CS131 Compilers: Writing Assignment 4
Due Tuesday, June 12, 2018 at 23:59

Yuyang Rong - 69850764

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. You should use the Latex template provided at the course web site to write your solution.

I worked with: (Name,ID), (Name,ID)...

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.

Table 1: Object Layout

Offset	A	B	C	Explanation
0	A tag	B tag	C tag	Class tag
4	5	6	7	Object Size
8	*	*	*	Dispatch Ptr
12	a1	a1	a1	Attribute
16	a2	a2	a2	Attribute
20	N/A	a3	a3	Attribute
24	N/A	N/A	a4	Attribute

Table 2: Dispatch Table

Offset	A	B	C
0	A.m1()	B.m1()	B.m1()
4	A.m2()	A.m2()	C.m2()
8	N/A	B.m3()	C.m3()

(b) Let `obj` be a variable whose static type is `A`. Assume that `obj` is stored in register `$a0`. Write MIPS code for the function invocation `obj.m2()`. You may use temporary registers such as `$t0` if you wish.

```
lw      $t0,    8($a0)   # Get access to dispatch table.
lw      $t0,    4($t0)    # Get function pointer
jalr    $t0
```

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

When `obj` has dynamic type B, the processor will look for B's dispatch table and call the function listed inside, even though they may be the same function that is inherited from A.

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.

`let a:T[e1] in e2`

$$\frac{\begin{array}{l} so, S_1, E \vdash T[e_1] : void, S_2 \\ l_1 = newloc(S_2) \\ S_3 = S_2[void/l_1] \\ E' = E[l_1/a] \\ so, S_3, E' \vdash e_2 : v_2, S_4 \end{array}}{so, S_1, E \vdash \texttt{let a:T[e}_1\texttt{] in e}_2 : v_2, S_4} \quad \text{[Let-Array]}$$

`a[e1]` $\leftarrow$ `e2`

$$\frac{\begin{array}{l} so, S_1, E \vdash e_2 : v, S_2 \\ E(a[e_1]) = l_a \\ S = S_1[v/l_a] \end{array}}{so, S_1, E \vdash \texttt{a[e}_1\texttt{]} \leftarrow \texttt{e}_2 : v, S} \quad \text{[Assign-Array]}$$

`a[e]`

$$\frac{\begin{array}{l} E(a[e]) = l_a \\ S(l_a) = v \end{array}}{so, S_1, E \vdash \texttt{a[e]} : v, S} \quad \text{[Reference-Array]}$$

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 $\texttt{x} < 10$ or `void` if $\texttt{x} \geq 10$.

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

$$\frac{so, S_1, E \vdash e_1 : Bool(false), S_2}{so, S_1, E \vdash \texttt{while } \texttt{e}_1 \texttt{ loop } \texttt{e}_2 \texttt{ pool} : void, S_2} \qquad \text{[Loop-False]}$$

$$\frac{\begin{array}{l} so, S, E \vdash e_1 : Bool(true), S_1 \\ so, S_1, E \vdash e_2 : v, S_2 \\ so, S_2, E \vdash e_1 : Bool(false), S_3 \end{array}}{so, S, E \vdash \texttt{while } \texttt{e}_1 \texttt{ loop } \texttt{e}_2 \texttt{ pool} : v, S_3} \qquad \text{[Loop-Last]}$$

$$\frac{\begin{array}{l} so, S, E \vdash e_1 : Bool(true), S_1 \\ so, S_1, E \vdash e_2 : v_1, S_2 \\ so, S_2, E \vdash e_1 : Bool(true), S_4 \\ so, S, E \vdash \texttt{while } \texttt{e}_1 \texttt{ loop } \texttt{e}_2 \texttt{ pool} : v_2, S_4 \end{array}}{so, S, E \vdash \texttt{while } \texttt{e}_1 \texttt{ loop } \texttt{e}_2 \texttt{ pool} : v_2, S_4} \qquad \text{[Loop-True]}$$

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:
        move  $fp $sp
        sw    $ra 0($sp)
        addiu $sp $sp -4
        lw    $a0 4($fp)
        sw    $a0 0($sp)
        addiu $sp $sp -4
        li    $a0 0
        lw    $t1 4($sp)
        addiu $sp $sp 4
        beq   $a0 $t1 true_branch
false_branch:
        lw    $a0 4($fp)
        sw    $a0 0($sp)
        addiu $sp $sp -4
        sw    $fp 0($sp)
        addiu $sp $sp -4
        lw    $a0 4($fp)
        sw    $a0 0($sp)
        addiu $sp $sp -4
        li    $a0 1
        lw    $t1 4($sp)
        sub   $a0 $t1 $a0
        addiu $sp $sp 4
        sw    $a0 0($sp)
        addiu $sp $sp -4
        jal   f_entry
        lw    $t1 4($sp)
        add   $a0 $a0 $t1
        addiu $sp $sp 4
        b     end_if
true_branch:
        li    $a0 0
end_if:
        lw    $ra 4($sp)
        addiu $sp $sp 12
        lw    $fp 0($sp)
        jr    $ra
```

```
void fib(const unsigned int a){
    if (a == 0){
        return 0;
    } else {
        int temp = a - 1;
        return fib(temp) + temp;
    }
}
```

5. (10 pts) Give a recursive definition of the cgen function for the following new construct.

$$\texttt{for i} = \texttt{e}_1 \texttt{ to e}_2 \texttt{ by e}_3 \texttt{ do e}_4$$

Assume that the subexpressions e_1, e_2, e_3 and e_4 are integer-valued. A "for loop" expression is evaluated according to the following rules. The first three subexpressions are evaluated once at the start of the loop in the order e_1, e_2, and then e_3. The subexpression e_4 is evaluated once per iteration of the loop. The index variable i is initialized to the value of e_1. The loop bound is the value of e_2 and i is incremented by the value of e_3 after each iteration. The loop terminates before executing an iteration where the value of i is greater than the loop bound. The value returned by the "for loop" expression is the value of the expression e_4 in the last iteration. If the loop does not execute at all, then the value returned is the integer 0.

Following is a more formal semantics of the for expression in terms of the Cool expressions.

let t: Int $\leftarrow e_1$ in
let bound:Int $\leftarrow e_2$ in
let incr:Int $\leftarrow e_3$ in
let result:Int $\leftarrow$ 0 in
let i:Int $\leftarrow t$ in
    while ($i \leq$ bound) loop {
        result $\leftarrow e_4$;
        $i \leftarrow i$ + incr;
    } pool;
    result

Note that the expressions e_1, e_2 and e_3 are evaluated ONLY once before the start of the loop. Also note that any occurences of variable i in e_1, e_2 and e_3 refer to the value of i just before the for loop. Any occurrence of variable i in expression e_4 refers to the loop index variable i.

$$
\begin{aligned}
& \text{cgen}(\texttt{for i} = \texttt{e}_1 \texttt{ to e}_2 \texttt{ by e}_3 \texttt{ do e}_4) \\
= & \text{cgen}(e_1) \\
& \text{push } (acc) \\
& \text{cgen}(e_2) \\
& \text{push } (acc) \\
& \text{cgen}(e_3) \\
& \$s3 \leftarrow \$acc \\
& \text{pop} \\
& \$s2 \leftarrow \$acc \\
& \text{pop} \\
\text{Loop:} & \\
& \$t0 \leftarrow top \\
& \text{pop} \\
& \text{bgt } \$t0\ \$s2 \text{ EndLoop} \\
& \text{cgen}(e_4) \\
& \text{pop} \\
& \text{addiu } \$t0, \$t0, \$s3 \\
& \text{push } \$t0 \\
& \text{b Loop} \\
\text{EndLoop:} &
\end{aligned} \tag{1}
$$

6. (2*10=20 pts) 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:
```

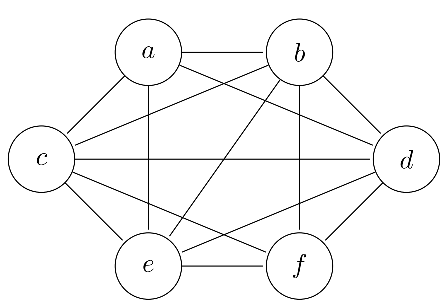
This program uses six temporaries `a-f`. Assume that our machine has only 4 available registers `$r0`, `$r1`, `$r2`, and `$r3` and that only `e` is live on exit from this program.

(a) Draw 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.

We will start by compute the set of live vars first:

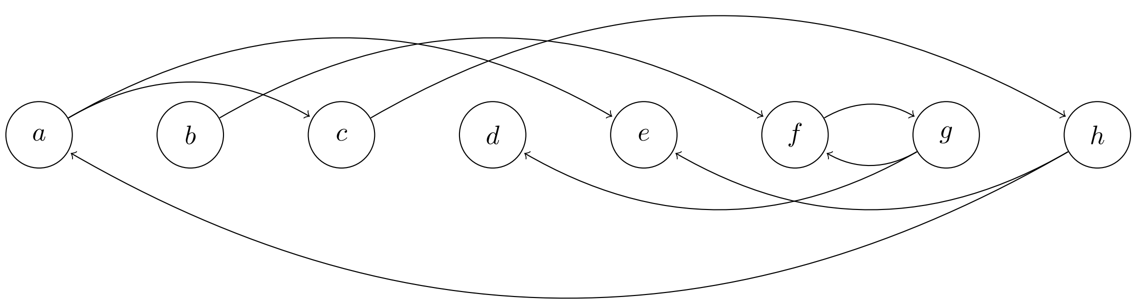
```
L0: e := 0                # {e}
    b := 1;               # {b, d}
    d := 2;               # {b, d, e}
L1: a := b+2              # {a, b, d, e}
    c := d+5              # {a, b, c, d, e}
    e := e + c            # {a, b, c, d, e}
    f := a * a            # {b, c, d, e, f}
    if f < c              # {b, d, e, f}
        goto L3           # {b, d, e}
L2: e := e + f            # {b, d, e}
    goto L4               #
L3: e := e + 2            # {b, d, e}
L4: d := d + 4            # {b, d, e}
    b := b - 4            # {b, d, e}
    if b != d             # {b, d, e}
        goto L1           # {b, d, e}
L5:                       # {e}
```



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

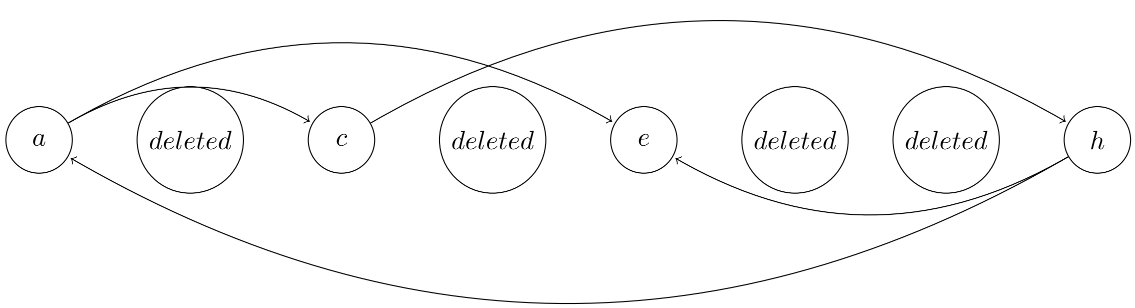
```
class C {
  x : C; y : C;
  setx(newx : C) : C { x <- newx };
  sety(newy : C) : C { y <- newy };
  setxy(newx : C, newy :C) : SELFT_TYPE {{ x <- newx; y <- newy; self; }};
};

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) (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).

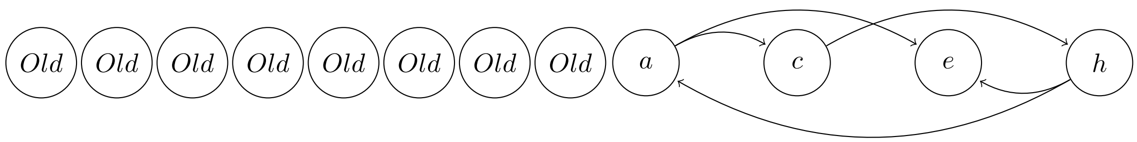


(b) (10 pts) For each of the garbage collection algorithms discussed in class (Mark and Sweep, Stop and Copy, Reference Counting), show the heap after garbage collection.

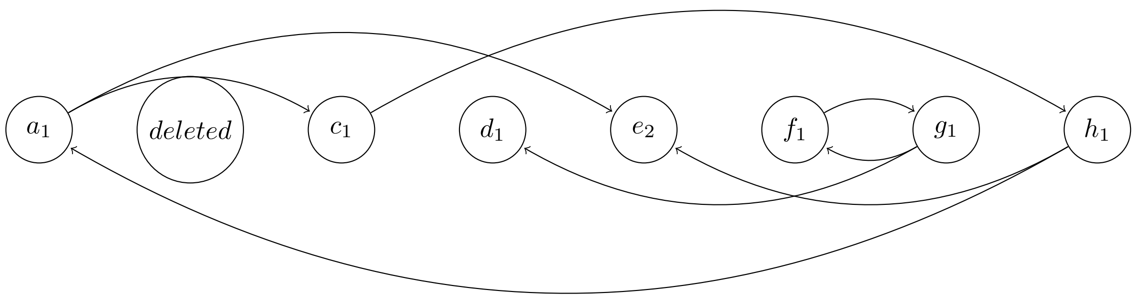
Mark and Sweep



Stop and Copy



Reference Counting



(c) (10 pts) Which technique performed the worst for the above program ? Describe why the technique failed to reclaim the memory occupied by one or more heap variables which are no longer reachable.

Performance is a rather vague word when used alone. We will analysis in terms of space performance and time performance.

Space Performance Reference counting performed the worth as it failed to remove all the garbage. This is due to it's counting mechanism, where one piece of memory, no matter used or not, can not be deleted as long as someone else is referencing it.

Time Performance Stop and Copy consumes the most time. It is deleting by coping the useful data to a new place and leave the useless behind. Yet this behavior is time consuming because delete is cheap, copy all the data verbatim is expensive.