CS143 Spring 2023 – Written Assignment 4 – 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:

```
class BinOp {
    optype(): String {
        "BinOp"
    };

    operate(a: Int, b: Int): Int {
        a + b
    };
};

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
    };

    -- Create a list [n, n-1, ..., 2, 1] for n ≥ 1
```

```
    generate(n: Int): IntList {
        let l: IntList <- New IntList in {
            l.set_head(n); -- Point A
            if (n = 1) then
                l.set_tail(empty_tail)
            else
                l.set_tail(generate(n-1))
            fi;
            l;
        }
    };
};

class Main {
    reduce(result: Int, op: BinOp, l: IntList): Int {{
        result <- op.operate(result,l.get_head());
        if (l.tail_is_empty()) then
            result -- Point B
        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.optype)`
$\texttt{maddr}_2$:	`cgen_m(BinOp.operate)`
$\texttt{maddr}_3$:	`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 ($\texttt{maddr}_{19}$)

↓

↑

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) 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 and IntList. You can ignore built-in methods like abort, type_name, and copy.

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`	$\texttt{maddr}_{5}$
1	`operate`	$\texttt{maddr}_{4}$

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 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:

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

or

4
6
$\texttt{maddr}_{17}$
3
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 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. 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.

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".

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

Notice that in the last three stack frames, we do not have a separate entry for “result”, since it is a formal parameter and not a local variable.

2. Consider the following assembly language used to program a stack machine (**r**, **r1**, and **r2** denote arbitrary registers):

   - **push** r: copies the value of r and pushes it onto the stack.
   - **top** r: copies the value at the top of the stack into r. This command does not modify the stack.
   - **pop**: discards the value at the top of the stack.
   - **swap**: swaps the value at top of the stack with the value right beneath it. E.g. if the stack was $\langle$ \$, .., 5, 2 $\rangle$ swap would change the stack to be $\langle$ \$, ..., 2, 5 $\rangle$
   - $r1+=r2$: adds $r1$ and $r2$ and saves the result in $r1$. $r1$ may be the same as $r2$.
   - $r1-=r2$: subtracts $r2$ from $r1$ and saves the result in $r1$. $r1$ may be the same as $r2$.
   - **clamp** r: sets r to 0 if r is negative otherwise leaves r unchanged
   - **jump** r: jumps to the address in r and resumes execution.
   - **ite** $r1$ $r2$ $r3$: if $r1$ is not equal to zero then jumps to the address in $r2$ else jumps to the address in $r3$.
   - **loadconst** r int: loads a constant int into r
   - **loadlabel** r label: loads the address of a labeled code segment into r.

   Provide a code generation function for each the of these instructions, except `loadlabel`, targeting MIPS. Assume that registers used in the stack language are valid MIPS registers. Use `$sp` to hold a pointer to the top of the stack and a single temporary register `$at` which is guaranteed to not appear in the stack language.

```
cgen(push r) =
        sw r 0($sp)
        addiu $sp $sp -4

cgen(top r) =
        lw r 4($sp)

cgen(pop) =
        addiu $sp $sp 4

cgen(swap) =
        lw $at 4($sp) // get the top of the stack
        sw $at 0($sp) // duplicate it
        lw $at 8($sp) // get the item below it
        sw $at 4($sp) // store it on top of the stack
        lw $at 0($sp) // get the duplicated item
        sw $at 8($sp) // store it in the second slot

cgen(r1 += r2) =
        addu r1 r1 r2

cgen(r1 -= r2) =
        subu r1 r1 r2

cgen(clamp r) =
        bltz r  neg
        j done
    neg:
        li r 0
    done:

cgen(jump r) =
        jr r

cgen(ite r1 r2 r3) =
        li $at 0
        beq $at r1 false_branch
        jr r2
    false_branch:
        jr r3

cgen(loadconst r int) =
        li r int
```

3. 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 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) 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):

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

Then the general case where we do enter the loop:

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

Where $\mathit{Int}(i_k)$ 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) 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.

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`.

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.

```
cgen(for id: Int <- e1 to e2 do e_3 rof, nt) =
    cgen(e1, nt)
    sw $a0 i($fp)            # Save id=val(e1) on i-th var
    cgen(e2, nt+4)
    sw $a0 4($sp)            # Save v2=val(e2) on stack
    addiu $sp $sp -4
  for_loop:
    lw $a0 i($fp)            # current value of id
    lw $t1 4($sp)            # value of e2
    bgt $a0 $t1 for_exit     # Exit if id > v2
    cgen(e3, nt)             # Might change id and clobber $a0
    lw $a0 i($fp)            # Load id again
    addiu $a0 $a0 1          # Increment it by 1
    sw $a0 i($fp)            # Save it back
    b for_loop               # Jump back to the loop check
  for_exit:
    addiu $sp $sp 4          # Remove value of e2 from stack
```

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

```
x := 0 * 5
y := a + b
z := x * x
c := y * x
x := x + 4
e := c - x
x := e * x
f := a + b
y := y + f
```

(a) Assume that the only variables that are live at the exit of this block are x and y, while a and b are given as inputs. In order, apply the following optimizations to this basic block. Show the result of each transformation. For each optimization, you must continue to apply it until no further applications of that transformation are possible (if any were), before writing out the result and moving on to the next.

i. Algebraic simplification
ii. Common sub-expression elimination
iii. Copy propagation / Constant propagation
iv. Algebraic simplification
v. Dead code elimination

Solution:

i.

```
x := 0
y := a + b
z := x * x
c := y * x
x := x + 4
e := c - x
x := e * x
f := a + b
y := y + f
```

ii.

```
x := 0
y := a + b
z := x * x
c := y * x
x := x + 4
e := c - x
x := e * x
f := y
y := y + f
```

iii

```
x := 0
y := a + b
z := 0 * 0
c := y * 0
```

```
x := 0 + 4
e := c - x
x := e * x
f := y
y := y + y
```

iv

```
x := 0
y := a + b
z := 0
c := 0
x := 4
e := c - x
x := e * x
f := y
y := y << 1
```

v

```
y := a + b
c := 0
x := 4
e := c - x
x := e * x
y := y << 1
```

(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: You can apply copy propagation, algebraic simplification, and dead code elimination to the output of the previous part to get the following:

```
y := a + b
x := -16
y := y << 1
```

5. Consider the following assembly-like pseudo-code, using 6 temporaries (abstract registers) a to f:

```
a := b + d
b := a + d
d := a - b
c := d + d
if c > 100:
    c := c + d
else:
    d := 1
e := d - c
f := e - c
```

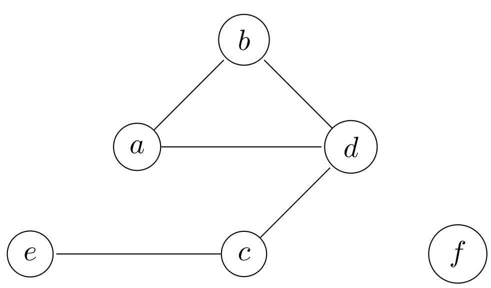
(a) At each program point, list the variables that are live. Note that **b** and **d** are inputs for the given code and **f** is a live value on exit.

Solution:

```
                          {b,d}
        a := b + d        {a,d}
        b := a + d        {a,b}
        d := a - b        {d}
        c := d + d        {c,d}
        if c > 100:
                          {c,d}
            c := c + d    {c,d}
        else:
                          {c}
            d := 1        {c,d}
        e := d - c        {c,e}
        f := e - c        {f}
```

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

Solution:

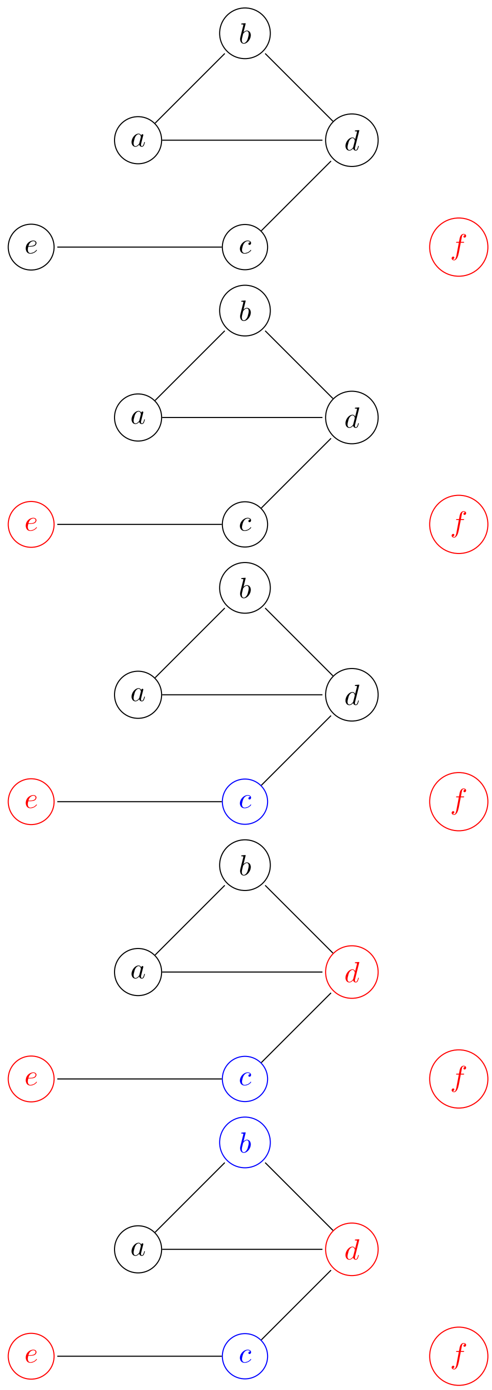


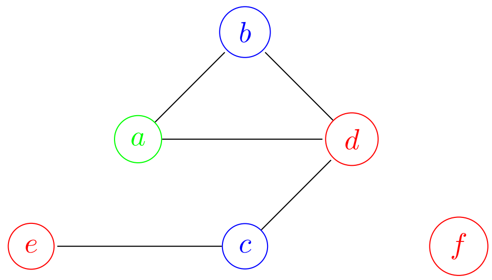
(c) Provide a lower bound on the number of registers required by the program induced from the interference graph. Can you explain why?

Solution: 3. It is mainly because there is a triangle in the graph.

(d) Using the algorithm described in class, provide a coloring of the graph in part(b). The number of colors used should be your lower bound in part (c). Provide the final k-colored graph (you may use the tikz package to typeset it or simply embed an image), along with the order in which the algorithm colors the nodes.

Solution:





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

Solution:

```
a: r1
b: r2
c: r2
d: r3
e: r3
f: r3
```