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

```cool
class BinOp {
    optype(): String {
        "BinOp"
    }
    operate(a: Int, b: Int): Int {
        a + b
    }
}

class SumOp inherits BinOp {
    optype(): String {
        "SumOp"
    }
}

class MulOp inherits BinOp {
    optype(): String {
        "MulOp"
    }
    operate(a: Int, b: Int): Int {
        a * b
    }
}

class IntList {
    head: Int;
    tail: IntList;
    empty_tail: IntList; -- Do not assign.
    tail_is_empty(): Bool {
        tail = empty_tail
    }
    get_head(): Int { head }
    set_head(n: Int): Int {
        head <- n
    }
    get_tail(): IntList { tail }
    set_tail(t: IntList): IntList {
        tail <- t
    }
    generate(n: Int): IntList {
```
let l: IntList <- New IntList in {
  l.set_head(n); -- Point A
  if (n = 1) then
    l.set_tail(empty_tail)
  else
    l.set_tail(generate(n-1))
  fi;
  l;
};

class Main {
  reduce (result : Int , op: BinOp , l: IntList): Int {
    result <- op.operate (result , l.get_head());
    if (l.tail_is_empty() = true) then
      result -- Point B
    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):

<table>
<thead>
<tr>
<th>Code segment:</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr₁: cgenₘ(BinOp.optype)</td>
</tr>
<tr>
<td>maddr₂: cgenₘ(BinOp.operate)</td>
</tr>
<tr>
<td>maddr₃: cgenₘ(SumOp.optype)</td>
</tr>
<tr>
<td>maddr₄: cgenₘ(MulOp.operate)</td>
</tr>
<tr>
<td>maddr₅: cgenₘ(MulOp.optype)</td>
</tr>
<tr>
<td>maddr₆: cgenₘ(IntList.tail_is_empty)</td>
</tr>
<tr>
<td>maddr₇: cgenₘ(IntList.get_head)</td>
</tr>
<tr>
<td>maddr₈: cgenₘ(IntList.set_head)</td>
</tr>
<tr>
<td>maddr₉: cgenₘ(IntList.get_tail)</td>
</tr>
<tr>
<td>maddr₁₀: cgenₘ(IntList.set_tail)</td>
</tr>
<tr>
<td>maddr₁₁: cgenₘ(IntList.generate)</td>
</tr>
<tr>
<td>maddr₁₂: cgenₘ(Main.reduce)</td>
</tr>
<tr>
<td>maddr₁₃: cgenₘ(Main.main)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dispatch tables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr₁₄: DT[BinOp]</td>
</tr>
<tr>
<td>maddr₁₅: DT[SumOp]</td>
</tr>
<tr>
<td>maddr₁₆: DT[MulOp]</td>
</tr>
<tr>
<td>maddr₁₇: DT[IntList]</td>
</tr>
<tr>
<td>maddr₁₈: DT[Main]</td>
</tr>
</tbody>
</table>

Stack (maddr₁₉)

↑

Heap

In the above, maddr₁ 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) 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>maddr_{12}\</td>
</tr>
<tr>
<td>1</td>
<td>main</td>
<td>maddr_{13}</td>
</tr>
</tbody>
</table>

Provide equivalent representations for the dispatch tables of BinOp, SumOp and IntList.

**BinOp:**

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>optype</td>
<td>maddr_{1}</td>
</tr>
<tr>
<td>1</td>
<td>operate</td>
<td>maddr_{2}</td>
</tr>
</tbody>
</table>

**SumOp:**

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>optype</td>
<td>maddr_{3}</td>
</tr>
<tr>
<td>1</td>
<td>operate</td>
<td>maddr_{2}</td>
</tr>
</tbody>
</table>

**MulOp:**

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>optype</td>
<td>maddr_{4}</td>
</tr>
<tr>
<td>1</td>
<td>operate</td>
<td>maddr_{5}</td>
</tr>
</tbody>
</table>

**IntList:**

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tail_is_empty</td>
<td>maddr_{6}</td>
</tr>
<tr>
<td>1</td>
<td>get_head</td>
<td>maddr_{7}</td>
</tr>
<tr>
<td>2</td>
<td>set_head</td>
<td>maddr_{8}</td>
</tr>
<tr>
<td>3</td>
<td>get_tail</td>
<td>maddr_{9}</td>
</tr>
<tr>
<td>4</td>
<td>set_tail</td>
<td>maddr_{10}</td>
</tr>
<tr>
<td>5</td>
<td>generate</td>
<td>maddr_{11}</td>
</tr>
</tbody>
</table>
(b) 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.

**Answer:**

<table>
<thead>
<tr>
<th>Class</th>
<th>Tag</th>
<th>Maddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>5</td>
<td>maddr18</td>
</tr>
<tr>
<td>MulOp</td>
<td>3</td>
<td>maddr16</td>
</tr>
<tr>
<td>IntList (in Main.main)</td>
<td>4</td>
<td>maddr17</td>
</tr>
<tr>
<td>IntList (in IntList.generate)</td>
<td>4</td>
<td>maddr17</td>
</tr>
</tbody>
</table>
(c) 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>l</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 - 4</td>
<td>FP</td>
</tr>
<tr>
<td>maddr19 - 24</td>
<td>IntList.generate</td>
<td>3</td>
<td>arg1</td>
</tr>
<tr>
<td>maddr19 - 28</td>
<td>IntList.generate</td>
<td>self</td>
<td>arg0</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>

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 maddri + δ, denoting an unknown address between maddri and maddri+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.

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”.

6
<table>
<thead>
<tr>
<th>Address</th>
<th>Method</th>
<th>Contents</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr_19</td>
<td>Main.main</td>
<td>self</td>
<td>arg_0</td>
</tr>
<tr>
<td>maddr_19 - 4</td>
<td>Main.main</td>
<td>...</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_19 - 8</td>
<td>Main.main</td>
<td>op</td>
<td>local</td>
</tr>
<tr>
<td>maddr_19 - 12</td>
<td>Main.main</td>
<td>1</td>
<td>local</td>
</tr>
<tr>
<td>maddr_19 - 16</td>
<td>Main.main</td>
<td>io</td>
<td>local</td>
</tr>
<tr>
<td>maddr_19 - 20</td>
<td>Main.reduce</td>
<td>maddr_19 - 4</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_19 - 24</td>
<td>Main.reduce</td>
<td>ptr to [3,2,1]</td>
<td>arg_3</td>
</tr>
<tr>
<td>maddr_19 - 28</td>
<td>Main.reduce</td>
<td>ptr to MulOp</td>
<td>arg_2</td>
</tr>
<tr>
<td>maddr_19 - 32</td>
<td>Main.reduce</td>
<td>3</td>
<td>arg_1</td>
</tr>
<tr>
<td>maddr_19 - 36</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_0</td>
</tr>
<tr>
<td>maddr_19 - 40</td>
<td>Main.reduce</td>
<td>maddr_13 + δ_1</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_19 - 44</td>
<td>Main.reduce</td>
<td>maddr_19 - 40</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_19 - 48</td>
<td>Main.reduce</td>
<td>ptr to [2,1]</td>
<td>arg_3</td>
</tr>
<tr>
<td>maddr_19 - 52</td>
<td>Main.reduce</td>
<td>ptr to MulOp</td>
<td>arg_2</td>
</tr>
<tr>
<td>maddr_19 - 56</td>
<td>Main.reduce</td>
<td>6</td>
<td>arg_1</td>
</tr>
<tr>
<td>maddr_19 - 60</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_0</td>
</tr>
<tr>
<td>maddr_19 - 64</td>
<td>Main.reduce</td>
<td>maddr_12 + δ_2</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_19 - 68</td>
<td>Main.reduce</td>
<td>maddr_19 - 64</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_19 - 72</td>
<td>Main.reduce</td>
<td>ptr to [1]</td>
<td>arg_3</td>
</tr>
<tr>
<td>maddr_19 - 76</td>
<td>Main.reduce</td>
<td>ptr to MulOp</td>
<td>arg_2</td>
</tr>
<tr>
<td>maddr_19 - 80</td>
<td>Main.reduce</td>
<td>6</td>
<td>arg_1</td>
</tr>
<tr>
<td>maddr_19 - 84</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_0</td>
</tr>
<tr>
<td>maddr_19 - 88</td>
<td>Main.reduce</td>
<td>maddr_12 + δ_2</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_19 - 92</td>
<td>Main.reduce</td>
<td>result</td>
<td>local</td>
</tr>
</tbody>
</table>
2. Consider the following assembly language used to program a stack machine (r, r1, and r2 denote arbitrary registers):

- push r: copies the value of r and pushes it onto the stack.
- top r: copies the value at the top of the stack into r. This command does not modify the stack.
- pop: discards the value at the top of the stack.
- swap: swaps the value at top of the stack with the value right beneath it. E.g. if the stack was <$ ... 5 2> swap would change the stack to be <$ ... 2 5>.
- r1 += r2: adds r1 and r2 and saves the result in r1. r1 may be the same as r2.
- r1 -= r2: subtracts r2 from r1 and saves the result in r1. r1 may be the same as r2.
- clamp r: sets r to 0 if r is negative otherwise leaves r unchanged.
- jump r: jumps to the address in r and resumes execution.
- ite r1 r2 r3: if r1 is not equal to zero then jumps to the address in r2 else jumps to the address in r3.
- loadconst r int: loads a constant int into r.
- loadlabel r label: loads the address of a labeled code segment into r.

Provide a code generation function for each of these instructions, except loadlabel, targeting MIPS. Assume that registers used in the stack language are valid MIPS registers. Use $sp to hold a pointer to the top of the stack and a single temporary register $at which is guaranteed to not appear in the stack language.
cgen(push r) =
    sw r 0($sp)
    addiu $sp $sp -4

cgen(top r) =
    lw r 4($sp)

cgen(pop) =
    addiu $sp $sp 4

cgen(swap) =
    lw $at 4($sp) // get the top of the stack
    sw $at 0($sp) // duplicate it
    lw $at 8($sp) // get the item below it
    sw $at 4($sp) // store it on top of the stack
    lw $at 0($sp) // get the duplicated item
    sw $at 8($sp) // store it in the second slot

cgen(r1 += r2) =
    addu r1 r1 r2

cgen(r1 -= r2) =
    subu r1 r1 r2

cgen(clamp r) =
    bltz r neg
    j done
    neg:
    li r 0
    done:

cgen(jump r) =
    jr r

cgen(ite r1 r2 r3) =
    li $at 0
    beq $at r1 false_branch
    jr r2
    false_branch:
    jr r3

cgen(loadconst r int) =
    li r int
3. Suppose you want to add a for-loop construct to Cool, having the following syntax:

\[
\text{expr ::= } \ldots \quad | \quad \text{for} \text{ expr to expr do expr rof }
\]

(1)

\[
O, M, C \vdash e_1 : \text{Int} \quad O, M, C \vdash e_2 : \text{Int} \quad O, M, C \vdash e_3 \colon T
\]

\[
\frac{\text{For}}{O, M, C \vdash \text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof } : \text{Int}}
\]

The above for-loop expression is evaluated as follows: expressions \( e_1 \) and \( e_2 \) are evaluated exactly one time, then the body of the loop \( e_3 \) is executed once for each integer in the range \([e_1, e_2)\) (exclusive). The for-loop returns the number of times it was executed. 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 each integer in the range \([e_1, e_2)\) (exclusive). The for-loop returns the number of times it was executed.

(a) Give the operational semantics for the for-loop construct above as a judgement.

\[
\frac{\text{ForTrue}}{so, S_1, E \vdash e_1 \mapsto \text{Int}(i_1), S_2}
\]

\[
\frac{so, S_2, E \vdash e_2 \mapsto \text{Int}(i_2), S_3}{i_1 < i_2}
\]

\[
\frac{so, S_3, E \vdash e_3 \mapsto v, S_4}{so, S_4, E \vdash \text{for } \text{Int}(i_1 + 1) \text{ to } \text{Int}(i_2) \text{ do } e_3 \text{ rof } \mapsto \text{Int}(i_3), S_5}
\]

\[
\frac{so, S_1, E \vdash e_1 \mapsto \text{Int}(i_1), S_2}{so, S_2, E \vdash e_2 \mapsto \text{Int}(i_2), S_3}
\]

\[
\frac{i_1 \geq i_2}{so, S_1, E \vdash \text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof } \mapsto \text{Int}(0), S_3}
\]
(b) Give the code generation function for the for-loop construct targeting stack machine
describe in question 2. Assume that the code generation for other expressions will leave
their results on the top of the stack with no other changes. The code generation for
the for-loop should similarly leave its result on the top of the stack without otherwise
changing the stack. You may use registers t0-t7 but you must assume they will be
mutated by the code code generated for other expressions.

```plaintext
cgen(for e1 to e2 do e3 rof) =
cgen(e1)
cgen(e2)
top t2 // get the results of e2
pop
top t1 // get the results of e1
pop
t2 -= t1 // calculate the number of iterations
clamp t2 // set to 0 if negative
push t2 // This will eventually be the return
push t2 // We will use this copy as a counter
cond:
top t1 // get the counter
pop
loadlabel t2 loop
loadlabel t3 done
ite t1 t2 t3
loop:
loadconst t2 1
t1 -= t2 // decrement the counter
push t1 // push back onto the stack
cgen(e3) // evaluate e3
pop // remove its evaluation from the stack
loadlabel t2 cond
jump t2
done:
```
4. Consider the following basic block, in which all variables are integers.

```
1     x := 0 * 5
2     y := a + b
3     z := x * x
4     c := y * x
5     x := x + 4
6     e := c - x
7     x := e * x
8     f := a + b
9     y := y + f
```

(a) Assume that the only variables that are live at the exit of this block are x and y, while a and b are given as inputs. 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 (if any were), before writing out the result and moving on to the next.

i. Algebraic simplification
ii. Common sub-expression elimination
iii. Copy propagation / Constant propagation
iv. Algebraic simplification
v. Dead code elimination

solution:

i.
```
1     x := 0
2     y := a + b
3     z := x * x
4     c := y * x
5     x := x + 4
6     e := c - x
7     x := e * x
8     f := a + b
9     y := y + f
```

ii.
```
1     x := 0
2     y := a + b
3     z := x * x
4     c := y * x
5     x := x + 4
6     e := c - x
7     x := e * x
8     f := y
9     y := y + f
```

iii.
```
1     x := 0
2     y := a + b
3     z := 0 * 0
4     c := y * 0
```
5 \quad x := 0 + 4
6 \quad e := c - x
7 \quad x := e * x
8 \quad f := y
9 \quad y := y + y

text iv
1 \quad x := 0
2 \quad y := a + b
3 \quad z := 0
4 \quad c := 0
5 \quad x := 4
6 \quad e := c - x
7 \quad x := e * x
8 \quad f := y
9 \quad y := y \ll 1

text v
1 \quad y := a + b
2 \quad c := 0
3 \quad x := 4
4 \quad e := c - x
5 \quad x := e * x
6 \quad y := y \ll 1

(b) The resulting program is still not optimal. What optimizations, in what order, can you apply to fully optimize the result? Show the maximally optimized codes (with least number of instructions).

solution: You can apply copy propagation, algebraic simplification, and dead code elimination to get the following.

1 \quad y := a + b
2 \quad x := -16
3 \quad y := y \ll 1
5. Consider the following assembly-like pseudo-code, using 6 temporaries (abstract registers) a to f:

1. a := b + d
2. b := a + d
3. d := a - b
4. c := d + d
5. if c > 100:
6. c := c + d
7. else:
8. d := 1
9. e := d - c
10. f := e - c

(a) At each program point, list the variables that are live. Note that b and d are inputs for the given code and f is a live value on exit.

solution:

1. \{b, d\} a := b + d \{a, d\}
2. b := a + d \{a, b\}
3. d := a - b \{d\}
4. c := d + d
5. if c > 100:
6. \{c, d\} c := c + d \{c, d\}
7. else:
8. \{c\} d := 1 \{c, d\}
9. e := d - c \{c, e\}
10. f := e - c \{f\}

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

(c) Provide a lower bound on the number of registers required by the program induced from the interference graph. Can you explain why?

Solution:

3. It is mainly because there is a triangle in the graph.

(d) Using the algorithm described in class, provide a coloring of the graph in part (b). The number of colors used should be your lower bound in part (c). 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 (labeled r1, r2, etc.).

1  a: r1
2  b: r2
3  c: r2
4  d: r3
5  e: r3
6  f: r3