CS131 Compilers: Writing Assignment 4
Due Saturday, June 12, 2021 at 23:59pm

Name - Numbber

This assignment asks you to prepare written answers to questions on run-time environment, object layout, operational semantics, code generation, register allocation and garbage collection. Each of the questions has a short answer. 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. Written assignments are turned in at the start of lecture. You should use the Latex template provided at the course web site to write your solution.

I worked with: ()()

Example for operational semantics rule in tex:

$$\frac{so, S_1, E \vdash e_1 : Bool(false), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_2} \qquad \text{[Loop-False]}$$

1. (10 pts) Consider the following Cool classes:

```
class A {
    a1 : Int;
    a2 : String;
    m1() : Object { ... };
    m2() : Object { ... };
};

class B inherits A {
    a3 : Int;
    m1() : Object { ... };
    m3() : Object { ... };
};

class C inherits B {
    a4 : Int;
    m2() : Object { ... };
    m3() : Object { ... };
};
```

(a) Draw a diagram that illustrates the layout of objects of type `A`, `B` and `C`, including their dispatch tables.

0: A Tag
4: 5
8: *
12: a_1
16: a_2

0: B Tag
4: 6
8: *
12: a_1
16: a_2
20: a_3

0: C Tag
4: 7
8: *
12: a_1
12: a_2
20: a_3
24: a_4

A
0: Object.abort
4: Object.type_name
8: Object.copy
12: A.m1
16: A.m2

B
0: Object.abort
4: Object.type_name
8: Object.copy
12: B.m1
16: A.m2
20: B.m3

C
0: Object.abort
4: Object.type_name
8: Object.copy
12: A.m1
16: C.m2
20: C.m3

(b) Let `obj` be a variable whose static type is `A`. Assume that `obj` is stored in register `$a0`. Write MIPS code for the function invocation `obj.2()`. You may use temporary registers such as `$t0` if you wish.

```
lw     $t0, 8($a0)
lw     $t1, 16($t0)
jalr   $t1
```

(c) Explain what happens in part (b) if `obj` has dynamic type `B`.

It will invoke correctly. We can visit the type's objects throuth the dispatch pointer at the offset 8. (b) and (c) will all visit A.m2 since B inherits from A and do not override A.m2.

2. (10 pts) Suppose you wish to add arrays to Cool using the following syntax:

`let a:T[e₁] in e₂`	Create an array a with size e_1 of T's, usable in e_2
`a[e₁] <- e₂`	Assign e_2 to element e_1 in a
`a[e]`	Get element e of a

Write the operational semantics for these three syntactic constructs. You may find it helpful to think of an array of type $T[n]$ as an object with n attributes of type T.

$$\frac{\begin{array}{l} so, S_1, E_1 \vdash e_1 : \text{Int}(n), S_2 \\ l_i = \text{newloc}(S_2) \text{ for } i = 0 \ldots n \text{ and each } l_i \text{ is distinct} \\ v_a = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ S_3 = S_2[v_a/l_0, D_T/l_1, \ldots, D_T/l_n] \\ E_2 = E_1[l_0/a] \\ so, S_3, E_2 \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E_1 \vdash \text{let } a : T[e_1] \text{ in } e_2 : v_2, S_4}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \text{Int}(m), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ E(a) = l_a \\ S_2(l_a) = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ 1 \leq m \leq n \\ S_4 = S_3[v_2/l_m] \end{array}}{so, S_1, E \vdash a[e_1] \texttt{ <- } e_2 : v_2, S_4}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : \text{Int}(m), S_2 \\ E(a) = l_a \\ S_2(l_a) = \text{array}(a_1 : l_1, \ldots, a_n : l_n) \\ 1 \leq m \leq n \\ v = S_2(l_m) \end{array}}{so, S_1, E \vdash a[e] : v, S_2}$$

3. (10 pts) The operational semantics for Cool's `while` expression show that result of evaluating such an expression is always `void`. (See page 28 of the Cool manual.)

   However, we could have used the following alternative semantics:

   - If the loop body executes at least once, the result of the `while` expression is the result from the last iteration of the loop body.
   - If the loop body never executes (i.e., the condition is false the first time it is evaluated), then the result of the `while` expression is `void`.

   For example, consider the following expression:

   ```
   while (x < 10) loop x <- x+1 pool
   ```

   The result of this expression would be 10 if `x` < 10 or `void` if `x` ≥ 10.

   Write new operational rules for the `while` construct that formalize these alternative semantics.

$$\frac{so, S_1, E \vdash e_1 : Bool(false), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : void, S_2} \quad \text{[Loop-False]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash e_1 : Bool(false), S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_2, S_4} \quad \text{[Loop-True-Last]}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Bool(true), S_2 \\ so, S_2, E \vdash e_2 : v_2, S_3 \\ so, S_3, E \vdash e_1 : Bool(true), S_4 \\ so, S_3, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_3, S_5 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : v_3, S_5} \quad \text{[Loop-True-Not-Last]}$$

4. (10 pts) Consider the following MIPS assembly code program. Using the stack-machine based code generation rules from lecture, what source program produces this code?

```
f_entry(int):
  addiu   $sp,$sp,-32
  sw      $31,28($sp)
  sw      $fp,24($sp)
  move    $fp,$sp
  sw      $4,32($fp)
  lw      $2,32($fp)
  slt     $2,$2,2
  beq     $2,$0,body
  li      $2,1
  b       end

body:
  lw      $2,32($fp)
  addiu   $2,$2,-1
  move    $4,$2
  jal     f_entry(int)
  move    $3,$2
  lw      $2,32($fp)
  mult    $3,$2
```

```
  mflo    $2
end:
  move    $sp,$fp
  lw      $31,28($sp)
  lw      $fp,24($sp)
  addiu   $sp,$sp,32
  j       $31

int f(int x){
  if (x < 2){
    return 1;
  }
  else{
    return x*f(x-1);
  }
}
```

5. (10 pts) Consider the following code in python dialect, The following steps are dependent, do the optimization until un-optimizable.

```
  p = 3
  r = 10
  s = p + r
  t = 2*r + s
  t = p
  u = p + r
  v = p + t
  y[1] = 3
  y[2] = 3
  y[3] = 3
  y[4] = 3
  for i in range(1,4):
    y[i] + v
  w = 3 + x
```

   (a) Do Dead Code Elimination.
   (b) Do Common Subexpression Elimination.
   (c) Do Copy Propagation
   (d) Do Constant Folding and Constant Propagation
   (e) Do Vectorization

(a)
− t = 2*r + s
− for i in range(1,4):
− y[i] + v
(b)
− u = p + r
+ u = s
(c)
− s = p + r
+ s = 3 + 10
− t = p
+ t = 3
− v = p + t
+ v = 3 + 3
(d)
−s = 3 + 10
+ s = 13
− v = 3 + 3
+ v = 6
(e)
−y[1] = 3
−y[2] = 3
−y[3] = 3
−y[4] = 3
+ {y[0], y[1], y[2], y[3]} = {3,3,3,3}
(c)
− u = s
+ u = 13
Result:

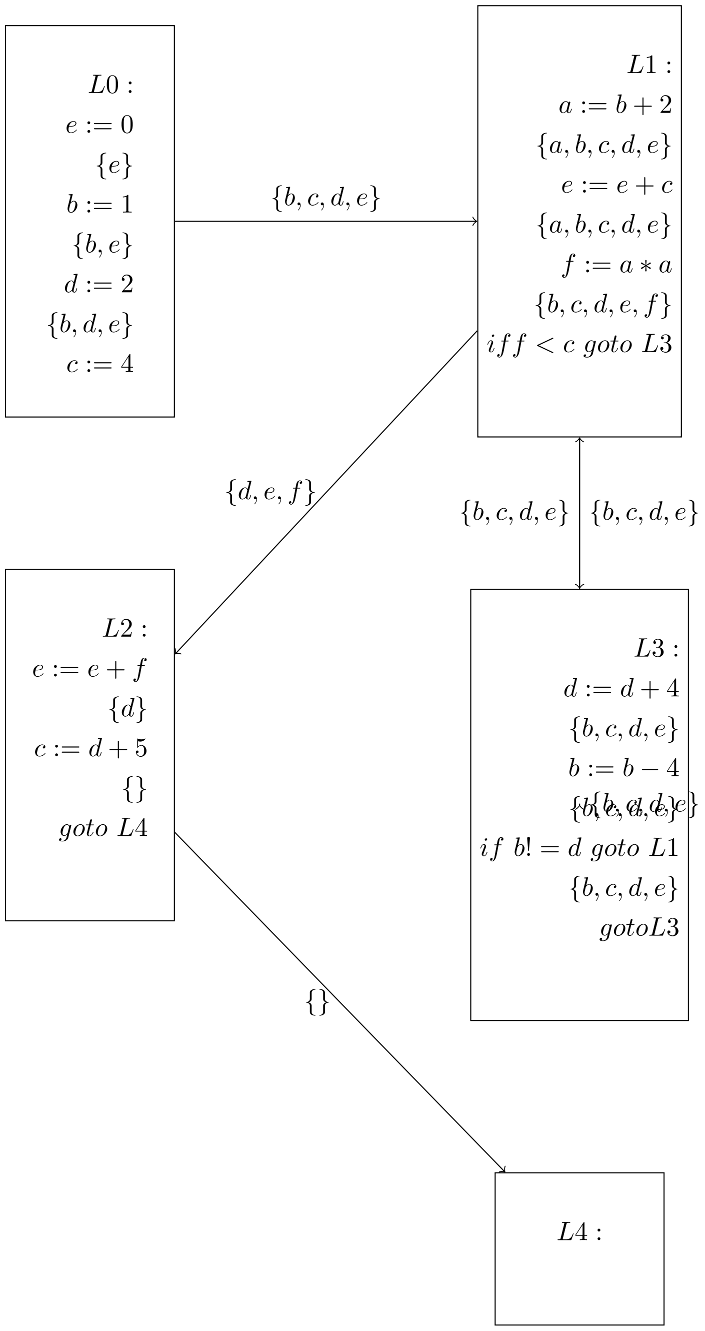
```
p = 3
r = 10
s = 13
t = 3
u = 13
v = 6
{y[0], y[1], y[2], y[3]} = {3,3,3,3}
w = 3 + x
```

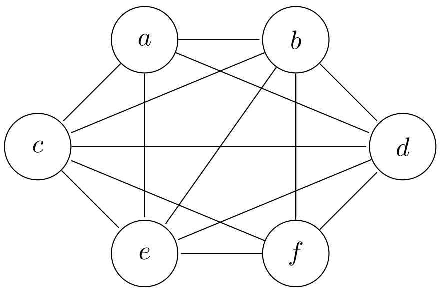
6. (2*10=20 pts) Consider the following program:

```
L0: e := 0
    b := 1;
    d := 2;
    c := 4;
L1: a := b+2
    e := e + c
    f := a * a
    if f < c goto L3
L2: e := e + f
    c := d + 5
    goto L4
L3: d := d + 4
    b := b - 4
    if b != d goto L1
    goto L3
L4:
```

This program uses six temporaries `a-f`. Assume that our machine has only 4 available registers `$r0`, `$r1`, `$r2`, and `$r3` and that only `e` is live on exit from this program.

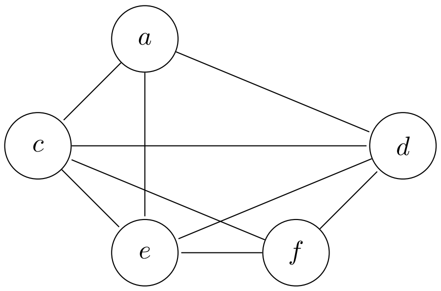
(a) Compute the Calculate live-in and live-out at each statement and drzaw the register interference graph. (Computing the sets of live variables at every program point may be helpful for this step.)



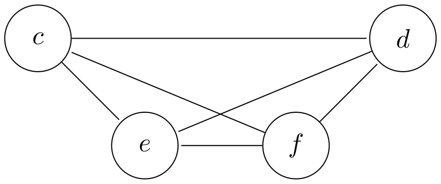


(b) Use the graph coloring heuristics discussed in lecture to assign each temporary to a register on a machine that has 4 registers. Rewrite the program replacing temporaries by registers and including whatever spill code is necessary. Use the pseudo-instructions `load x` and `store x` to load and spill the value of `x` from memory.

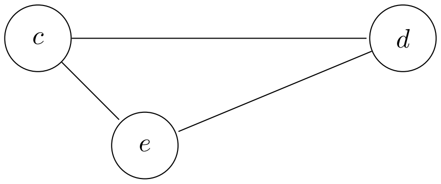
(a) pick b as a candidate for spilling



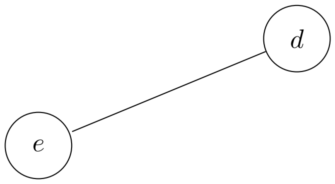
(b) remove a, stack:a



(c) remove f, stack:a,f



(d) remove c, stack:a,f,c



(e) remove e,d, stack:a,f,c,e,d

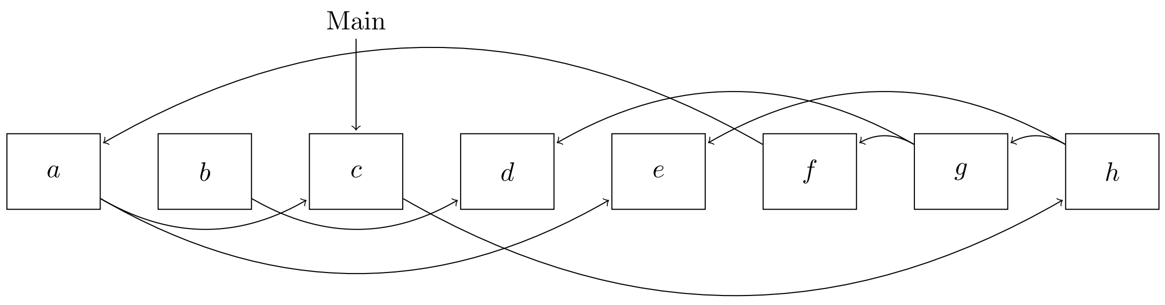
The result is
e: \$r0
d: \$r1
c: \$r2
f: \$r3
a: \$r3
b: in Memory

```
L0:
    $r0 := 0;
    $r3 := 1;
    store $r3

    $r1 = 2;
    $r2 := 4;
L1:
    load $r3
    $r3 := $r3+2
    $r0 := $r0 + $r2
    $r3 := $r3 * $r3
    if $r3 < $r2 goto L3
L2: $r0 := $r0 + $r3
    $r2 := $r1 + 5
    goto L4
L3: $r1 := $r1 + 4
    load $r3
    $r3 := $r3 - 4
    if $r3!= $r1 goto L1
    goto L3
L4:
```

7. (10*3=30 pts) Consider the following Cool program:

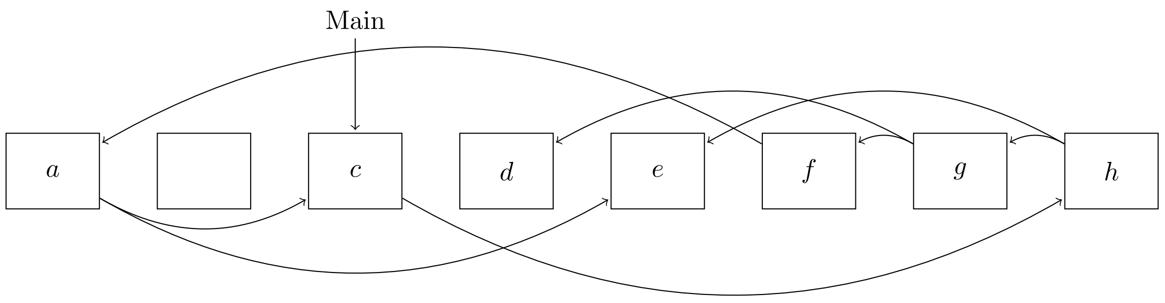
```
class C {
  x : C; y : C;
  setx(newx : C) : C { x <- newx };
  sety(newy : C) : C { y <- newy };
  setxy(newx : C, newy :C) : SELF_TYPE {{ x <- newx; y <- newy; self; }};
};

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(a), a.setxy(e, c); b.setx(d); g.setxy(d,f); c.sety(h); h.setxy(e, g); x <- c;
    }
  };
};
```

(a) (10 pts) Draw the heap at the end of execution of the above program, identifying objects by the variable 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 this object is not in the heap (note that Main is pointing to c).

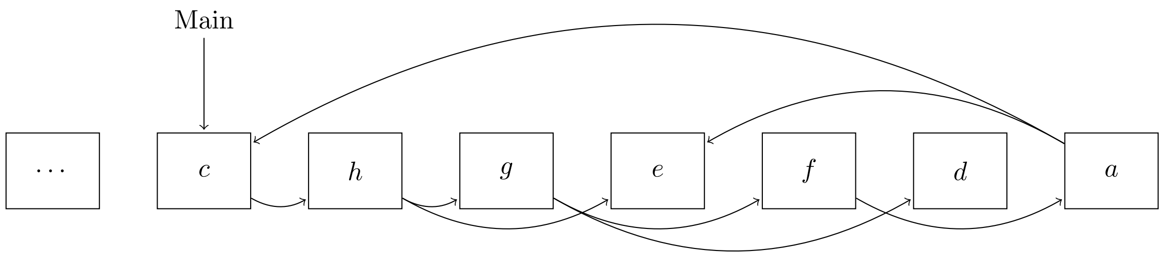


(b) (10 pts) For each of the garbage collection algorithms discussed in class (Mark and Sweep, Copy Collector, Reference Counting), show the heap after garbage collection.

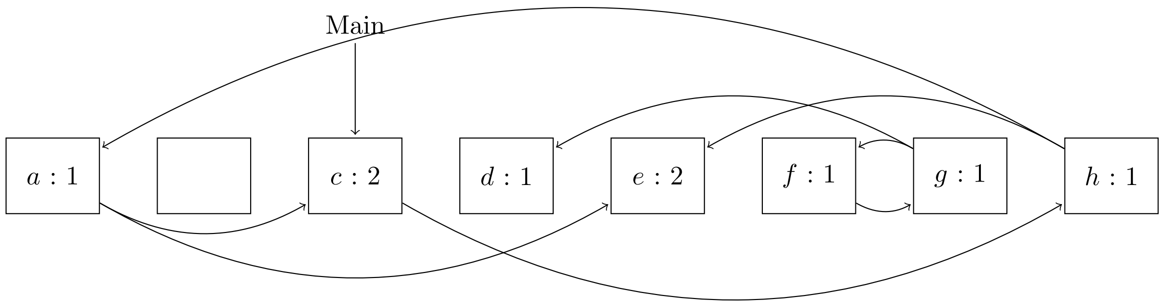
i. Mark and Sweep



ii. Copy Collector



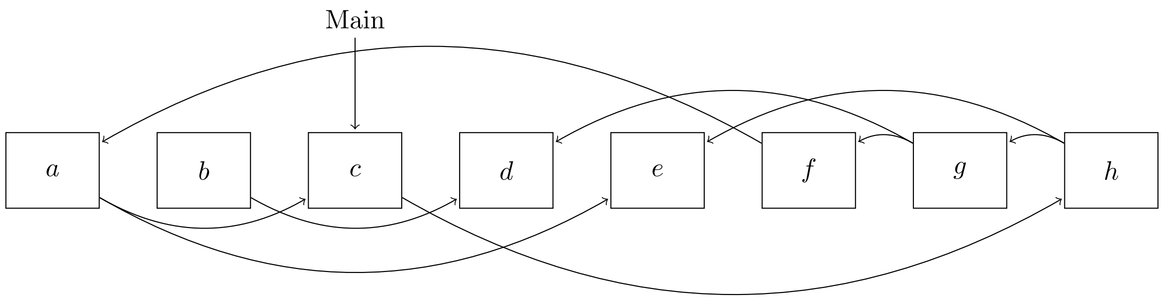
iii. Reference Counting

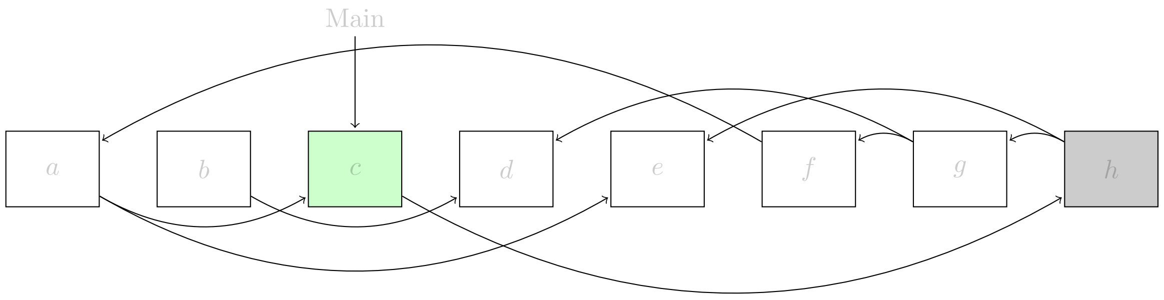


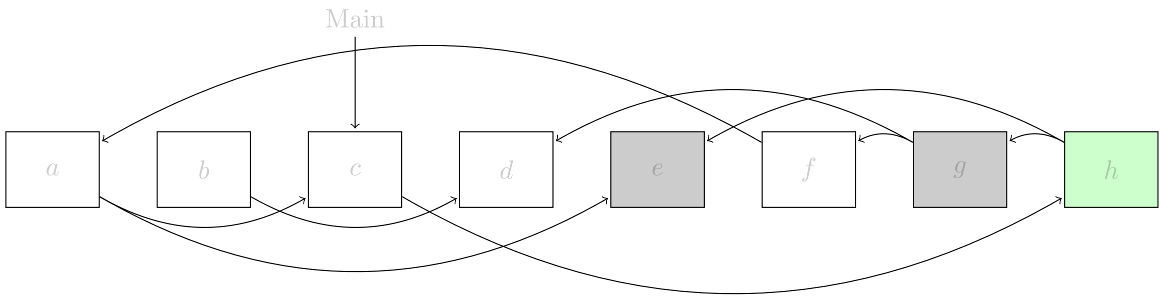
(c) (10 pts) Suppose the cost c_i is every reachable words' instruction number, comapare 3 algorithms, let H be the size of the heap, let R be the reachable word's data size, when will the Mark and

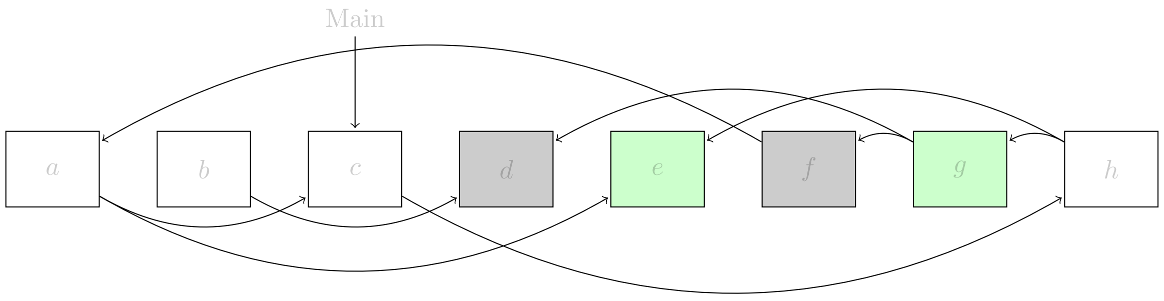
Sweep outperform Copy Collector.
The cost of Mark and Sweep:

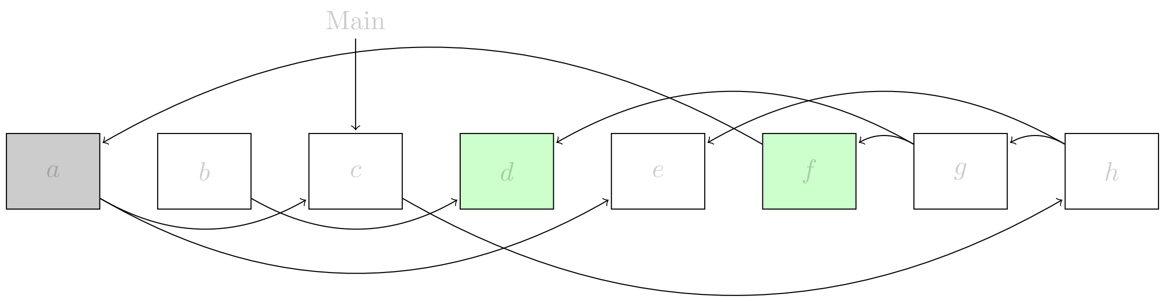
(d) (10 pts) Suppose the marking process is multithreaded, use the above example show that tricolor marking with write barrier is thread safe.










Main
a
b
c
d
e
f
g
h