# Solutions to Written Assignment 4

1. Suppose that we want to add the following conditional expression to Cool.

   `cond <p1> => <e1>; <p2> => <e2>; ... ; <pn> => <en>; dnoc`

   There must be at least one predicate and expression pair (that is, $n \geq 1$). The evaluation of a `cond` expression begins with the evaluation of the predicate `<p1>`, which must have static type `Bool`. If `<p1>` evaluates to `true`, then `<e1>` is evaluated, and the evaluation of the `cond` expression is complete. If `<p1>` evaluates to `false`, then `<p2>` is evaluated, and this process is repeated until one of the predicates evaluates to `true`. The value of the `cond` expression is the value of the expression `<ei>` corresponding to the first predicate `<pi>` that evaluates to `true`. If all the predicates evaluate to `false`, then the value of the `cond` expression is `void`.

   Write operational semantics rules for this conditional expression in Cool.

   The first rule applies to `cond` expressions with a single predicate and expression pair, and handles the case that all of the predicates in a conditional expression evaluate to `false`.

$$\frac{so, S_1, E \vdash p : Bool(false), S_2}{so, S_1, E \vdash \text{cond } p \Rightarrow e;\ \text{dnoc} : void, S_2} \qquad \text{[Cond-Single-False]}$$

   The other rules handle the case that the first predicate evaluates to `true`, and the case that the first predicate in a `cond` expression with multiple predicate and expression pairs evaluates to `false`.

$$\frac{\begin{array}{c} so, S_1, E \vdash p_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_1 : v_1, S_3 \end{array}}{so, S_1, E \vdash \text{cond } p_1 \Rightarrow e_1; \ldots; p_n \Rightarrow e_n;\ \text{dnoc} : v_1, S_3} \qquad \text{[Cond-True]}$$

$$\frac{\begin{array}{c} so, S_1, E \vdash p_1 : Bool(false), S_2 \\ so, S_2, E \vdash \text{cond } p_2 \Rightarrow e_2; \ldots; p_n \Rightarrow e_n;\ \text{dnoc} : v, S_3 \end{array}}{so, S_1, E \vdash \text{cond } p_1 \Rightarrow e_1; p_2 \Rightarrow e_2; \ldots; p_n \Rightarrow e_n;\ \text{dnoc} : v, S_3} \qquad \text{[Cond-Multiple-False]}$$

2. Write a code generation function `cgen(cond <p1> => <e1>; ... ; <pn> => <en>; dnoc)` for the conditional expression described in Question 1. For concreteness, assume that $n = 2$. Use the stack machine architecture and conventions from the lectures.

```
cgen(cond p1 => e1; p2 => e2; dnoc) =
    cgen(p1)          // now acc ($a0) has a pointer to the value of p1
    lw    $a0 12($a0) // read the value attribute of the Bool
    beq   $a0 0 PRED2 // go to second predicate if value is false
    cgen(e1)          // now acc ($a0) has a pointer to the value of e1
    b DONE            // evaluation of cond is complete
label PRED2:
    cgen(p2)          // now acc ($a0) has a pointer to the value of p2
    lw    $a0 12($a0) // read the value attribute of the Bool
    beq   $a0 0 VOID  // value is void if all predicates are false
    cgen(e2)          // now acc ($a0) has a pointer to the value of e2
    b DONE            // evaluation of cond is complete
label VOID:
    li    $a0 0       // put void into the accumulator
label DONE:
```

3. Consider the following basic block, in which all variables are integers, and ** denotes exponentiation.

```
a := b + c
z := a ** 2
x := 0 * b
y := b + c
w := y * y
u := x + 3
v := u + w
```

Assume that the only variables that are live at the exit of this block are v and z. In order, apply the following optimizations to this basic block. Show the result of each transformation.

(a) algebraic simplification

(b) common sub-expression elimination

(c) copy propagation

(d) constant folding

(e) dead code elimination

| Original code: | (a) Result of algebraic simplification |
|---|---|
| `a := b + c`<br>`z := a ** 2`<br>`x := 0 * b`<br>`y := b + c`<br>`w := y * y`<br>`u := x + 3`<br>`v := u + w` | `a := b + c`<br>`z := a * a`<br>`x := 0`<br>`y := b + c`<br>`w := y * y`<br>`u := x + 3`<br>`v := u + w` |

(b) Result of common sub-expression elimination

```
a := b + c
z := a * a
x := 0
y := a
w := y * y
u := x + 3
v := u + w
```

(c) Result of copy propagation

```
a := b + c
z := a * a
x := 0
y := a
w := a * a
u := 0 + 3
v := u + w
```

(d) Result of constant folding

```
a := b + c
z := a * a
x := 0
y := a
w := a * a
u := 3
v := u + w
```

(e) Result of dead code elimination

```
a := b + c
z := a * a
w := a * a
u := 3
v := u + w
```

When you've completed part (e), the resulting program will still not be optimal. What optimizations, in what order, can you apply to optimize the result of (e) further?

Notice that when we're done with (e), the code still will assign `u` to 3 before using `u`, and will calculate `a * a` twice. We can apply another round of common sub-expression elimination, copy propagation, and dead code elimination to remove `u` and `w` completely from the basic block. The result is as follows:

```
a := b + c
z := a * a
v := 3 + z
```

This is a general feature of this style of optimization: these optimizations typically have to be repeated multiple times for optimal effect.

4. Consider the following program.

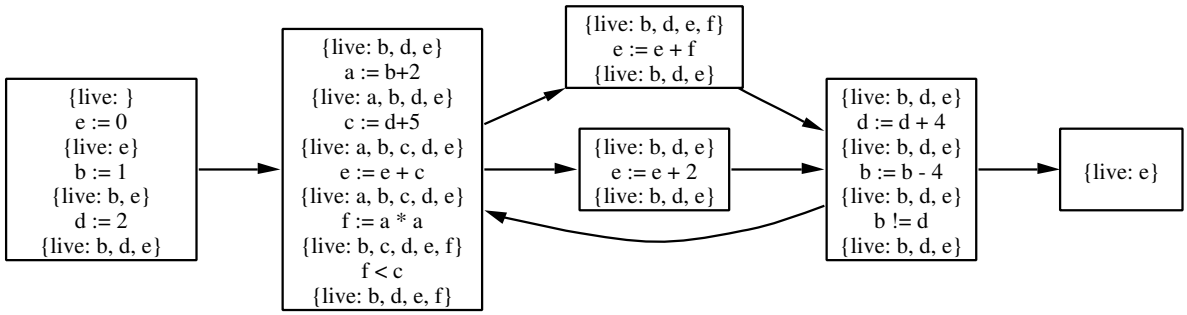
```
L0: e := 0
    b := 1
    d := 2
L1: a := b + 2
    c := d + 5
    e := e + c
    f := a * a
    if f < c goto L3
L2: e := e + f
    goto L4
L3: e := e + 2
```

```
L4: d := d + 4
    b := b - 4
    if b != d goto L1
L5:
```

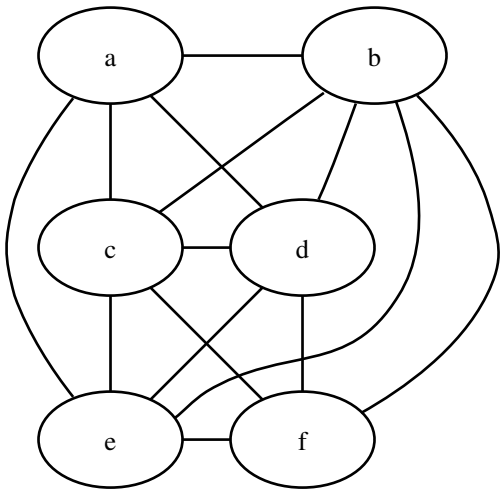
This program uses six temporaries, `a`-`f`. Assume that the only variable that is live on exit from this program is `e`.

Draw the register interference graph. (Drawing a control-flow graph and computing the sets of live variables at every program point may be helpful.)

First, we draw a control-flow graph and compute the live variable sets:




The register interference graph can then be determined based on which pairs of variables are live simultaneously.



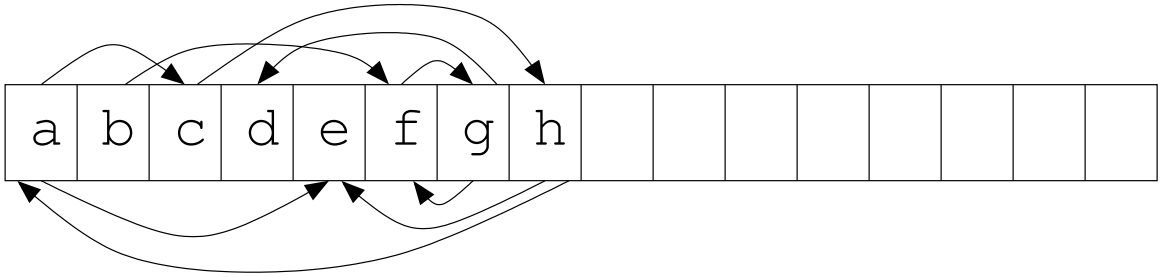

5. Suppose that the following Cool program is executed.

```
class C {
  x : C; y : C;
  setx(newx : C) : C { x <- newx };
  sety(newy : C) : C { y <- newy };
```

```
  setxy(newx : C, newy : C) : SELF_TYPE { { x <- newx; y <- newy; self; } }; };
class Main {
  x : C;
  main() : Object {
    let a : C <- new C, b : C <- new C, c : C <- new C, d : C <- new C,
        e : C <- new C, f : C <- new C, g : C <- new C, h : C <- new C in {
      f.sety(g); a.setxy(e, c); b.setx(f); g.setxy(f, d); c.sety(h); h.setxy(e, a);
      x <- c;
  } }; };
```

(a) Draw the heap at the end of the execution of the program, identifying objects by the names to which they are bound in the `let` expression. Assume that the root is the `Main` object created at the start of the program, and that this object is not in the heap. If the value of an attribute is `void`, don't show that attribute as a pointer on the diagram.

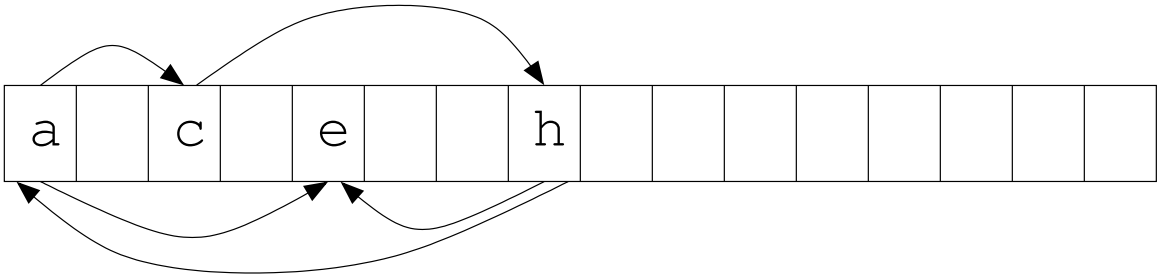
The heap is shown below. The only object pointed to by the roots is `c`.




(b) For each of the garbage collection algorithms discussed in class (Mark and Sweep, Stop and Copy, and Reference Counting), show the heap after garbage collection. When the pointers of an object are processed, assume that the processing order is the order of the attributes in the source program. Assume that the heap has space for at least 16 objects.

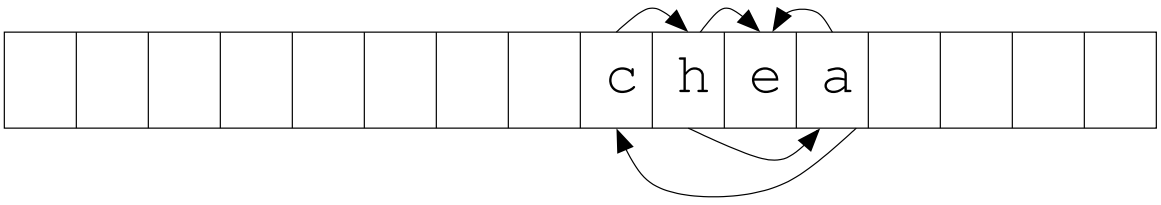
In each of the following heap diagrams, the only object pointed to by the roots is `c`.

- Mark and Sweep



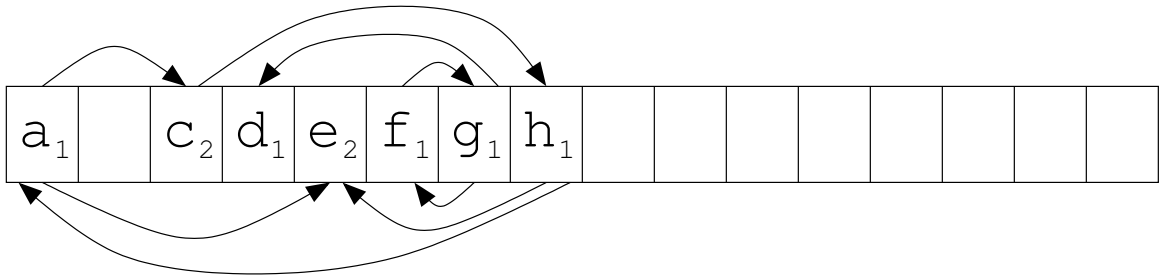

All of the slots in the heap not occupied by objects are on the free list.

- Stop and Copy



Note that object `e` appears before object `a` in the new heap because the pointers of `h` are processed in the order in which the attributes appear in the source text.

- Reference Counting




The diagram shows the reference count for each object. The pointer from the roots to the object `c` is counted as a reference for `c`.