Question 1. (14 points) Runtime data structures. Suppose we have the following three Java classes:

```java
public class Marsupial {
    int weight;

    public void eat() { ... }
    public void speak() { ... }
}

public class Wombat extends Marsupial {
    boolean happy;
    int age;

    public void play() { ... }
    public void sleep() { ... }
    public void speak() { ... }
}

public class Main {
    public static void main(String[] ignored) {
        Marsupial fred = new Marsupial();
        Marsupial matilda = new Wombat();
        Wombat mate = new Wombat();

        fred.eat();
        matilda.speak();
        mate.play();
    }
}
```

On the next page draw a diagram of the runtime data structures for this program after the declarations in the Main method have been processed, as follows:

(a) Draw pictures showing the variables in the program, the objects they refer to, and how the objects and their data members would be organized and laid out in memory.

(b) Add to your diagram from part (a) any mechanisms that support dynamic method binding as in Java (e.g., vtables). You may assume that this class structure is fixed at compile time and no new classes or methods will be added at runtime. You may also ignore constructors.

(You may remove this page from the exam if that is convenient.)
**Question 1 (cont).** Draw your diagram for question 1 below.

![Diagram of Marsupial and Wombat objects and vtables]

**Notes:** There is some freedom of the order of fields in the objects and vtables. But the first fields in Wombat objects must match the fields in Marsupial objects, and the order of methods in the Marsupial vtable must match the order of the methods in the beginning of the Wombat vtable exactly.
A few short questions on optimizations.

**Question 2.** (6 points) The new optimizing compiler we’ve been working on is designed to move computations outside a loop if they always produce the same value. Given the following original code,

```c
for (i = 0; i < n; i++)
    a[i] = sqrt(x/y);
```

the optimizer rewrites it as follows to avoid recalculating the value inside the loop. (*sqrt* is a library function; it always will return the same result given the same input value and it has no side effects. All values are doubles.)

```c
temp = sqrt(x/y);
for (i = 0; i < n; i++)
    a[i] = temp;
```

Assuming that we are running this code on a single processor with no concurrency, is this optimization always safe and correct? Give a brief argument why or why not.

**No.** The trouble is that *sqrt(x/y)* is always evaluated in the optimized version, but is not evaluated in the original version if \( n \leq 0 \). Evaluation of *sqrt(x/y)* can potentially generate a division by 0 error or can produce an error if \( x/y < 0 \), and that should not happen if the loop is not executed.

The optimization would be safe if we guarded the assignment to *temp* by doing something like this:

```c
if (n > 0) temp = sqrt(x/y);
```

**Question 3.** (6 points) All optimizing compilers perform **dead code elimination**, which eliminates code that is never executed, or which computes values that are never used. Most optimizing compilers perform dead code elimination several times during the optimization passes. Why? Why is there any advantage to doing it more than once?

Optimizations may themselves generate dead code by eliminating the need for various intermediate computations. So even after we run dead code elimination once it may be profitable to run it again later. Further even if we run it again later, it is worth doing it early since it is cheap and reduces the amount of intermediate code that later parts of the compiler need to deal with.
Question 4. (10 points) Suppose we have the code sequence shown on the left. The variables are assumed to be static, global variables.

```java
if ( a + b < 0 ) {
    x = a + b;
} else {
    y = a + b;
}
temp = a + b;
if ( temp < 0 ) {
    x = temp;
} else {
    y = temp;
}
```

An optimization that would reduce the size of the generated code is shown on the right, where the value \( a+b \) is computed once and stored in a temporary variable, then used when it is needed later.

[The question should have included an explicit mention that temp was a compiler temporary or register and not visible to other threads. That is implicit in most of these kinds of examples, but did cause a bit of confusion in a couple of cases. We took that into account when grading the question.]

(a) Is this optimization always legal (i.e., safe and correct) if the code is executed in a single thread with no other concurrent threads? Give a brief argument in support of your answer.

Yes. The result of evaluating \( a+b \) will be the same if it is re-evaluated.

(b) Is this optimization always legal (i.e., safe and correct) if the code is executed in one thread of a multi-threaded program? Give a brief argument in support of your answer.

No. If \( a \) and \( b \) are global then another thread can change their values at any time. So, in the original case, \( a+b \) may have a different value the second time it is evaluated. In the optimized version, \( a+b \) is only evaluated once and changes between the original evaluation and the assignment would be missed.
**Question 5.** (20 points) x86 hacking. Consider the following C function that returns the sum of a sequence of numbers recursively.

```c
/* return the sum k + (k+1) + (k+2) + ... n */
int sum(int k, int n) {
    int ans;
    if (k == n) {
        ans = n;
    } else {
        ans = k + sum(k+1, n);
    }
    return ans;
}
```

(a) (6 points) In the space below draw a picture of the stack frame for function `sum` right before executing the `return` statement at the end of the function. Your picture should show where the parameters and variables are located, as well as any additional items that are part of the stack frame, such as the return address. You should also draw labeled arrows showing where in the stack frame the registers ebp (frame pointer) and esp (stack pointer) point, and indicate the numeric offset of each parameter and local variable from the frame pointer ebp.

```
<table>
<thead>
<tr>
<th>High addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>(space belonging to calling function)</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>n (+12)</td>
</tr>
<tr>
<td>k (+8)</td>
</tr>
<tr>
<td>return address</td>
</tr>
<tr>
<td>ebp</td>
</tr>
<tr>
<td>esp</td>
</tr>
<tr>
<td>old ebp</td>
</tr>
<tr>
<td>ans (-4)</td>
</tr>
</tbody>
</table>
```

(continued next page)
**Question 5. (cont) (b) (14 points)** Translate the `sum` function into x86 assembly language. Your code does not need to look like the code generated by your compiler – any clean x86 code will do. However, your code must conform to the standard x86 C language calling conventions. Further, your code must include all of the statements in the original function, including the assignments to the local variable `ans` and the recursive function call. You may use either the Intel or GNU assembler syntax for your code – just be sure to pick one and not mix them. Note: The standard conventions require that registers ebx, esi, edi, ebp, and esp must be saved and restored if they are used in the body of a function. Code

```c
/* return the sum k + (k+1) + (k+2) + ... + n */
int sum(int k, int n) {
    int ans;
    if (k == n) { ans = n; } else { ans = k + sum(k+1, n); }
    return ans;
}
```

`; answer using intel syntax

```assembly
sum:  push ebp         ; prologue
    mov ebp,esp
    sub esp,4   ; frame with space for ans
    mov eax,[ebp+12] ; eax = n
    mov edx,[ebp+8] ; edx = k
    cmp eax,edx
    jne else   ; jump if n != k
    mov [ebp-4],eax ; ans = n
    jmp exit   ; return
else: mov eax,[ebp+12] ; call sum(k+1,n)
    push eax    ; push n
    mov eax,[ebp+8]
    inc eax
    push eax    ; push k+1
    call sum    ; recursive call – result in eax
    add esp,8   ; pop arguments
    add eax,[ebp+8] ; add k to result
    mov [ebp-4],eax ; store ans
exit: mov eax,[ebp-4] ; return ans in eax
    mov esp,ebp ; standard return
    pop ebp
    ret
```

There are obviously many possible solutions to this question. The above is fairly straightforward code, and definitely not the most compact possible.

Although the instructions said to include all of the original code in the solution, many people used eax to hold variable ans without storing it in the stack frame. We let that go when grading.
**Question 6.** (32 points) Compiler hacking: the question of many parts.

Most programming languages have loops that either test the loop condition before the loop body executes (while, for) or after (do-while). But often it would be very convenient to have a loop with a test in the middle. An example is when reading an input file that is terminated with an end marker in the data. This is easy to express if we have a loop that looks like this (pseudo-code, not necessarily real MiniJava):

```
loop
  read(value)
  while (value != eof-marker)
    process(value)
repeat
```

We’d like to add such a loop to our MiniJava compiler. The syntax of the loop statement is:

```
loop statement1 while (condition) statement2 repeat
```

(Aside: We ignore the question of whether there should be a semicolon following repeat – for the sake of this question, assume there is no semicolon.)

The meaning is as expected from the example. First, `statement1` is executed. Then the `condition` is evaluated. If it is false, execution of the `loop` statement terminates, and control continues with whatever follows the keyword `repeat`. If `condition` is true, `statement2` is executed, then we loop back to the top and execute `statement1` again to begin the next iteration.

Answer the rest of this question on the next few pages. You can remove this page and the following one, which contains the MiniJava grammar, and use those for reference as you work on the question.

Also, for reference, remember that the AST package in MiniJava contains the following key classes.

<table>
<thead>
<tr>
<th>abstract ASTNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract Exp extends ASTNode</td>
</tr>
<tr>
<td>abstract Statement extends ASTNode</td>
</tr>
</tbody>
</table>

Specific classes in the AST have constructors like `While(Exp cond, Statement body, int line_nbr)`, and contain suitable instance variables to hold references to appropriate subtrees in the AST.

(continued next page)
Question 6 (cont.) (a) (3 points) What new tokens would need to be added to the scanner and parser of our Minijava compiler to add the new loop statement to the original Minijava grammar? Just list the tokens; you don’t need to give a JFlex or CUP specification for them.

```plaintext
LOOP   REPEAT
```

(b) (7 points) Complete the following class to define a new AST node class for the loop statement. 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.

```java
public class Loop extends Statement {
    // add instance variables below
    Statement stmt1, stmt2;
    Exp cond;

    // constructor - add parameters and body
    public Loop( Exp e, Statement s1, Statement s2, int line_nbr ) {
        super(line_nbr);
        cond = e;
        stmt1 = s1;
        stmt2 = s2;
    }
}
```
Question 6 (cont.) (c) (7 points) Complete the CUP specification below to define a new production for the loop statement and the associated semantic action(s) needed to parse a loop and insert an appropriate Loop node (as defined in part (b)) into the AST. We have added the necessary additional code to the parser rule for Statement as shown below.

```
Statement ::= ...
    | LoopStatement:s (: RESULT = s; :)
    ...
;
LoopStatement ::= LOOP Statement:s1 WHILE LPAREN Exp:e RPAREN Statement:s2 REPEAT
               {: RESULT = new Loop(e,s1,s2,s1left); :}
```

(d) (5 points) Describe the checks that would be needed in the semantics/type-checking part of the compiler to verify that a loop statement was legal. You do not need to give code for a visitor method or anything like that – just describe what rules (if any) need to be checked.

Verify that the condition expression has type Boolean
Question 6 (cont.) (e) (10 points) Describe the code that would be generated for the new loop. You need to show the instructions, labels, and any other assembly language code that need to be generated for the loop statement itself, and show where the generated code for statement1, statement2, and the condition would appear in the code sequence for loop. In writing your code, you should assume that the compiled code for condition will leave the value 1 in eax during execution if the condition evaluates to true, and will leave a 0 in eax if it evaluates to false. Use that value to control whether the loop continues or not; don’t use any fancier branching scheme.

Many people included a great deal of detail here, but answers were ok as long as they included the labels, branches, and other code for the loop itself and showed where the nested statements and expressions would be included.

    loop_label:
    
        statement1 code

        condition code

        cmp     eax,0

        je      loop_exit

        statement2 code

        jmp     loop_label

    loop_exit:
Question 7. (12 points) A little coloring. Considering the following code fragment:

```c
a = read();
b = read();
c = a+b;
if (c > 0) {
    d = b+c;
} else {
    d = c+1;
}
print(d);
```

(a) Draw the control flow graph for the code, keeping the diagram to the left side of the paper.

(b) To the right of the control flow graph, neatly show the live ranges of the variables.

(c) Below, draw the interference graph for the variables. Use the left side of the paper.

(d) To the right of the interference graph, indicate which groups of variables can occupy the same register, based on the information in the interference graph. You do not need to go through the steps of the graph coloring algorithm explicitly, although it 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.