1. (20 pts) Consider the following program in Cool, representing a “slightly” over-engineered implementation which calculates the factorial of 4 using an operator class and a reduce() method:

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

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;
    }
  }
};
```
```java
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(4);
            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):
## Code segment:

<table>
<thead>
<tr>
<th>maddr</th>
<th>Code Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr_1</td>
<td>cgen_m(BinOp.operate)</td>
</tr>
<tr>
<td>maddr_2</td>
<td>cgen_m(BinOp.optype)</td>
</tr>
<tr>
<td>maddr_3</td>
<td>cgen_m(SumOp.optype)</td>
</tr>
<tr>
<td>maddr_4</td>
<td>cgen_m(MulOp.optype)</td>
</tr>
<tr>
<td>maddr_5</td>
<td>cgen_m(MulOp.operate)</td>
</tr>
<tr>
<td>maddr_6</td>
<td>cgen_m(IntList.tail_is_empty)</td>
</tr>
<tr>
<td>maddr_7</td>
<td>cgen_m(IntList.get_head)</td>
</tr>
<tr>
<td>maddr_8</td>
<td>cgen_m(IntList.set_head)</td>
</tr>
<tr>
<td>maddr_9</td>
<td>cgen_m(IntList.get_tail)</td>
</tr>
<tr>
<td>maddr_{10}</td>
<td>cgen_m(IntList.set_tail)</td>
</tr>
<tr>
<td>maddr_{11}</td>
<td>cgen_m(IntList.generate)</td>
</tr>
<tr>
<td>maddr_{12}</td>
<td>cgen_m(Main.reduce)</td>
</tr>
<tr>
<td>maddr_{13}</td>
<td>cgen_m(Main.main)</td>
</tr>
</tbody>
</table>

## Dispatch tables:

<table>
<thead>
<tr>
<th>maddr</th>
<th>Dispatch Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>maddr_{14}</td>
<td>DT[BinOp]</td>
</tr>
<tr>
<td>maddr_{15}</td>
<td>DT[SumOp]</td>
</tr>
<tr>
<td>maddr_{16}</td>
<td>DT[MulOp]</td>
</tr>
<tr>
<td>maddr_{17}</td>
<td>DT[IntList]</td>
</tr>
<tr>
<td>maddr_{18}</td>
<td>DT[Main]</td>
</tr>
</tbody>
</table>

### Stack (maddr_{19})

![Stack Diagram]

### Heap

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.
(a) (4 pts) Assume the MIPS assembly code to be stored starting at address maddr_{12} and ending immediately before maddr_{13} (i.e. not including the instruction starting at maddr_{13}) was generated using the code generation process from Lecture 12. How many instructions using the frame pointer register ($fp$) will be present within such code? Why?

Answer: Address maddr_{12} corresponds to the code generated for the definition of method Main.reduce(). The MIPS code corresponding to this method is quite large, however, we can make use of the fact that $fp$ is used exclusively in two cases: during the code generated by our calling convention and when accessing the method’s arguments. Following the calling convention of slides 22-23 of Lecture 12, we will have a move and a load on $fp$ generated for the definition of the method reduce() itself, independent of its body:

```
move $fp $sp
...
lw $fp 0($sp)
jr $ra
```

Additionally, every method call included in the body of reduce() will begin by storing the current frame pointer into the stack, at the beginning of the called method’s AR. There are 5 such calls inside the body of reduce(): l.get_head(), op.operate(), l.tail_is_empty(), l.get_tail() and the recursive call to Main.reduce().

Finally, whenever the code references one of the arguments to Main.reduce(), we generate code that loads it from a stack address based on an offset from the frame pointer. If the operation is an assignment, the generated code instead stores into that address. Either way, a single instruction is generated involving $fp$. The method includes 9 explicit references to arguments as sub-expressions. Additionally, there is an implicit reference to self required to retrieve the dispatch table necessary for the recursive call to Main.reduce().

This gives us a total of 17 instructions making use of the frame pointer ($2$ for the calling convention for the method itself, $5$ for the methods called and $10$ for the arguments accessed). It is not clear from Lecture 12 whether or not self is passed as an argument in the stack or in a register, so 16 is also an acceptable answer.
(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>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.

**Answer:**

**BinOp:**

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>operate</td>
<td>maddr_1</td>
</tr>
<tr>
<td>1</td>
<td>optype</td>
<td>maddr_2</td>
</tr>
</tbody>
</table>

**SumOp** (note that operate references the same body as BinOp.operate):

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>operate</td>
<td>maddr_1</td>
</tr>
<tr>
<td>1</td>
<td>optype</td>
<td>maddr_3</td>
</tr>
</tbody>
</table>

**MulOp** (note the order needs to be compatible with BinOp):

<table>
<thead>
<tr>
<th>Method Idx</th>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>operate</td>
<td>maddr_5</td>
</tr>
<tr>
<td>1</td>
<td>optype</td>
<td>maddr_4</td>
</tr>
</tbody>
</table>

**IntList** (this one is actually fairly straightforward):

<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>

Note that we actually forgot to ask for MulOp in the question, which was one of the interesting cases. Provided here for educational purposes.
(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.

**Answer:** At Point A the following objects have been added to the heap (none would be collected even if our implementation of Cool had a GC): 1 Main object, 1 MulOp object and 2 IntList objects. All this objects have the default values for each of their attributes, as none of them have been assigned to yet. The corresponding object layouts look like this (we are mostly looking for the presence of all necessary elements: class tag, object size, dispatch pointer and attributes, correct value of the object size and some reference to the correct dispatch table):

<table>
<thead>
<tr>
<th>Class tag</th>
<th>Object size</th>
<th>Dispatch ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>maddr₁₈</td>
</tr>
<tr>
<td></td>
<td>12 (3 words)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class tag</th>
<th>Object size</th>
<th>Dispatch ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>maddr₁₆</td>
</tr>
<tr>
<td></td>
<td>12 (3 words)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class tag</th>
<th>Object size</th>
<th>Dispatch ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>maddr₁₇</td>
</tr>
<tr>
<td></td>
<td>24 (6 words)</td>
<td></td>
</tr>
</tbody>
</table>

| head      | 0           | maddr₁₇      |
| tail      | void        |              |
| empty_tail| void        |              |

x2
(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>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>IntList.generate</td>
<td>maddr_{19}</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_{19} + 24</td>
<td>IntList.generate</td>
<td>self</td>
<td>arg_{0}</td>
</tr>
<tr>
<td>maddr_{19} + 28</td>
<td>IntList.generate</td>
<td>4</td>
<td>arg_{1}</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”.

Answer:
<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}</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_{19} + 24</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_{0}</td>
</tr>
<tr>
<td>maddr_{19} + 28</td>
<td>Main.reduce</td>
<td>4</td>
<td>arg_{1}</td>
</tr>
<tr>
<td>maddr_{19} + 32</td>
<td>Main.reduce</td>
<td>op</td>
<td>arg_{2}</td>
</tr>
<tr>
<td>maddr_{19} + 36</td>
<td>Main.reduce</td>
<td>1</td>
<td>arg_{3}</td>
</tr>
<tr>
<td>maddr_{19} + 40</td>
<td>Main.reduce</td>
<td>maddr_{13} + \delta</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_{19} + 44</td>
<td>Main.reduce</td>
<td>maddr_{19} + 24</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_{19} + 48</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_{0}</td>
</tr>
<tr>
<td>maddr_{19} + 52</td>
<td>Main.reduce</td>
<td>12</td>
<td>arg_{1}</td>
</tr>
<tr>
<td>maddr_{19} + 56</td>
<td>Main.reduce</td>
<td>op</td>
<td>arg_{2}</td>
</tr>
<tr>
<td>maddr_{19} + 60</td>
<td>Main.reduce</td>
<td>1</td>
<td>arg_{3}</td>
</tr>
<tr>
<td>maddr_{19} + 64</td>
<td>Main.reduce</td>
<td>maddr_{12} + \delta</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_{19} + 68</td>
<td>Main.reduce</td>
<td>maddr_{19} + 48</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_{19} + 72</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_{0}</td>
</tr>
<tr>
<td>maddr_{19} + 76</td>
<td>Main.reduce</td>
<td>24</td>
<td>arg_{1}</td>
</tr>
<tr>
<td>maddr_{19} + 80</td>
<td>Main.reduce</td>
<td>op</td>
<td>arg_{2}</td>
</tr>
<tr>
<td>maddr_{19} + 84</td>
<td>Main.reduce</td>
<td>1</td>
<td>arg_{3}</td>
</tr>
<tr>
<td>maddr_{19} + 88</td>
<td>Main.reduce</td>
<td>maddr_{12} + \delta</td>
<td>Return</td>
</tr>
<tr>
<td>maddr_{19} + 92</td>
<td>Main.reduce</td>
<td>maddr_{19} + 72</td>
<td>FP</td>
</tr>
<tr>
<td>maddr_{19} + 96</td>
<td>Main.reduce</td>
<td>self</td>
<td>arg_{0}</td>
</tr>
<tr>
<td>maddr_{19} + 100</td>
<td>Main.reduce</td>
<td>24</td>
<td>arg_{1}</td>
</tr>
<tr>
<td>maddr_{19} + 104</td>
<td>Main.reduce</td>
<td>op</td>
<td>arg_{2}</td>
</tr>
<tr>
<td>maddr_{19} + 108</td>
<td>Main.reduce</td>
<td>1</td>
<td>arg_{3}</td>
</tr>
<tr>
<td>maddr_{19} + 112</td>
<td>Main.reduce</td>
<td>maddr_{12} + \delta</td>
<td>Return</td>
</tr>
</tbody>
</table>
2. (4 pts) Consider the following arithmetic expression: \((82 + 19) - (12/4) \times 9 + 5 \times (13 + 4)\).

(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 will execute the operations given in the expression above in their original order (e.g. it 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)?

Answer: 7 pushes and 7 pops. The pseudo-code below describes a possible translation of the expression above to a stack machine with an accumulator (note that the following is not actual MIPS machine code or directly translatable to it, it is meant only to show the operation of the abstract stack machine with accumulator model).

```plaintext
1 acc <- 82
2 push acc // 1 push
3 acc <- 19
4 acc <- sum 0(sp) acc => 82 + 19 => 101 // 1 pop
5 push acc // 2 push
6 acc <- 12
7 push acc // 3 push
8 acc <- 4
9 acc <- div 0(sp) acc => 12 / 4 => 3 // 2 pop
10 push acc // 4 push
11 acc <- 9
12 acc <- mul 0(sp) acc => 3 * 9 => 27 // 3 pop
13 acc <- sub 0(sp) acc => 101 - 27 => 74 // 4 pop
14 push acc // 5 push
15 acc <- 5
16 push acc // 6 push
17 acc <- 13
18 push acc // 7 push
19 acc <- sum 0(sp) acc => 13 + 4 => 17 // 5 pop
20 acc <- mul 0(sp) acc => 5 * 17 => 85 // 6 pop
21 acc <- sum 0(sp) acc => 74 + 85 => 159 // 7 pop
```

Note that only the numbers of pushes and pops are required to correctly answer this question.
(b) **(2 pts)** You are now given MIPS code that evaluates the same expression using a register machine with only 2 registers. Again, this code includes no optimizations and will perform every operation in the original expression, in the same order. How many loads from and stores to memory will this code perform, at a minimum (give a separate count for loads and for stores)?

**Answer:** 3 loads and 3 stores. The pseudo-code below describes a possible translation of the expression above to a machine with a stack and two registers (again, this is an abstract model, not proper MIPS code). We use push and pop to store spilled registers into memory, so that it is easier to track which values we are saving or restoring (since the stack forces a particular order on the results in memory). For this expression, we never need to load values from memory in a different order than the one used to store them, thus using a stack as our model does not prevent us from reaching the optimal number of loads and stores.

```
1   r1 <- 82
2   r2 <- 19
3   r1 <- r1 + r2 => 82 + 19 => 101
4   r2 <- 12
5   push r1 // 1 store
6   r1 <- 4
7   r1 <- r2 / r1 => 12 / 4 => 3
8   r2 <- 9
9   r1 <- r1 * r2 => 3 * 9 => 27
10  pop r2 // 1 load
11  r1 <- r2 - r1 => 101 - 27 => 74
12  r2 <- 5
13  push r1 // 2 store
14  r1 <- 13
15  push r2 // 3 store
16  r2 <- 4
17  r1 <- r1 + r2 => 13 + 4 => 17
18  pop r2 // 2 load
19  r1 <- r2 * r1 => 5 * 17 => 85
20  pop r2 // 3 load
21  r1 <- r2 + r1 => 74 + 85 => 159
```

Note that only the numbers of loads and stores are required to correctly answer this question.
3. (16 pts) Suppose you want to add a for-loop construct to Cool, having the following syntax:

\[
\text{for id : Int} \leftarrow e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof}
\]

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 value of \( \text{id} \) starting with the value of \( e_1 \) and incremented by 1 thereafter until reaching the value of \( e_2 \) (inclusive). Similar to the while loop, the for-loop returns void.

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

**Answer:** These are similar to the operational semantics of the let and while expressions, with some care applied to requirement that \( e_1 \) and \( e_2 \) be evaluated only once. First, we need a rule for when the for loop executes zero times (the bounds don’t include any number):

\[
\frac{so, S, E \vdash e_1 : \text{Int}(i_1), S_1 \quad so, S_1, E \vdash e_2 : \text{Int}(i_2), S_2 \quad i_1 > i_2}{so, S, E \vdash \text{for id : Int} \leftarrow e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof} : \text{void}, S_2}
\]

Then the general case where we do enter the loop:

\[
\begin{align*}
so, S, E & \vdash e_1 : \text{Int}(i_1), S_1 \\
so, S_1, E & \vdash e_2 : \text{Int}(i_2), S_2 \\
i_1 & \leq i_2 \\
l_1 & = \text{newloc}(S_2) \\
so, S_2[\text{Int}(i_1)/l_1], E[l_1/id] & \vdash e_3 : v_3, S_3 \\
i_n & = S_3[l_1] + 1 \\
so, S_3, E & \vdash \text{for id : Int} \leftarrow i_n \text{ to } i_2 \text{ do } e_3 \text{ rof} : \text{void}, S_4 \\
so, S, E & \vdash \text{for id : Int} \leftarrow e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof} : \text{void}, S_4
\end{align*}
\]

Where \( \text{Int}(i_k) \) represents a Cool Int object with \( i_k \) as its corresponding numerical value. When used in the program syntax, \( i_k \) refers to the corresponding integer literal.
(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.

**Answer:** Note that id is a new local introduced by the for construct. Thus, it gets assigned a space in the AR. In the code below we will assume that a variable i is defined while generating the code for the body of each method, and contains the offset from the frame pointer at which id should be stored in the AR. Also, we assume our compiler does integer unboxing, so we can operate directly with id (and other Cool Ints) as MIPS int32 values directly, avoiding additional accesses to the heap.

```plaintext
1  cgen(for id: Int <- e₁ to e₂ do e₃ rof, nt) =
2      cgen(e₁, nt)
3      sw $a0 i($fp)       # Save id=val(e₁) on i-th var
4      cgen(e₂, nt+4)
5      sw $a0 4($sp)      # Save v2=val(e₂) on stack
6      addiu $sp $sp -4
7  for_loop:
8      lw $a0 i($fp)      # current value of id
9      lw $t₁ 4($sp)     # value of e₂
10     bgt $a0 $t₁ for_exit # Exit if id > v2
11     cgen(e₃, nt)      # Might change id and clobber $a₀
12     lw $a₀ i($fp)     # Load id again
13     addiu $a₀ $a₀ 1   # Increment it by 1
14     sw $a₀ i($fp)     # Save it back
15     b for_loop        # Jump back to the loop check
16  for_exit:
17      addiu $sp $sp 4  # Remove value of e₂ from stack
```
4. (8 pts) Consider the following basic block, in which all variables are integers.

```plaintext
1  a := 0 * f
2  b := f * v
3  c := b + b
4  d := a * 3
5  x := f * v
6  y := x + x
7  z := y / c
```

Assume that the only variables that are live at the exit of this block are y and z, while v and f 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, 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 optimizations, in what order, can you apply to optimize the result further?

**Answer:**

(a) Algebraic simplification

```plaintext
1  a := 0 * f
2  b := f * v
3  c := b + b
4  d := a * 3
5  x := f * v
6  y := x + x
7  z := y / c
```

1  a := 0

(b) Copy propagation

```plaintext
1  a := 0 * f
2  b := f * v
3  c := b + b
4  d := a * 3
5  x := f * v
6  y := x + x
7  z := y / c
```

1  a := 0

2  b := f * v

3  c := b + b

4  d := 0 * 3

5  x := f * v

6  y := x + x

7  z := y / c

13
(c) Common sub-expression elimination

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0 * 3
5 x := f * v
6 y := x + x
7 z := y / c

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0 * 3
5 x := b
6 y := x + x
7 z := y / c

(d) Constant folding

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0
5 x := b
6 y := x + x
7 z := y / c

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0
5 x := b
6 y := b + b
7 z := y / c

(e) Copy propagation

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0
5 x := b
6 y := b + b
7 z := y / c

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0
5 x := b
6 y := b + b
7 z := y / c

(f) Dead code elimination

1 a := 0
2 b := f * v
3 c := b + b
4 d := 0
5 x := b
6 y := b + b
7 z := y / c

We can then add one extra round of common sub-expression elimination and copy propagation:

(a) Common sub-expression elimination

1 b := f * v
2 c := b + b
3 y := b + b
4 z := y / c
1 b := f * v
2 c := b + b
3 y := c
4 z := y / c
(b) Copy propagation:

\[
\begin{align*}
&1 \quad b := f \ast v \\
&2 \quad c := b + b \\
&3 \quad y := c \\
&4 \quad z := y / c
\end{align*}
\]

\[
\begin{align*}
&1 \quad b := f \ast v \\
&2 \quad c := b + b \\
&3 \quad y := c \\
&4 \quad z := c / c
\end{align*}
\]

No other optimizations covered during the lecture apply at this point. We can’t eliminate \(y\), as it is live on exit from the basic block. We also can’t perform arithmetic simplification on the division of the last instruction, since \(c\) might be 0, leading to an unsafe optimization.
5. (12 pts) Consider the following assembly-like pseudo-code, using 11 temporaries (abstract registers) \( t_0 \) to \( t_{10} \):

\[
\begin{array}{l}
1. \quad t_1 := t_0 \times t_0 \\
2. \quad t_2 := -t_1 \\
3. \quad t_3 := t_2 + t_0 \\
4. \quad \text{if } t_3 > 0: \\
5. \quad \quad t_4 := t_0 - t_3 \\
6. \quad \quad t_5 := t_2 \times 3 \\
7. \quad \quad t_6 := t_4 - t_5 \\
8. \quad \text{else:} \\
9. \quad \quad t_7 := t_0 + t_3 \\
10. \quad \quad t_8 := t_2 \times 3 \\
11. \quad \quad t_6 := t_7 + t_8 \\
12. \quad t_9 := t_0 \times t_3 \\
13. \quad t_{10} := t_6 + t_9
\end{array}
\]

Using the graph coloring algorithm for register allocation described in class, provide a version of this code that uses as few registers possible (number your registers by: \( r_0, r_1, \ldots \) etc). Note that \( t_0 \) is the only input temporary for this code and \( t_{10} \) will be the only live value on exit. You must also indicate, for the original temporaries, which of them are live at the end of each statement. You should also provide the k-colored graph generated by your algorithm (you may use the tikz package to typeset it or simply embed an image).

**Answer:** The following shows the lifetime of each temporary in the pseudo-code above:

\[
\begin{array}{l}
1. \quad \# \text{Live: } t_0 \text{ (input)} \\
2. \quad \# \text{Live: } t_0, t_1 \\
3. \quad \# \text{Live: } t_0, t_2 \\
4. \quad \# \text{Live: } t_0, t_2, t_3 \\
5. \quad \# \text{Live: } t_0, t_2, t_3 \\
6. \quad \# \text{Live: } t_0, t_2, t_3, t_4 \\
7. \quad \# \text{Live: } t_0, t_3, t_4, t_5 \\
8. \quad \# \text{Live: } t_0, t_3, t_6 \\
9. \quad \# \text{Live: } t_0, t_2, t_3 \\
10. \quad \# \text{Live: } t_0, t_2, t_3, t_4 \\
11. \quad \# \text{Live: } t_0, t_2, t_3, t_6 \\
12. \quad \# \text{Live: } t_0, t_3, t_7 \\
13. \quad \# \text{Live: } t_0, t_3, t_7, t_8 \\
14. \quad \# \text{Live: } t_0, t_3, t_6 \\
15. \quad \# \text{Live: } t_6, t_9 \\
16. \quad \# \text{Live: } t_6, t_{10}
\end{array}
\]
Note that, since there are points in time at which 4 temporaries are live (e.g. t0, t2, t3 and t4), there is no way of solving this allocation problem with less than $K = 4$ registers.

Based on the information of which temporaries are live at the same time, we can produce the following interference graph:

![Interference Graph]

We progressively remove nodes with less than 4 neighbors in the graph, since those can always be colored once their neighbors themselves have been colored:
This shows that the graph is four colorable, and in fact, by reintroducing the nodes in the order we removed them, we quickly arrive to a suitable coloring with 4 colors:
Note that the coloring is not unique. For example, t6 could have been assigned blue as opposed to orange.

Once we have the graph, we can assign the temporaries corresponding to each color to one of the following 4 registers: r0 (green), r1 (blue), r2 (red), r3 (orange). This gives us the following code for the exercise, using exactly 4 registers:

```
1   r1 := r0 * r0
2   r1 := -r1
3   r2 := r1 + r0
4   if r2 > 0:
5       r3 := r0 - r2
6       r1 := r1 * 3
7       r3 := r3 - r1
8   else:
9       r3 := r0 + r2
10      r1 := r1 * 3
11      r3 := r3 + r1
12     r0 := r0 * r2
13     r0 := r3 + r0
```