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
1 class BinOp {
2   optype(): String {
3       "BinOp"
4   };
5
6   operate(a: Int, b: Int): Int {
7       a + b
8   };
9 }
10
11 class SumOp inherits BinOp {
12   optype(): String {
13       "SumOp"
14   };
15 }
16
17 class MulOp inherits BinOp {
18   optype(): String {
19       "MulOp"
20   };
21
22   operate(a: Int, b: Int): Int {
23       a * b
24   };
25 }
26
27 class IntList {
28   head: Int;
29   tail: IntList;
30   empty_tail: IntList; -- Do not assign.
31
32   tail_is_empty(): Bool {
33       tail = empty_tail
34   };
35
36   get_head(): Int { head };
37
38   set_head(n: Int): Int {
39       head <- n
40   };
41```
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>maddr(_1): cgen_m(BinOp_opts)</td>
</tr>
<tr>
<td>maddr(_2): cgen_m(BinOp_operate)</td>
</tr>
<tr>
<td>maddr(_3): cgen_m(SumOp_opts)</td>
</tr>
<tr>
<td>maddr(_4): cgen_m(MulOp_operate)</td>
</tr>
<tr>
<td>maddr(_5): cgen_m(MulOp_opts)</td>
</tr>
<tr>
<td>maddr(_6): cgen_m(IntList_tail_is_empty)</td>
</tr>
<tr>
<td>maddr(_7): cgen_m(IntList_get_head)</td>
</tr>
<tr>
<td>maddr(_8): cgen_m(IntList_set_head)</td>
</tr>
<tr>
<td>maddr(_9): cgen_m(IntList_get_tail)</td>
</tr>
<tr>
<td>maddr(_10): cgen_m(IntList_set_tail)</td>
</tr>
<tr>
<td>maddr(_11): cgen_m(IntList_generate)</td>
</tr>
<tr>
<td>maddr(_12): cgen_m(Main_reduce)</td>
</tr>
<tr>
<td>maddr(_13): cgen_m(Main_main)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dispatch tables:</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr(_14): DT[BinOp]</td>
</tr>
<tr>
<td>maddr(_15): DT[SumOp]</td>
</tr>
<tr>
<td>maddr(_16): DT[MulOp]</td>
</tr>
<tr>
<td>maddr(_17): DT[IntList]</td>
</tr>
<tr>
<td>maddr(_18): DT[Main]</td>
</tr>
</tbody>
</table>

In the above, 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.
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.

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.

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>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>l</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>IntList.generate</td>
<td>maddr_{19} - 4</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_{19} - 24</td>
<td>IntList.generate</td>
<td>3</td>
<td>arg_{1}</td>
</tr>
<tr>
<td>maddr_{19} - 28</td>
<td>IntList.generate</td>
<td>self</td>
<td>arg_{0}</td>
</tr>
<tr>
<td>maddr_{19} - 32</td>
<td>IntList.generate</td>
<td>maddr_{13} + δ</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_{19} - 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 `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.

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”.

4
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 not is 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.
3. Suppose you want to add a for-loop construct to Cool, having the following syntax:

$$\text{expr ::= ...}\quad |\quad \text{for expr to expr do expr rof}$$

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

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.

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

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

\[
\begin{align*}
1 & \quad x := 0 \times 5 \\
2 & \quad y := a + b \\
3 & \quad z := x \times x \\
4 & \quad c := y \times x \\
5 & \quad x := x + 4 \\
6 & \quad e := c - x \\
7 & \quad x := e \times x \\
8 & \quad f := a + b \\
9 & \quad y := y + f
\end{align*}
\]

(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

(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).
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.

(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?

(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.).