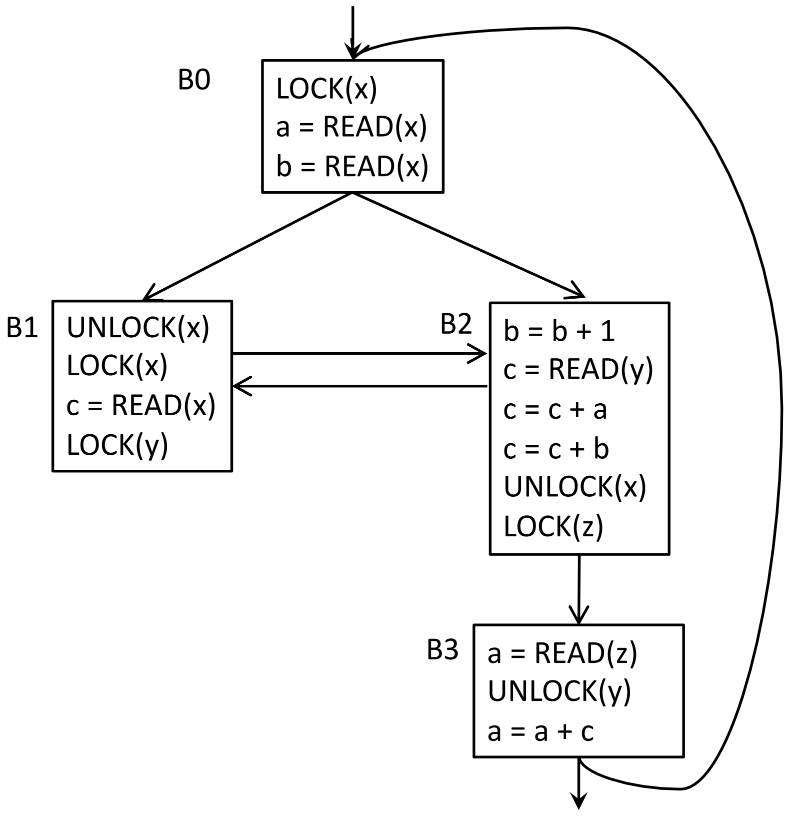
Several of the program analysis techniques pioneered in programming language compilers have been found useful for analyzing similar problems in other areas. For the next question we will use dataflow to analyze code designed to be used with concurrent threads. We want to look at code that acquires and releases locks on streams and reads data from them. The three operations we want to track are:

LOCK(x) – acquire a lock x. If the lock is currently held by another thread, execution blocks until the other thread releases the lock, then this thread acquires the lock and proceeds. Attempting to acquire a lock that is held by another thread only delays this thread, but it is not an error. If lock x is currently held by this thread, then the LOCK(x) operation does nothing.

UNLOCK(x) – release lock x. If lock x is not currently held by this thread (has not been previously locked by this thread), then this operation does nothing. It does not release lock x if another thread has locked it.

READ(x) – return a value from the resource associated with lock x. It is an error if this thread has not previously done a LOCK(x) operation to acquire this lock. The READ operation does not change the status of the lock x.

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



```
B0:
LOCK(x)
a = READ(x)
b = READ(x)

B1:
UNLOCK(x)
LOCK(x)
c = READ(x)
LOCK(y)

B2:
b = b + 1
c = READ(y)
c = c + a
c = c + b
UNLOCK(x)
LOCK(z)

B3:
a = READ(z)
UNLOCK(y)
a = a + c
```

Question 5. (18 points) We would like to use dataflow analysis to discover which locks are held at various points in the program. As usual we will construct a control flow graph from basic blocks, and for each basic block B we will calculate the following information:

IN(B) – the set of locks that might be held by this thread on entry to block B
OUT(B) – the set of locks that might be held by this thread on exit from block B
ACQUIRE(B) – the set of locks that are acquired in block B and not later released in block B
RELEASE(B) – the set of locks that are released in block B and not later acquired in block B

These sets are related by the following equations:

$$IN(B) = \cup_{x \in \mathit{pred}(B)} OUT(x)$$
$$OUT(B) = ACQUIRE(B) \cup (IN(B) - RELEASE(B))$$

(a) (14 points) Complete the following table using iterative dataflow analysis to identify the locks in the IN and OUT sets for each block in the above flow graph. You should first fill in the ACQUIRE and RELEASE sets for each block (which do not depend on other blocks) and the iteratively solve for IN and OUT. You can choose which ever direction (forward or backward) you wish to solve the equations. Only the lock name (x, y, and z) should appear in the table.

	ACQUIRE	RELEASE	IN	OUT	IN	OUT	IN	OUT
B0								
B1								
B2								
B3								

(b) (2 points) Is there any READ operation that attempts to read from input that definitely is not locked? (i.e., is there a path to a READ where we are guaranteed that no LOCK operation has been done on the stream before encounter the READ operation?) If so, which READ statements in which blocks could have this problem? Provide a brief technical justification for your answer.

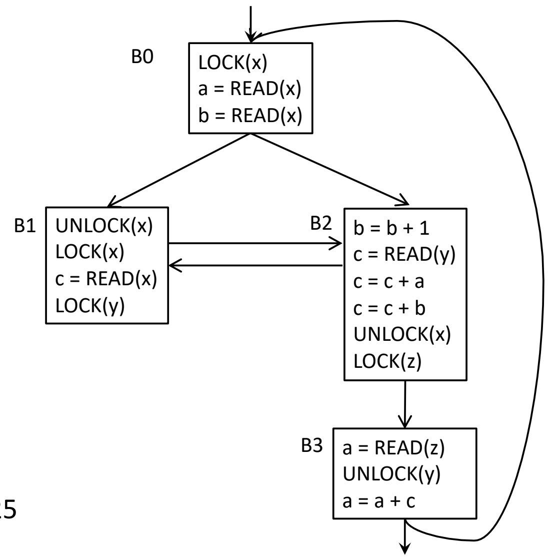
(c) (2 points) Are all locks guaranteed to be released (unlocked) by the end of the program (exit from block B3)? If not, which locks might not be released at that point? Provide a brief technical justification for your answer.

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* ≠ *y*.
- The **dominator set** of a node *x* is the set of nodes *dominated by x*.
  - | Dom(x) | ≥ 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*

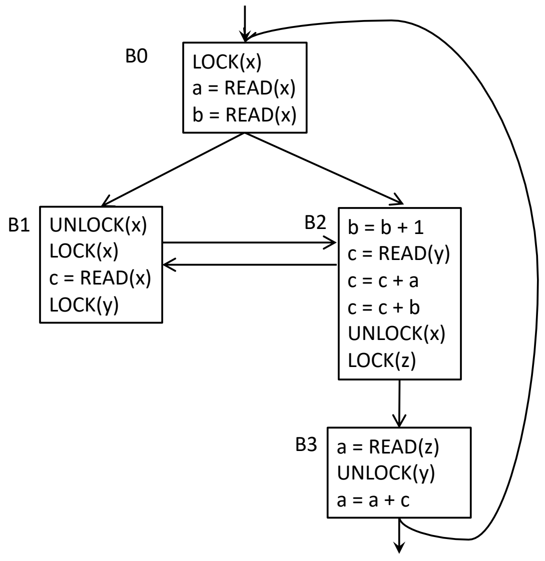
(a) (8 points) Using the same control flow graph from the previous problem, complete the following table. List for each node: the node(s) that it dominates, successor nodes to the dominated nodes, and the nodes that are in its dominance frontier (if any):

Node	Nodes dominated by this node	Successor(s) to nodes dominated by this node	Dominance Frontier of this node
B0			
B1			
B2			
B3			



CSE 401/M501 25au Final Exam 12/9/25

Question 6. (cont.) (b) (10 points) Now redraw the flowgraph using SSA (static single-assignment) form for the ordinary variables (a, b, c), but *not* the lock names (x, y, z). (Assume that the locks retain their original non-SSA names, are not assigned versions, and do not appear in Φ-functions.) You need to insert all other 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 given SSA definitions 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.



The next 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 of 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. Here is a sequence of instructions produced by the code generator.

```
a.  LOAD    t1 <- w          // t1 = w
b.  LOAD    t2 <- x          // t2 = x
c.  ADD     t3 <- t1, t2     // t3 = w + x
d.  SHIFT   t4 <- t3, 2      // t4 = (w+x) * 4
e.  LOAD    t5 <- y          // t5 = y
f.  LOAD    t6 <- z          // t6 = z
g.  MULT    t7 <- t5, t6     // t7 = y * z
h.  ADD     t8 <- t2, t7     // t8 = x + (y*z)
i.  MULT    t9 <- t4, t8     // t9 = ((w+x)*4) * (x+(y*z))
j.  STORE   w <- t9          // w = result
```

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 credit fairly (☺).

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

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

(a) (8 points) Draw the interference graph for the temporary variables (t1-t9) in the code on the previous page (repeated here for convenience). You should assume that the code is executed in the sequence given and not rearranged before assigning registers.

```
a.  LOAD   t1 <- w
b.  LOAD   t2 <- x
c.  ADD    t3 <- t1, t2
d.  SHIFT  t4 <- t3, 2
e.  LOAD   t5 <- y
f.  LOAD   t6 <- z
g.  MULT   t7 <- t5, t6
h.  ADD    t8 <- t2, t7
i.  MULT   t9 <- t4, t8
j.  STORE  w <- t9
```

(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 shown below, 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 correct path that respects the dependencies.

```
a.  LOAD   t1 <- w
b.  LOAD   t2 <- x
c.  ADD    t3 <- t1, t2
d.  SHIFT  t4 <- t3, 2
e.  LOAD   t5 <- y
f.  LOAD   t6 <- z
g.  MULT   t7 <- t5, t6
h.  ADD    t8 <- t2, t7
i.  MULT   t9 <- t4, t8
j.  STORE  w <- t9
```

Operation	Cycles
LOAD	3
STORE	3
ADD	1
MULT	2
SHIFT	1

(continued on next page)

Question 8. (cont) (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 operation (LOAD, ADD, etc.) from the original sequence above and the cycle number on which it begins execution (you do not need to write all of the instruction operands, just the letter and opcode). 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.

Question 9. (2 free points – all answers get the free points)

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

Have a great winter holiday 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.