CSE 401 Final Exam

March 17, 2010

Name ___________________________________

You may have one sheet of handwritten notes plus the handwritten notes from the midterm. Otherwise, the 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	/ 14
2	/ 6
3	/ 6
4	/ 10
5	/ 20
6	/ 32
7	/ 12
Total	/ 100

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

```
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.

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,

```
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`.)

```
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.

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?

Question 4. (10 points) Suppose we have the code sequence shown on the left. The variables are assumed to be static, global variables.

```
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.

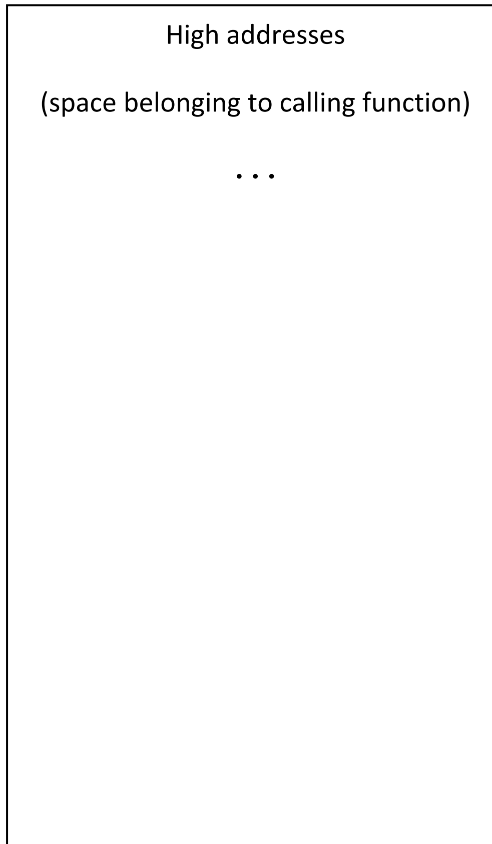
(a) Is this optimization always legal (i.e., safe and correct) if the code is executed in a single thread with no other concurrent threads? Give a brief argument in support of your answer.

(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.

Question 5. (20 points) x86 hacking. Consider the following C function that returns the sum of a sequence of numbers recursively.

```
/* 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`.



(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 repeated for reference, but reformatted to save space:

```
/* 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;
}
```

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.

abstract ASTNode
abstract Exp extends ASTNode
abstract Statement extends ASTNode

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.

(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.

```
public class Loop extends Statement {
   // add instance variables below
   [                                        ]

   // constructor – add parameters and body

   public Loop( ______________________________ ) {
      [ super(line_nbr);                      ]
   }
}
```

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 ::=
```

(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.

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.

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

```
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.