YOURNAME - SUNETID

CS143 - Written Assignment 4

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, June 6, 2017 11:59 PM PDT. A LaTeX template for writing your solutions is available on the course website.

1. **(20 pts)** Consider the following program in Cool, representing an over-engineered implementation which calculates the factorial of 3 using an operator class and a reduce() method:

```
class BinOp {
    operate(a: Int, b: Int): Int {
        a + b
    };
    optype(): String {
        "BinOp"
    };
};
class SumOp inherits BinOp {
    optype(): String {
        "SumOp"
    };
};
class MulOp inherits BinOp {
    operate(a: Int, b: Int): Int {
        a * b
    };
    optype(): String {
        "MulOp"
    };
};
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 {
            -- Point A
            l.set_head(n);
```

```
            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
            -- Point B
            result
        else
            reduce(result,op,l.get_tail())
        fi;
    }};
    main(): Object {
        let op: BinOp <- New MulOp, l: IntList <- New IntList, io:
            IO <- New IO in {
            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):

Code segment:

$\texttt{maddr}_1$:	`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]`

Stack ($\texttt{maddr}_{19}$)

$\downarrow$

$\uparrow$

Heap

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.

(a) **(4 pts)** 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. In particular, assume that the caller is responsible for saving and 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 and restored (2 uses). For 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). There are thus $5 * 2 + 1 + 8 + 1 = 20$ uses of the frame pointer.

(b) **(4 pts)** 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) **(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, and that attributes are laid out in the same order as the class definition.

Answer:

Main

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

MulOp

3
3
$\texttt{maddr}_{16}$

IntList (in Main.main)

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

IntList (in IntList.generate)

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

(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".

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	$\texttt{maddr}_{19}$	FP
$\texttt{maddr}_{19} + 24$	IntList.generate	self	arg_0
$\texttt{maddr}_{19} + 28$	IntList.generate	3	arg_1
$\texttt{maddr}_{19} + 32$	IntList.generate	$\texttt{maddr}_{13} + \delta$	Return
$\texttt{maddr}_{19} + 36$	IntList.generate	l	local

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".

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 $\texttt{maddr}_i + \delta$, 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.

Answer:

(See next page)

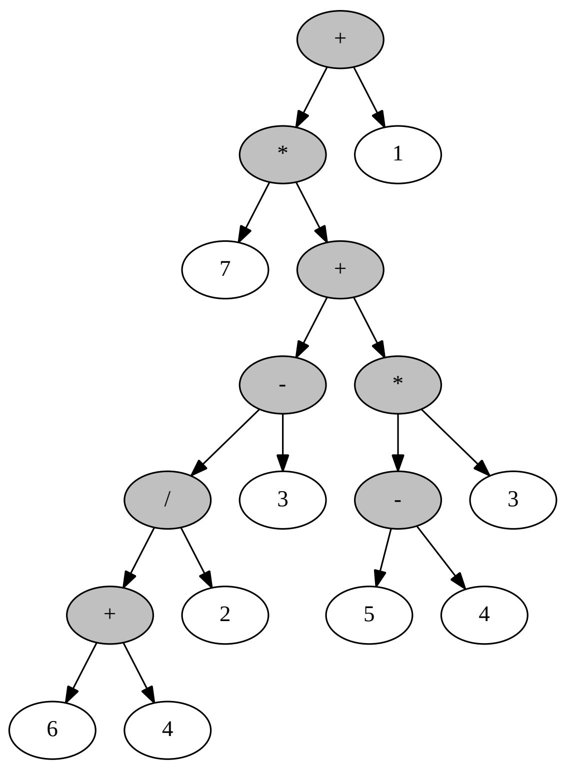
Address	Method	Contents	Description
$\mathtt{maddr}_{19}$	Main.main	self	arg_0
$\mathtt{maddr}_{19} + 4$	Main.main	...	Return
$\mathtt{maddr}_{19} + 8$	Main.main	op	local
$\mathtt{maddr}_{19} + 12$	Main.main	l	local
$\mathtt{maddr}_{19} + 16$	Main.main	io	local
$\mathtt{maddr}_{19} + 20$	Main.reduce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

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

   (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 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 on the stack. After computing the value of the right child, the previously computed value is popped. Therefore there will be 8 pushes and 8 pops in total.



(b) **(2 pts)** Now suppose that you have access to two registers r1, 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 $r2$ as a temporary. However, the left subtree only uses $r2$ to store the value of 7 while its sibling, the "+" subtree, is evaluated. As a result, we can replace a total of two loads and stores from the stack with register-register moves, and therefore 6 loads and stores are needed.

We also accept the following solution, which is more efficient than the one derived from the lecture notes. Instead of using the temporaries greedily, we can store 7 on the stack while computing its sibling "+" subtree. After the left child of the "+" subtree is computed, we store that value on the stack too. Every other temporary can be saved in the register $r2$. Here we only need 2 loads and 2 stores.

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 integer in the range $[e_1, e_2]$ (inclusive) in order. Similar to the while loop, the for-loop returns void. Also assume that *id* is not in the environment and hence cannot be modified in the loop body.

(a) **(10 pts)** Give the operational semantics for the for-loop construct above.
Answer: There are multiple ways to solve this problem; here we present two solutions.
Solution 1: Reduce to semantics of a while loop.

$$\frac{\begin{array}{c} so, S, E \vdash e_1 \mapsto Int(n_1), S_1 \\ so, S_1, E \vdash e_2 \mapsto Int(n_2), S_2 \\ addr = newloc(S_2) \\ E' = E[addr/\text{id}] \qquad S_3 = S_2[Int(n_1)/addr] \\ so, S_3, E' \vdash \text{while id} <= n_2 \text{ loop } \{e_3;\ \text{id} \leftarrow \text{id} + 1;\ \} \text{ pool} \mapsto v, S_4 \end{array}}{so, S, E \vdash \text{for id : Int } \leftarrow e_1 \text{ to } e_2 \text{ do } e_3 \text{ rof} \mapsto void, S_4}$$

Solution 2: Recursive technique.

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

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

(b) **(6 pts)** Give the code generation function cgen(for id : Int $\leftarrow e_1$ 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,

```
    cgen(e1)              # compute lower bound (store in a0)
    sw $a0 0($sp)         # push a0 onto stack
    addiu $sp, $sp, -4
    cgen(e2)              # compute upper bound (store in a0)
    sw $a0 0($sp)         # push a0 onto the stack
    addiu $sp, $sp, -4
    j compare             # jump to the end of the loop where
                          # comparison is performed
  loop:
    addiu $t0, $t0, 1     # increment counter
    sw $t0, 8($sp)        # save counter back to stack
    cgen(e3)              # execute loop iteration
  compare:
    lw $a0, 4($sp)        # load a0 with upper bound
    lw $t0, 8($sp)        # load t0 with lower bound
    ble $t0, $a0, loop    # repeat loop if within bounds
    addiu $sp, $sp, 8     # pop the stack
    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. **(8 pts)** Consider the following basic block, in which all variables are integers.

```
a := f * f
b := 1 * f + 0
c := 7 * 7
d := b + c
e := f * f
g := a + d
x := e + d
k := b * f
h := g * x
y := h + k
```

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.

(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

```
a := f * f
b := f
c := 7 * 7
d := b + c
e := f * f
g := a + d
x := e + d
k := b * f
h := g * x
y := h + k
```

(b) Copy propogation

```
a := f * f
b := f
c := 7 * 7
d := f + c
```

```
e := f * f
g := a + d
x := e + d
k := f * f
h := g * x
y := h + k
```

(c) Common subexpression elimination

```
a := f * f
b := f
c := 7 * 7
d := f + c
e := a
g := a + d
x := e + d
k := a
h := g * x
y := h + k
```

(d) Constant folding

```
a := f * f
b := f
c := 49
d := f + c
e := a
g := a + d
x := e + d
k := a
h := g * x
y := h + k
```

(e) Copy Propogation

```
a := f * f
b := f
c := 49
d := f + 49
e := a
g := a + d
x := a + d
k := a
h := g * x
y := h + a
```

(f) Dead Code Elimination

```
a := f * f
d := f + 49
g := a + d
x := a + d
```

```
h := g * x
y := h + a
```

Finally we can perform common subexpression elimination to attain a slightly more optimal program. However, since x is a live-out value, there is little more we can do. (Were x not live-out, we could benefit from copy propagation and dead code elimination as well).

(g) Optimized Code

```
a := f * f
d := f + 49
g := a + d
x := g
h := g * x
y := h + a
```

5. **(12 pts)** Consider the following assembly-like pseudo-code, using 10 temporaries (abstract registers) $t0$ to $t9$:

```
t1 = 2 * t0
t2 = 10 + t0
if t1 < t2
  t3 = t1 + t2
  t4 = t1 + t3
else
  t4 = t1 + t2
fi
t5 = t4 + t1
t6 = t4 - t1
t7 = t4 * t4
t8 = t5
t8 = t6 + t8
t8 = t7 + t8
if t6 < t8
  t9 = t6
else
  t9 = t8
fi
```

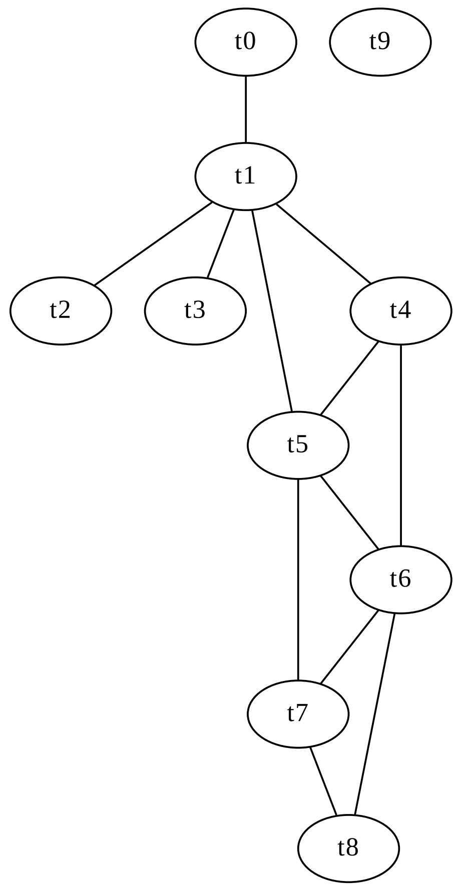
(a) At each program point, list the temporaries that are live. Note that $t0$ is the only input temporary for this code and $t9$ will be the only live value on exit.
Answer:
See next page.

```
                                  LIVE: t0
t1 = 2 * t0
                                  LIVE: t0, t1
t2 = 10 + t0
                                  LIVE: t1, t2
if t1 < t2
                                  LIVE: t1, t2
  t3 = t1 + t2
                                  LIVE: t1, t3
  t4 = t1 + t3
                                  LIVE: t1, t4
else
                                  LIVE: t1, t2
  t4 = t1 + t2
                                  LIVE: t1, t4
fi
                                  LIVE: t1, t4
t5 = t4 + t1
                                  LIVE: t1, t4, t5
t6 = t4 - t1
                                  LIVE: t4, t5, t6
t7 = t4 * t4
                                  LIVE: t5, t6, t7
t8 = t5
                                  LIVE: t6, t7, t8
t8 = t6 + t8
                                  LIVE: t6, t7, t8
t8 = t7 + t8
                                  LIVE: t6, t8
if t6 < t8
                                  LIVE: t6
  t9 = t6
                                  LIVE: t9
else
                                  LIVE: t8
  t9 = t8
                                  LIVE: t9
fi
                                  LIVE: t9
```

(b) Provide a lower bound on the number of registers required by the program.
Answer: At least three registers must be required as t1, t4 and t5 are live simultaneously.

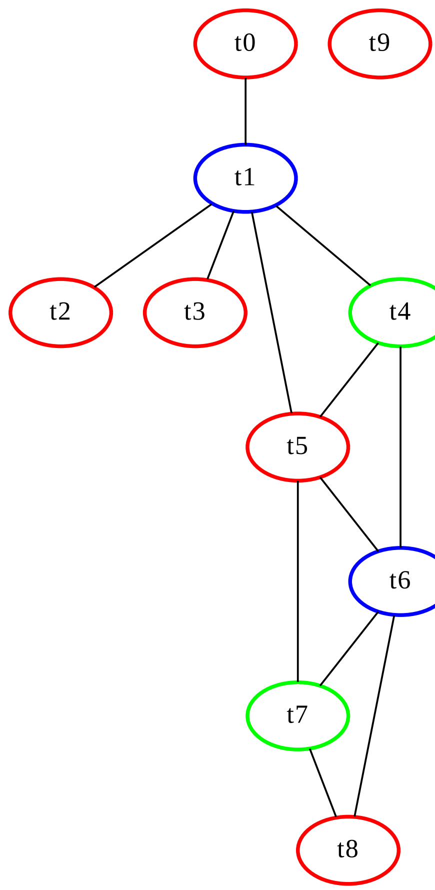
(c) Draw the interference graph between temporaries in the above program as described in class.

Answer:



(d) Using the algorithm described in class, provide a coloring of the above graph. 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: 8, 7, 6, 5, 4, 1, 3, 2, 0, 9



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

Answer:

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