CS143 Spring 2022 – Written Assignment 4 – Solutions

Tuesday, May 31, 2022 11:59 PM PDT

This assignment covers code generation, operational semantics, and optimization. You may discuss this assignment with other students and work on the problems together. However, your write-up should be your own individual work, and you should indicate in your submission who you worked with, if applicable. Assignments can be submitted electronically through Gradescope as a PDF by Tuesday, May 31, 2022 at 11:59 PM PDT. A LaTeX template for writing your solutions is available on the course website.

1. Consider the following program in Cool, representing a "slightly" over-engineered implementation which calculates the sum of $\{0, 1, 2\}$ using an operator class and a reduce() method:

```
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
42      get_tail(): IntList { tail };
43
44      set_tail(t: IntList): IntList {
45          tail <- t
46      };
47
48      generate(n: Int): IntList {
49          let l: IntList <- New IntList in {
50              l.set_head(n);
51              -- Point A
52              if (n = 1) then
53                  l.set_tail(empty_tail)
54              else
55                  l.set_tail(generate(n-1))
56              fi;
57              l;
58          }
59      };
60  };
61
62  class Main {
63      reduce(result: Int, op: BinOp, l: IntList): Int {{
64          result <- op.operate(result,l.get_head());
65          if (l.tail_is_empty() = true) then
66              -- Point B
67              result
68          else
69              reduce(result,op,l.get_tail())
70          fi;
71      }};
72
73      main(): Object {
74          let op: BinOp <- new SumOp, l: IntList <- new IntList,
75              io: IO <- new IO in {
76              l <- l.generate(2);
77              io.out_int(self.reduce(0,op,l));
78          }
79      };
80  };
```

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:

$\texttt{maddr}_1$:	`cgen_m(BinOp.optype)`
$\texttt{maddr}_2$:	`cgen_m(BinOp.operate)`
$\texttt{maddr}_3$:	`cgen_m(SumOp.optype)`
$\texttt{maddr}_4$:	`cgen_m(MulOp.optype)`
$\texttt{maddr}_5$:	`cgen_m(MulOp.operate)`
$\texttt{maddr}_6$:	`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]`

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

↓

↑

Heap *(low 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) 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. Assume that the Object class has no methods.

Solution:

BinOp:

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

SumOp:

Method Idx	Method Name	Address
0	`optype`	$\texttt{maddr}_{3}$
1	`operate`	$\texttt{maddr}_{2}$

MulOp:

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

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`	$\texttt{maddr}_{11}$

(b) Consider the state of the program at runtime when reaching (for the first time) 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.

Solution:

Main (on top-level):

5
3
$\texttt{maddr}_{18}$

SumOp (in Main.main):

2
3
$\texttt{maddr}_{15}$

IntList (in Main.main):

4
6
$\texttt{maddr}_{17}$
0
void
void

IntList (in IntList.generate):

4
6
$\texttt{maddr}_{17}$
2
void
void

(c) The following table represents an abstract view of the layout of the stack at runtime when reaching (for the first time) 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`	2	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

Assume that the activation record is set as in Lecture 12 (e.g., slide 20). 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.

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

Solution:

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`	l	arg_3
$\texttt{maddr}_{19} - 28$	`Main.reduce`	op	arg_2
$\texttt{maddr}_{19} - 32$	`Main.reduce`	2	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`	l' (tail of l)	arg_3
$\texttt{maddr}_{19} - 52$	`Main.reduce`	op	arg_2
$\texttt{maddr}_{19} - 56$	`Main.reduce`	3	arg_1
$\texttt{maddr}_{19} - 60$	`Main.reduce`	self	arg_0
$\texttt{maddr}_{19} - 64$	`Main.reduce`	$\texttt{maddr}_{12} + \delta_2$	Return

2. Consider the following arithmetic expression: $(1 + (2 * 3 - 8/4)) + (5 - 6) * 7$.

   (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.

   Solution: 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).

```
1      acc <- 1
2      push acc // 1 push
3      acc <- 2
4      push acc // 2 push
5      acc <- 3
6      acc <- mul 0(sp) acc => 2*3 => 6 // 1 pop
7      push acc // 3 push
8      acc <- 8
9      push acc // 4 push
10     acc <- 4
11     acc <- div 0(sp) acc => 8/4 => 2 // 2 pop
12     acc <- sub 0(sp) acc => 6-2 => 4 // 3 pop
13     acc <- add 0(sp) acc => 1+4 => 5 // 4 pop
14     push acc // 5 push
15     acc <- 5
16     push acc // 6 push
17     acc <- 6
18     acc <- sub 0(sp) acc => 5-6 => -1 // 5 pop
19     push acc // 7 push
20     acc <- 7
21     acc <- mul 0(sp) acc => (-1)*7 => -7 // 6 pop
22     acc <- add 0(sp) acc => 5+(-7) => -2 // 7 pop
```

   Only the numbers of pushes and pops are required to correctly answer this question.

(b) 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.

Solution: 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. Note that we never need to load values from memory in a different order than the one used to store them.

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

Only the numbers of loads and stores are required to correctly answer this question. Note that we accept 2 loads and 2 stores as another answer, which can be obtained by taking a different order of loading constants to registers, since the order was a bit unclear in the problem description.

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: e_1 is evaluated first, e_2 is evaluated next, and then the body of the loop (e_3) is executed once for every integer in the range $[e_1, e_2]$ (inclusive) in increasing order. Here e_1 and e_2 should be evaluated only once. Similar to the while loop, the for-loop returns void.

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

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

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

$$\frac{\begin{array}{c} so, S, E \vdash e_1 : \text{Int}(n_1), S_1 \\ so, S_1, E \vdash e_2 : \text{Int}(n_2), S_2 \\ n_1 \leq n_2 \\ so, S_2, E \vdash e_3 : v_3, S_3 \\ so, S_3, E \vdash \text{for } n_1 + 1 \text{ to } n_2 \text{ do } e_3 \text{ rof} : \text{void}, S_4 \end{array}}{so, S, E \vdash \text{for } e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof} : \text{void}, S_4} \text{ ForTrue}$$

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

(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.

Assume that cgen(e_1) (and similarly cgen(e_2)) does integer unboxing, i.e., the evaluation result of e_1 will be stored in `$a0` after executing cgen(e_1). You can use the instruction `ble r1, r2, label` in order to branch to `label` if `r1` $\leq$ `r2`.

Solution: There are multiple possible solutions here. One possible solution is as follows:

```
1     cgen(for e1 to e2 do e3 rof) =
2         cgen(e1)                # compute lower bound (store in a0)
3         sw $a0, 0($sp)          # push a0 onto stack
4         addiu $sp, $sp, -4
5         cgen(e2)                # compute upper bound (store in a0)
6         sw $a0, 0($sp)          # push a0 onto stack
7         addiu $sp, $sp, -4
8         b compare               # jump to the end of the loop where
9                                 # comparison is performed
10      loop:
11        addiu $t1, $t1, 1       # increment counter
12        sw $t1, 8($sp)          # save counter back to stack
13        cgen(e3)                # execute loop iteration
14      compare:
15        lw $a0, 4($sp)          # load a0 with upper bound
16        lw $t1, 8($sp)          # load t1 with lower bound
17        ble $t1, $a0, loop      # repeat loop if within bounds
18        li $a0, 0               # load a0 with void (i.e., 0)
19        addiu $sp, $sp, 8       # pop the stack
```

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

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

(a) Assume that the only variable that is live at the exit of this block is `z`, while `f` and `g` 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. Assume that `+` is faster than `*` and `<<`.

i. Algebraic simplification

ii. Copy propagation

iii. Common sub-expression elimination

iv. Copy propagation

v. Dead code elimination

Solution:

i. Algebraic simplification

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

```
1       a := f
2       b := f * g
3       c := b + b
4       d := a - 3
5       x := a * g
6       y := x + x
7       z := y * c
```

ii. Copy propagation

```
1       a := f
2       b := f * g
3       c := b + b
4       d := a - 3
5       x := a * g
6       y := x + x
7       z := y * c
```

```
1       a := f
2       b := f * g
3       c := b + b
4       d := f - 3
5       x := f * g
6       y := x + x
7       z := y * c
```

iii. Common sub-expression elimination

```
1       a := f
2       b := f * g
3       c := b + b
4       d := f - 3
5       x := f * g
6       y := x + x
7       z := y * c
```

```
1       a := f
2       b := f * g
3       c := b + b
4       d := f - 3
5       x := b
6       y := x + x
7       z := y * c
```

iv. Copy propagation

```
1       a := f                          1       a := f
2       b := f * g                      2       b := f * g
3       c := b + b                      3       c := b + b
4       d := f - 3                      4       d := f - 3
5       x := b                          5       x := b
6       y := x + x                      6       y := b + b
7       z := y * c                      7       z := y * c
```

v. Dead code elimination

```
1       a := f                          1
2       b := f * g                      2       b := f * g
3       c := b + b                      3       c := b + b
4       d := f - 3                      4
5       x := b                          5
6       y := b + b                      6       y := b + b
7       z := y * c                      7       z := y * c
```

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

i. Common sub-expression elimination

```
1       b := f * g                      1       b := f * g
2       c := b + b                      2       c := b + b
3       y := b + b                      3       y := c
4       z := y * c                      4       z := y * c
```

ii. Copy propagation:

```
1       b := f * g                      1       b := f * g
2       c := b + b                      2       c := b + b
3       y := c                          3       y := c
4       z := y * c                      4       z := c * c
```

iii. Dead code elimination:

```
1       b := f * g                      1       b := f * g
2       c := b + b                      2       c := b + b
3       y := c                          3
4       z := c * c                      4       z := c * c
```

No other optimizations covered during the lecture apply at this point.

5. Consider the following assembly-like pseudo-code, using 11 temporaries (abstract registers) `t0` to `t10`:

```
1       t1 := t0 * t0
2       t2 := -t1
3       t3 := t2 + t0
4       if t3 > 0:
5           t4 := t0 - t3
6           t5 := t2 * 3
7           t6 := t4 - t5
8       else:
9           t7 := t0 + t3
10          t8 := t2 * 3
11          t6 := t7 + t8
12      t9 := t0 * t3
13      t10 := t6 + t9
```

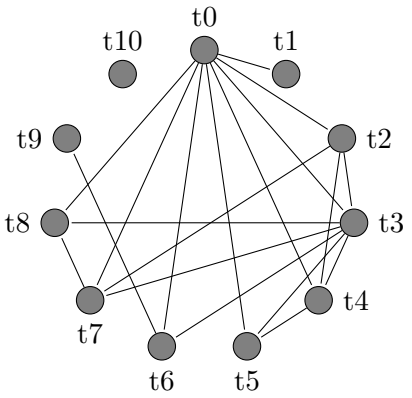
(a) At each program point, list the variables that are live. Note that `t0` is the only input for the given code and `t10` is the only live value on exit.

Solution:

```
1                           # Live: t0 (input)
2       t1 := t0 * t0
3                           # Live: t0, t1
4       t2 := -t1
5                           # Live: t0, t2
6       t3 := t2 + t0
7                           # Live: t0, t2, t3
8       if t3 > 0:
9                           # Live: t0, t2, t3
10          t4 := t0 - t3
11                          # Live: t0, t2, t3, t4
12          t5 := t2 * 3
13                          # Live: t0, t3, t4, t5
14          t6 := t4 - t5
15                          # Live: t0, t3, t6
16      else:
17                          # Live: t0, t2, t3
18          t7 := t0 + t3
19                          # Live: t0, t2, t3, t7
20          t8 := t2 * 3
21                          # Live: t0, t3, t7, t8
22          t6 := t7 + t8
23                          # Live: t0, t3, t6
24      t9 := t0 * t3
25                          # Live: t6, t9
26      t10 := t6 + t9
27                          # Live: t10
```

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

Solution:

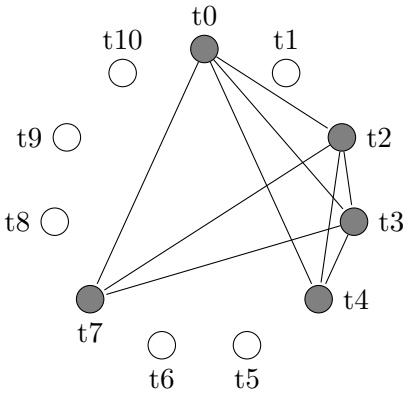


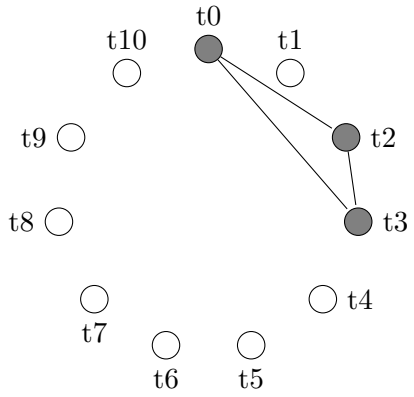
(c) Provide a lower bound on the number of registers required by the program that does not involve spilling, based on (a) or (b). Can you explain why?

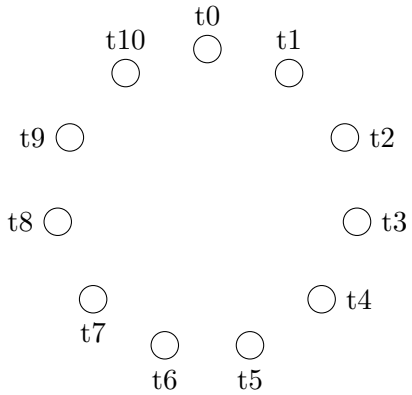
Solution: A lower bound is 4. Since there are program points at which 4 temporaries are live (e.g. t0, t2, t3 and t4), there is no way of solving this allocation problem with less than 4 registers.

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

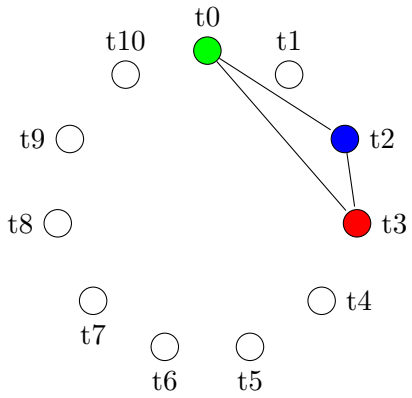
Solution: We progressively remove nodes with less than 4 neighbors in the graph, since those can always be colored once their neighbors themselves have been colored:

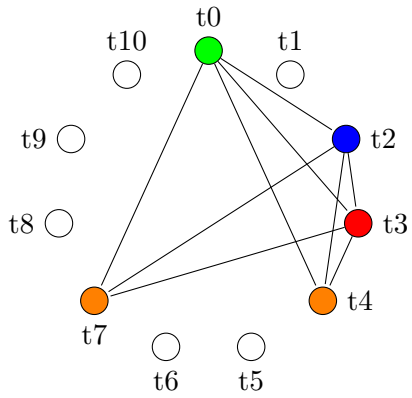


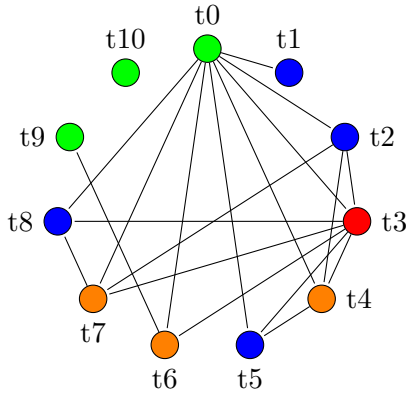




This shows that the graph is four colorable. By reintroducing the nodes in the order we removed them, we quickly arrive to a suitable coloring with 4 colors:







The above coloring is performed in the following order: `t0, t2, t3, t4, t7, t1, t5, t6, t8, t9, t10`. Note that the coloring is not unique. For example, t6 could have been assigned blue as opposed to orange.

(e) Based on your coloring, write down a mapping from temporaries to registers (labeled `r0`, `r1`, etc.).

Solution: We can assign temporaries of each color to each register: r0 (green), r1 (blue), r2 (red), r3 (orange).

```
1          t0:  r0
2          t1:  r1
3          t2:  r1
4          t3:  r2
5          t4:  r3
6          t5:  r1
7          t6:  r3
8          t7:  r3
9          t8:  r1
10         t9:  r0
11         t10: r0
```