CS143 - Written Assignment 4 Reference Solutions

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:

```
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                    l.set_tail(empty_tail)
43                else
44                    l.set_tail(generate(n-1))
45                fi;
46                l;
47            }
48        };
49    };
50    class Main {
```

```
51          reduce(result: Int, op: BinOp, l: IntList): Int {{
52              result <- op.operate(result,l.get_head());
53              if (l.tail_is_empty() = true) then
54                  -- Point B
55                  result
56              else
57                  reduce(result,op,l.get_tail())
58              fi;
59          }};
60          main(): Object {
61              let op: BinOp <- new MulOp, l: IntList <- new IntList, io:
                    IO <- new IO in {
62                  l <- l.generate(3);
63                  io.out_int(self.reduce(1,op,l));
64              }
65          };
66      };
```

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: (*low address*)

$\texttt{maddr}_1$:	`cgen_m(BinOp.operate)`
$\texttt{maddr}_2$:	`cgen_m(BinOp.optype)`
$\texttt{maddr}_3$:	`cgen_m(SumOp.optype)`
$\texttt{maddr}_4$:	`cgen_m(MulOp.operate)`
$\texttt{maddr}_5$:	`cgen_m(MulOp.optype)`
$\texttt{maddr}_6$:	`cgen_m(IntList.tail_is_empty)`
$\texttt{maddr}_7$:	`cgen_m(IntList.get_head)`
$\texttt{maddr}_8$:	`cgen_m(IntList.set_head)`
$\texttt{maddr}_9$:	`cgen_m(IntList.get_tail)`
$\texttt{maddr}_{10}$:	`cgen_m(IntList.set_tail)`
$\texttt{maddr}_{11}$:	`cgen_m(IntList.generate)`
$\texttt{maddr}_{12}$:	`cgen_m(Main.reduce)`
$\texttt{maddr}_{13}$:	`cgen_m(Main.main)`

Dispatch tables:

$\texttt{maddr}_{14}$:	`DT[BinOp]`
$\texttt{maddr}_{15}$:	`DT[SumOp]`
$\texttt{maddr}_{16}$:	`DT[MulOp]`
$\texttt{maddr}_{17}$:	`DT[IntList]`
$\texttt{maddr}_{18}$:	`DT[Main]`

Heap

↓

↑

Stack ($\texttt{maddr}_{19}$) (*high address*)

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. Note that the stack starts at a high address and grows towards lower addresses.

(a) 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 12 (beginning at Slide 18). In particular, assume that the caller is responsible for saving the frame pointer, and the callee is responsible for 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?

Answer: For each dispatch, the frame pointer is saved (1 use). Each read and write of `self` or a function parameter is a use as well. In Main.reduce, there are five method calls (op.operate(), l.get_head(), l.tail_is_empty(), l.get_tail(), and self.reduce()). There is one write to a parameter (result). There are 8 other reads of parameters (op on line 52, result on line 52, l on line 52, l on line 53, result on line 55, result on line 57, op on line 57, l on line 57) and one implicit use of self (in the call to reduce on line 57). Finally, at the beginning of a function declaration, the frame pointer is overwritten with the current stack pointer; at the end of the declaration, the old frame pointer is restored. There are thus $5 + 1 + 8 + 1 + (1 + 1) = 17$ uses of the frame pointer.

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

Method Idx	Method Name	Address
0	reduce	$\texttt{maddr}_{12}$
1	main	$\texttt{maddr}_{13}$

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

Answer:

BinOp:

Method Idx	Method Name	Address
0	operate	$\texttt{maddr}_{1}$
1	optype	$\texttt{maddr}_{2}$

SumOp :

Method Idx	Method Name	Address
0	operate	$\texttt{maddr}_{1}$
1	optype	$\texttt{maddr}_{3}$

MulOp:

Method Idx	Method Name	Address
0	operate	$\texttt{maddr}_{4}$
1	optype	$\texttt{maddr}_{5}$

IntList:

Method Idx	Method Name	Address
0	tail_is_empty	$\texttt{maddr}_{6}$
1	get_head	$\texttt{maddr}_{7}$
2	set_head	$\texttt{maddr}_{8}$
3	get_tail	$\texttt{maddr}_{9}$
4	set_tail	$\texttt{maddr}_{10}$
5	generate	$\texttt{maddr}_{11}$

(c) 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.

Answer:

Main

5
3
`maddr`$_{18}$

MulOp

3
3
`maddr`$_{16}$

IntList (in Main.main)

4
6
`maddr`$_{17}$
0
void
void

IntList (in IntList.generate)

4
6
`maddr`$_{17}$
0
void
void

(d) 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":

Address	Method	Contents	Description
$\texttt{maddr}_{19}$	Main.main	self	arg_0
$\texttt{maddr}_{19} - 4$	Main.main	...	Return
$\texttt{maddr}_{19} - 8$	Main.main	op	local
$\texttt{maddr}_{19} - 12$	Main.main	l	local
$\texttt{maddr}_{19} - 16$	Main.main	io	local
$\texttt{maddr}_{19} - 20$	IntList.generate	$\texttt{maddr}_{19} - 4$	FP
$\texttt{maddr}_{19} - 24$	IntList.generate	3	arg_1
$\texttt{maddr}_{19} - 28$	IntList.generate	self	arg_0
$\texttt{maddr}_{19} - 32$	IntList.generate	$\texttt{maddr}_{13} + \delta$	Return
$\texttt{maddr}_{19} - 36$	IntList.generate	l	local

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.

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 $\texttt{maddr}_i + \delta$, denoting an unknown address between $\texttt{maddr}_i$ and $\texttt{maddr}_{i+1}$. 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. For objects that have no variable names, you may give a short description of the object (e.g., "ptr to [2, 1]" to represent a pointer to an IntList consisting of [2, 1]).

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".

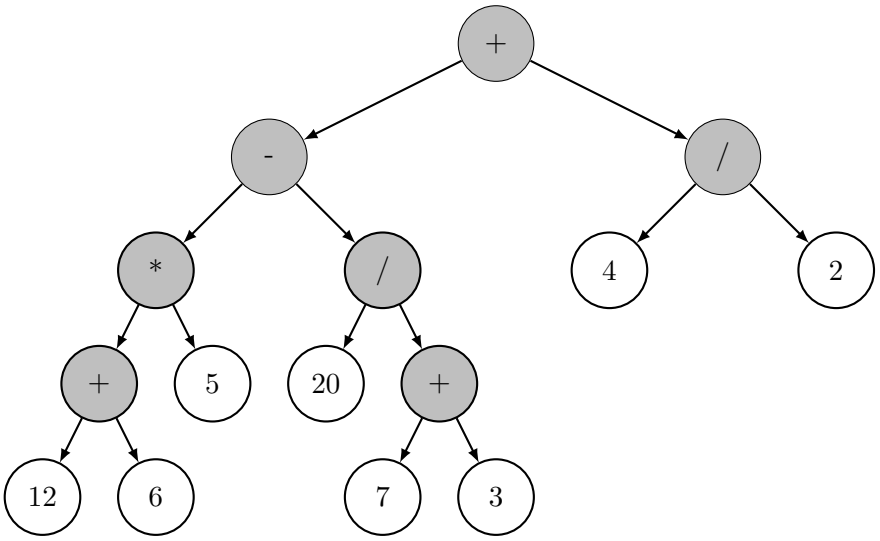
Answer:

Address	Method	Contents	Description
$\texttt{maddr}_{19}$	Main.main	self	arg_0
$\texttt{maddr}_{19} - 4$	Main.main	...	Return
$\texttt{maddr}_{19} - 8$	Main.main	op	local
$\texttt{maddr}_{19} - 12$	Main.main	l	local
$\texttt{maddr}_{19} - 16$	Main.main	io	local
$\texttt{maddr}_{19} - 20$	Main.reduce	$\texttt{maddr}_{19} - 4$	FP
$\texttt{maddr}_{19} - 24$	Main.reduce	ptr to [3, 2, 1]	arg_3
$\texttt{maddr}_{19} - 28$	Main.reduce	ptr to MulOp	arg_2
$\texttt{maddr}_{19} - 32$	Main.reduce	3	arg_1
$\texttt{maddr}_{19} - 36$	Main.reduce	self	arg_0
$\texttt{maddr}_{19} - 40$	Main.reduce	$\texttt{maddr}_{13} + \delta_1$	Return
$\texttt{maddr}_{19} - 44$	Main.reduce	$\texttt{maddr}_{19} - 40$	FP
$\texttt{maddr}_{19} - 48$	Main.reduce	ptr to [2, 1]	arg_3
$\texttt{maddr}_{19} - 52$	Main.reduce	ptr to MulOp	arg_2
$\texttt{maddr}_{19} - 56$	Main.reduce	6	arg_1
$\texttt{maddr}_{19} - 60$	Main.reduce	self	arg_0
$\texttt{maddr}_{19} - 64$	Main.reduce	$\texttt{maddr}_{12} + \delta_2$	Return
$\texttt{maddr}_{19} - 68$	Main.reduce	$\texttt{maddr}_{19} - 64$	FP
$\texttt{maddr}_{19} - 72$	Main.reduce	ptr to [1]	arg_3
$\texttt{maddr}_{19} - 76$	Main.reduce	ptr to MulOp	arg_2
$\texttt{maddr}_{19} - 80$	Main.reduce	6	arg_1
$\texttt{maddr}_{19} - 84$	Main.reduce	self	arg_0
$\texttt{maddr}_{19} - 88$	Main.reduce	$\texttt{maddr}_{12} + \delta_2$	Return

2. Consider the following arithmetic expression: $(12 + 6) * 5 - (20/(7 + 3)) + (4/2)$.

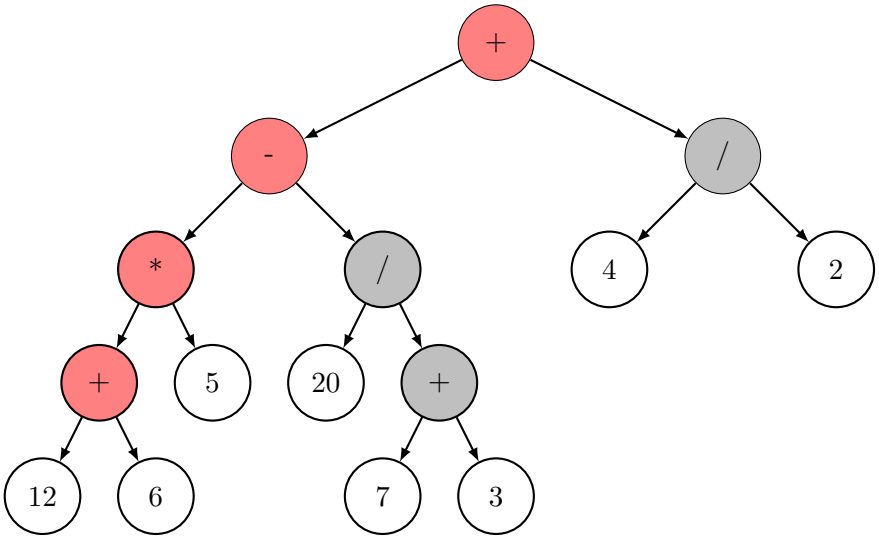
   (a) You are given MIPS code that evaluates this expression using a stack machine with a single accumulator register (similar to the method given in class Lecture 12). This code is wholly unoptimized and will execute the operations given in the expression above in their original order (e.g. it does not 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: Below is a parse tree for this expression. For each shaded node, code is generated to compute the value of the left child, and then this value is pushed onto the stack. After computing the value of the right child, the previously computed value is popped. Therefore there will be 7 pushes and 7 pops in total.



(b) Now suppose that you have access to two registers `r1` and `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?

Answer:



Instead of pushing and popping from the stack, the intermediate results are held in temporary locations. We use `r2` to hold the first temporary. Following the procedure in lecture, when generating code for the root node we store the result of its left child in the first temporary (`r2`) while evaluating the right child. We also allow the computation of the left subexpression of the root node to use `r2`. We recurse on the left subtree with `r2` as an available temporary until reaching the leftmost '+' node with depth 3. As a result, all of the shaded nodes of the parse tree reached by repeatedly following the left child node are able to use `r2` as a temporary (colored in red), so thus, we save a load and store for such nodes. Hence, we only need a load and store for all the nodes shaded in gray. We end up requiring 3 loads and 3 stores.

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

$$\text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof}$$

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 integer in the range $[e_1, e_2]$ (inclusive) in order. Similar to the while loop, the for-loop returns void.

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

Answer: There are multiple ways to solve this problem; one solution is as follows:

$$\frac{\begin{array}{c} so, S, E \vdash e_1 \mapsto Int(n_1), S_1 \\ so, S_1, E \vdash e_2 \mapsto Int(n_2), S_2 \\ so, S_2, E \vdash e_3 \mapsto v, S_3 \\ n_1 \leq n_2 \\ so, S_3, E \vdash \text{for } n_1 + 1 \text{ to } n_2 \text{ do } e_3 \text{ rof} \mapsto v', S_4 \end{array}}{so, S, E \vdash \text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof} \mapsto void, S_4}$$

$$\frac{\begin{array}{c} so, S, E \vdash E_1 \mapsto Int(n_1), S_1 \\ so, S_1, E \vdash e_2 \mapsto Int(n_2), S_2 \\ n1 > n2 \end{array}}{so, S, E \vdash \text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof} \mapsto void, S_2}$$

(b) Give the code generation function cgen(for e_1 to e_2 do e_3 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: There are multiple possible solutions here. One possible solution is as follows:

```
1         cgen(e1)             # compute lower bound (store in a0)
2         sw $a0 0($sp)        # push a0 onto stack
3         addiu $sp, $sp, -4
4         cgen(e2)             # compute upper bound (store in a0)
5         sw $a0 0($sp)        # push a0 onto the stack
6         addiu $sp, $sp, -4
7         j compare            # jump to the end of the loop where
8                              # comparison is performed
9       loop:
10        addiu $t0, $t0, 1    # increment counter
11        sw $t0, 8($sp)       # save counter back to stack
12        cgen(e3)             # execute loop iteration
13      compare:
14        lw $a0, 4($sp)       # load a0 with upper bound
15        lw $t0, 8($sp)       # load t0 with lower bound
16        ble $t0, $a0, loop   # repeat loop if within bounds
17        addiu $sp, $sp, 8    # pop the stack
18        move $a0, $0         # return void
```

Note that in this solution, we assume that integers are unboxed, as they are in the lecture. We accept a solution with boxed integers (like those in Cool) for full credit.

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

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

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 optimization.

(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?

Answer:

(a) Algebraic simplification

```
1           a := f * f
2           b := a
3           c := 2 + 8
4           d := c * b
5           e := f * f
6           x := e + d
7           g := b + d
8           h := b + d
9           i := g
10          y := i / h
```

(b) Copy propagation

```
1           a := f * f
2           b := a
```

```
3           c := 2 + 8
4           d := c * a
5           e := f * f
6           x := e + d
7           g := a + d
8           h := a + d
9           i := g
10          y := g / h
```

(c) Common sub-expression elimination

```
1           a := f * f
2           b := a
3           c := 2 + 8
4           d := c * a
5           e := a
6           x := e + d
7           g := a + d
8           h := g
9           i := g
10          y := g / h
```

(d) Constant folding

```
1           a := f * f
2           b := a
3           c := 10
4           d := c * a
5           e := a
6           x := e + d
7           g := a + d
8           h := g
9           i := g
10          y := g / h
```

(e) Copy propagation

```
1           a := f * f
2           b := a
3           c := 10
4           d := 10 * a
5           e := a
6           x := a + d
7           g := a + d
8           h := g
9           i := g
10          y := g / g
```

(f) Dead code elimination

```
1           a := f * f
2           d := 10 * a
```

```
3           x := a + d
4           g := a + d
5           y := g / g
```

We can then perform common sub-expression elimination, copy propagation, and dead code elimination, in that order, to achieve the following:

(a) Common sub-expression elimination

```
1           a := f * f
2           d := 10 * a
3           x := a + d
4           g := x
5           y := g / g
```

(b) Copy propagation

```
1           a := f * f
2           d := 10 * a
3           x := a + d
4           g := x
5           y := x / x
```

(c) Dead code elimination

```
1           a := f * f
2           d := 10 * a
3           x := a + d
4           y := x / x
```

No other optimizations covered during the lecture apply at this point. We cannot perform arithmetic simplification on the division of the last instruction since x might be 0, leading to an unsafe optimization.

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

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

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

Answer:

```
1                                   LIVE: t0
2             t1 = t0 + 15
3                                   LIVE: t0, t1
4             t2 = t0 * 3
5                                   LIVE: t1, t2
6             if t1 < t2
7                                   LIVE: t1, t2
8             then
9                                   LIVE: t1, t2
10              t3 = t1 * t2
11                                  LIVE: t1, t2, t3
12              t4 = t1 * 2
13                                  LIVE: t2, t3, t4
14            else
15                                  LIVE: t1, t2
16              t3 = t1 + t2
17                                  LIVE: t1, t2, t3
18              t4 = t1 + t3
19                                  LIVE: t2, t3, t4
20            fi
21                                  LIVE: t2, t3, t4
22            t5 = t4 - t2
23                                  LIVE: t3, t5
24            t6 = t5 + t3
```

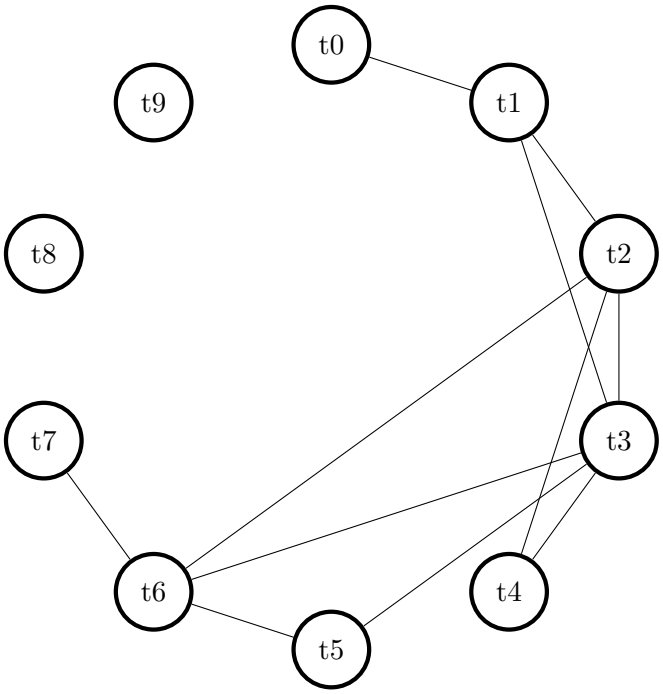
```
                                LIVE: t3, t5, t6
t2 = t5 + 1
                                LIVE: t2, t3, t6
t7 = t2 + t3
                                LIVE: t6, t7
if t6 < t7
                                LIVE: t6, t7
then
                                LIVE: t6
  t8 = t6
                                LIVE: t8
else
                                LIVE: t7
  t8 = t7
                                LIVE: t8
fi
                                LIVE: t8
t9 = t8 * 2
                                LIVE: t9
```

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

Answer: At least three registers will be required as `t1`, `t2`, and `t3` are live simultaneously.

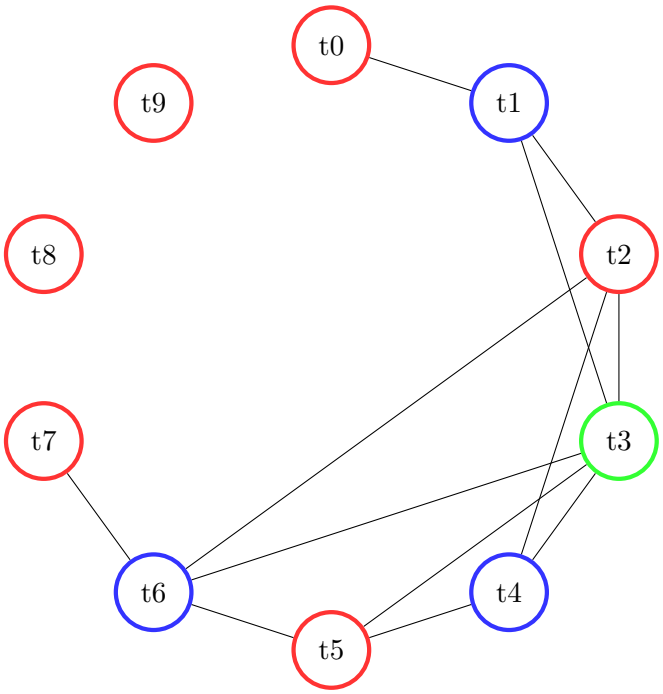
(c) Draw the register interference graph between temporaries in the above program as described in class.

Answer:



(d) Using the algorithm described in class, provide a coloring of the graph in part(c). 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.

Answer: One possible order to color the nodes: `t9, t8, t7, t6, t5, t3, t2, t4, t1, t0`



(e) Based on your coloring, write down a mapping from temporaries to registers (labeled `r1`, `r2`, etc.).

Answer:

temporary	register
t0	r1
t1	r2
t2	r1
t3	r3
t4	r2
t5	r1
t6	r2
t7	r1
t8	r1
t9	r1