1. (20 pts) Consider the following program in Cool, representing an over-engineered implementation which calculates the factorial of 3 using an operator class and a reduce() method:

```cool
1 class BinOp {
2     operate(a: Int, b: Int): Int {
3         a + b
4     };
5     optype(): String {
6         "BinOp"
7     };
8 };
9 class SumOp inherits BinOp {
10     optype(): String {
11         "SumOp"
12 };
13 }; 
14 class MulOp inherits BinOp {
15     operate(a: Int, b: Int): Int {
16         a * b
17     };
18     optype(): String {
19         "MulOp"
20 };
21 }; 
22 class IntList {
23     head: Int;
24     tail: IntList;
25     empty_tail: IntList; -- Do not assign.
26     tail_is_empty(): Bool {
27         tail = empty_tail
28     };
29     get_head(): Int { head };
30     set_head(n: Int): Int {
31         head <- n
32     };
33     get_tail(): IntList { tail };
34     set_tail(t: IntList): IntList {
35         tail <- t
36     };
37     generate(n: Int): IntList {
38         let l: IntList <- New IntList in {
39             -- Point A
40             l.set_head(n);
41             if (n = 1) then
42         }
```
class Main {
    reduce(result: Int, op: BinOp, l: IntList): Int {{
        result <- op.operate(result, l.get_head());
        if (l.tail_is_empty() = true) then
            -- Point B
            result
        else
            reduce(result, op, l.get_tail())
        fi;
    }};
    main(): Object {
            l <- l.generate(3);
            io.out_int(self.reduce(1, op, l));
        }
    }
}

The following is an abstracted representation of a memory layout of the program generated by a hypothetical Cool compiler for the above code (note that this might or might not correspond to the layout generated by your compiler or the reference coolc):
In the above, \texttt{maddr}_i represents the memory address at which the corresponding method’s code or dispatch table starts. You should assume that the above layout is contiguous in memory.

(a) (4 pts) Assume the MIPS assembly code to be stored starting at address \texttt{maddr}_{12} and ending immediately before \texttt{maddr}_{13} (i.e. not including the instruction starting at \texttt{maddr}_{13}) was generated using the code generation process from lecture. In particular, assume that the caller is responsible for saving and restoring the frame pointer. In addition, assume that the address to the self object is stored on the stack along with the other parameters. How many instructions using the frame pointer register ($fp$) will be present within such code? Why?

(b) (4 pts) The following is a representation of the dispatch table for class Main:

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reduce</td>
<td>\texttt{maddr}_{12}</td>
</tr>
<tr>
<td>1</td>
<td>main</td>
<td>\texttt{maddr}_{13}</td>
</tr>
</tbody>
</table>
Provide equivalent representations for the dispatch tables of BinOp, SumOp, MulOp, and IntList.

(c) (4 pts) Consider the state of the program at runtime when reaching (for the first time) the beginning of the line marked with the comment “Point A”. Give the object layout (as per Lecture 12) of every object currently on the heap which is of a class defined by the program (i.e. ignoring Cool base classes such as IO or Int). For attributes, you can directly represent Int values by integers and an unassigned pointer by void. However, note that in a real Cool program, Int is an object and would have its own object layout, omitted here for simplicity. Finally, you can assume class tags are numbers from 1 to 5 given in the same order as the one in which classes appear in the layout above, and that attributes are laid out in the same order as the class definition.

(d) (8 pts) The following table represents an abstract view of the layout of the stack at runtime when reaching (for the first time) the beginning of the line marked with the comment “Point A”.

<table>
<thead>
<tr>
<th>Address</th>
<th>Method</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr19</td>
<td>Main.main</td>
<td>self</td>
<td>arg0</td>
</tr>
<tr>
<td>maddr19 +4</td>
<td>Main.main</td>
<td>...</td>
<td>Return</td>
</tr>
<tr>
<td>maddr19 +8</td>
<td>Main.main</td>
<td>op</td>
<td>local</td>
</tr>
<tr>
<td>maddr19 +12</td>
<td>Main.main</td>
<td>1</td>
<td>local</td>
</tr>
<tr>
<td>maddr19 +16</td>
<td>Main.main</td>
<td>io</td>
<td>local</td>
</tr>
<tr>
<td>maddr19 +20</td>
<td>IntList.generate</td>
<td>maddr19</td>
<td>FP</td>
</tr>
<tr>
<td>maddr19 +24</td>
<td>IntList.generate</td>
<td>self</td>
<td>arg0</td>
</tr>
<tr>
<td>maddr19 +28</td>
<td>IntList.generate</td>
<td>3</td>
<td>arg1</td>
</tr>
<tr>
<td>maddr19 +32</td>
<td>IntList.generate</td>
<td>maddr13 + δ</td>
<td>Return</td>
</tr>
<tr>
<td>maddr19 +36</td>
<td>IntList.generate</td>
<td>1</td>
<td>local</td>
</tr>
</tbody>
</table>

Give a similar view of the stack at runtime when reaching (for the first time) the beginning of the line marked with the comment “Point B”.

Note that we are assuming there are no stack frames above Main.main(...). This doesn’t necessarily match a real implementation of the Cool runtime system, where main must return control to the OS or the Cool runtime on exit. For the purposes of this exercise, feel free to ignore this issue. Also, since you don’t have the generated code for every method above, you cannot directly calculate the return address to be stored on the stack. You should however give it as maddr_i + δ, denoting an unknown address between maddr_i and maddr_i+1. This notation is used in the example above. For locals, you should use the variable name, but remember that in practice it is the heap address that gets stored in memory for objects.

2. (4 pts) Consider the following arithmetic expression: \( 7 * ((6 + 4) / 2 - 3 + (5 - 4) * 3) + 1 \).

(a) (2 pts) You are given MIPS code that evaluates this expression using a stack machine with a single accumulator register (similar to the method given in class Lecture 12). This code is wholly unoptimized and does not perform transformations such as arithmetic simplification or constant folding. How many times in total will this code push a value
to or pop a value from the stack (give a separate count for the number of pushes and the number of pops)?

(b) (2 pts) Now suppose that you have access to two registers r1, r2 in addition to the stack pointer. Consider the code generated using the revised process described in lecture 12 starting on slide 30, with r1 as an accumulator and r2 storing temporaries. How many loads and stores are now required?

3. (16 pts) Suppose you want to add a for-loop construct to Cool, having the following syntax:

   for id : Int ← e1 to e2 do e3 rof

The above for-loop expression is evaluated as follows: expressions $e_1$ and $e_2$ are evaluated only once, then the body of the loop ($e_3$) is executed once for every integer in the range $[e_1, e_2]$ (inclusive) in order. Similar to the while loop, the for-loop returns void. Also assume that $id$ is not in the environment and hence cannot be modified in the loop body.

(a) (10 pts) Give the operational semantics for the for-loop construct above.

(b) (6 pts) Give the code generation function $cgen$($for$ id : Int ← e1 to e2 do e3 rof) for this construct. Use the code generation conventions from the lecture. The result of $cgen(...)$ must be MIPS code following the stack-machine with one accumulator model.

4. (8 pts) Consider the following basic block, in which all variables are integers.

   1. $a := f * f$
   2. $b := 1 * f + 0$
   3. $c := 7 * 7$
   4. $d := b + c$
   5. $e := f * f$
   6. $g := a + d$
   7. $x := e + d$
   8. $k := b * f$
   9. $h := g * x$
  10. $y := h + k$

Assume that the only variables that are live at the exit of this block are $x$ and $y$, while $f$ is given as an input. In order, apply the following optimizations to this basic block. Show the result of each transformation. For each optimization, you must continue to apply it until no further applications of that transformation are possible, before writing out the result and moving on to the next.

(a) Algebraic simplification

(b) Copy propagation

(c) Common sub-expression elimination

(d) Constant folding

(e) Copy propagation

(f) Dead code elimination
When you have completed the last of the above transformations, the resulting program will still not be optimal. What optimization(s), in what order, can you apply to optimize the result further?

5. (12 pts) Consider the following assembly-like pseudo-code, using 10 temporaries (abstract registers) t0 to t9:

```
1  t1 = 2 * t0
2  t2 = 10 + t0
3  if t1 < t2
4     t3 = t1 + t2
5     t4 = t1 + t3
6   else
7     t4 = t1 + t2
8  fi
9  t5 = t4 + t1
10  t6 = t4 - t1
11  t7 = t4 * t4
12  t8 = t5
13  t8 = t6 + t8
14  t8 = t7 + t8
15  if t6 < t8
16     t9 = t6
17   else
18     t9 = t8
19  fi
```

(a) At each program point, list the temporaries that are live. Note that t0 is the only input temporary for this code and t9 will be the only live value on exit.

(b) Provide a lower bound on the number of registers required by the program.

(c) Draw the interference graph between temporaries in the above program as described in class.

(d) Using the algorithm described in class, provide a coloring of the above graph. The number of colors used should be your lower bound in part (b). Provide the final k-colored graph (you may use the tikz package to typeset it or simply embed an image), along with the order in which the algorithm colors the nodes.

(e) Based on your coloring, write down a mapping from temporaries to registers (labelled \( r_1, r_2, \) etc.).