CS143 Final
Spring 2016

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam, all with multiple parts.
- This exam is designed to take 2 hours, but you have the full 3 hours to complete the exam.
- The exam is open note. You may use laptops, phones and e-readers to read electronic notes, but not for computation or access to the internet for any reason.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. Do not write on the back of exam pages or other pages.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. You may get as few as 0 points for a question if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.

NAME: _____

In accordance with both the letter and spirit of the Honor Code, I have neither given nor received assistance on this examination.

SIGNATURE: _____

Problem	Max points	Points
1	30	
2	20	
3	30	
4	20	
5	20	
TOTAL	120	

1. **Language Design** (30 points)

Consider extending COOL with exceptions. There are two new expressions. The first throws an exception:

```
throw e
```

The expression `e` must evaluate to a string, and that string is propagated as an exception value.

Except for the try-catch expression described below, for any other kind of expression `e`, if a subexpression of `e` throws an exception then `e` throws that same exception. This is described further below. Exception values can be caught using a try-catch statement:

```
try e1 catch x in e2
```

In this statement, expression `e1` is evaluated first. If it completes normally with a value v (that is, without throwing an exception), then the whole try-catch expression completes normally with value v. Otherwise, if `e1` throws an uncaught exception, the variable `x` is bound to the exception string, and `e2` is evaluated. As examples, the expression `try 1 catch x in 2` evaluates to $Int(1)$, and `try (throw "a") catch x in x` evaluates to the object $String(1, a)$.

To give operational semantics to exceptions, we need to give the operational semantics for the two new statements. However, we also need to define the exception propagation behavior for existing constructs in the language. We introduce Exc(m) (where m is a string object) as a new kind of value that expressions can evaluate to—i.e., expressions can evaluate to objects (as usual) or exceptions. The behavior of exceptions is different from the behavior of objects. For example, consider an arithmetic operation `e1+e2`: If the first operand `e1` evaluates to Exc(m), then the whole expression evaluates to Exc(m). If `e1` completes normally with a value v and `e2` evaluates to Exc(m), then the whole expression evaluates to Exc(m). Otherwise (i.e., both expressions evaluate normally), the usual rule from the COOL manual applies. Formally, the rules for the propagation of exceptions in addition expressions are:

$$\frac{so, S_1, E \vdash e_1 \mapsto \text{Exc}(m), S_2}{so, S_1, E \vdash e_1 + e_2 \mapsto \text{Exc}(m), S_2}$$

$$\frac{so, S_1, E \vdash e_1 \mapsto Int(i_1), S_2 \quad so, S_2, E \vdash e_2 \mapsto \text{Exc}(m), S_3}{so, S_1, E \vdash e_1 + e_2 \mapsto \text{Exc}(m), S_3}$$

(a) Give the operational semantics for throw and try-catch expressions. Make sure you cover both exception propagation as well as regular executions.

(b) Give sound type-checking rules for throw and try-catch expressions. Hint: Make sure that your type-checking rules can correctly type-check the following program (you don't have to show the derivation):

```
if 1 < 2 then throw "error" else 1 + 2
```

2. **Garbage Collection** (20 points)

Assume we are programming in a safe language like Java or python. Consider the following fragment of code:

```
class Foo { Int i; }

for (Int j = 0; j < 1000000000; j++) {
  // Allocate memory
  Foo data = new Foo;    // Allocates a Foo object on the heap.

  // Generate random number
  data.i = random_number();

  // Print random number
  print("Number is %d\n", data.i);
}
```

(a) Which GC Algorithm (Reference Counting, Mark and Sweep, Stop and Copy) is best suited (gives the fastest start-to-finish execution times) for this code? Justify your choice.

(b) Assume the same program is run on a machine with sufficient physical memory to hold n `Foo` objects. What is the maximum number of objects that can be allocated on the heap at any single time by an implementation using a Mark and Sweep collector? Justify your answer.

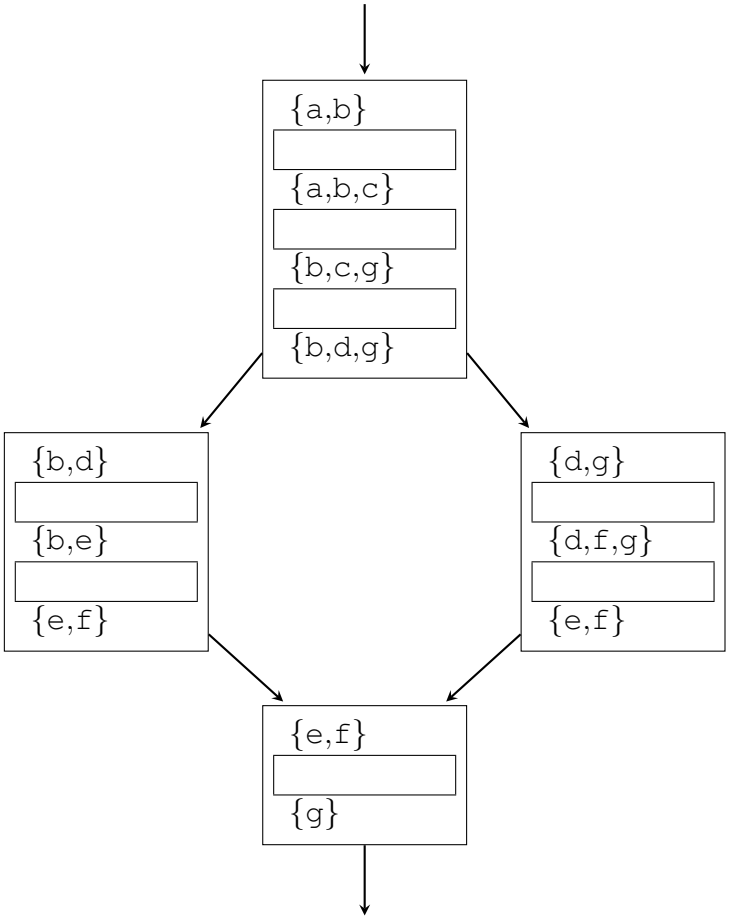
(c) What is the maximum number of objects that can be allocated on the heap at any single time by an implementation using a Reference Counting collector? Justify your answer.

(d) What is the maximum number of objects that can be allocated on the heap at any single time by an implementation using a Stop and Copy collector? Justify your answer.

3. **Global Optimization & Register Allocation** (30 points)

(a) You are given the control-flow graph for a function which uses variables `a` to `g`, with `a` and `b` being the arguments and `g` the return value. The program statements have been removed from this graph, and replaced by empty boxes. Below/above every such box we give the set of variables that are live right after/before the corresponding statement.

Fill in program statements in the boxes, so that the given live sets are consistent with the code. Program statements can take one of the following forms: `x = 1`, `x = y` or `x = y + z` (where `x`, `y` and `z` can be any one of the variables used by the function). If more than one statement would work in some position, pick the simplest one (prefer `x = 1` over `x = y`, and `x = y` over `x = y + z`).



(b) Give the register interference graph for the above code, by filling in the adjacency matrix below (mark X on the cell at row v_1 and column v_2 iff there's an edge between nodes v_1 and v_2). You may also want to draw the graph to help with the next questions, but we won't grade your drawing.

	a	b	c	d	e	f	g
a							
b							
c							
d							
e							
f							
g							

(c) Perform register allocation for the above code, using the graph coloring algorithm presented in class. Fill in (and use!) the minimum number of registers k for which you can color the graph without spilling. If at any point there's a choice between multiple variables to push on the stack, pick the one which comes first alphabetically. Show the state of the stack right after you've removed the last node from the graph:

k =

top (pushed last) | | | | | | | | bottom (pushed first)

(d) Show your final coloring on the following table, by listing the variables assigned to each register. Use only the first k registers, where k is the number of colors you used.

Register	Variables
r_1	
r_2	
r_3	
r_4	
r_5	
r_6	
r_7	

4. **Local Optimization** (20 points)

Consider three address programs that consist only of the following types of operations:

- `x = y` for variable `x` and variable or constant `y`
- `x = y + z` for variable `x` and variables or constants `y` and `z`.

Answer each of the following questions. For each problem, state which variable(s) are live on exit. For full credit, your solution should be as short as possible.

(a) Give a program where performing local optimizations in the order copy propagation, constant folding and then dead code elimination results in a final program with fewer instructions than performing first constant folding, copy propagation and then dead code elimination.

(b) Give a program for which the opposite is true: performing constant folding, copy propagation and then dead code elimination is better than performing copy propagation, constant folding and then dead code elimination.

(c) Give a program for which both of copy propagation and constant folding must be done at least twice each before performing dead code elimination to achieve the best results.

5. **Activation Records** (20 points)

Consider the following COOL class and method definitions:

```
1     class Main {
2         apple(a: Int, b: Int, c: Bool): Int {
3             if c then
4               if a - b = 0 then
5                 0
6               else
7                 a + b
8               fi
9             else
10            {
11              a <- a * b;
12              a + b;
13            }
14            fi
15        };
16
17        banana(x: Int, y: Int): String {
18            if x < cherry(y) then
19              "Peel"
20            else
21              "Split"
22            fi
23        };
24
25        cherry(z: Int): Int {
26            let p: Int <- 2 in {
27              apple(z, p, false);
28            }
29        };
30
31        main(): Object {
32            let q: Int <- 3, r: Int <- 4 in {
33              banana(q, r);
34            }
35        };
36    };
```

A compiler with an unknown runtime management strategy is used to compile this code—in particular, the strategy used may not be exactly one that you have seen before. A partial listing of the stack at a certain point in the execution of the compiled program is given below. There are three kinds of entries on the stack: a return address, a stack address, and a reference to an object in the heap (e.g., `Int(3)`).

(a) Fill in the missing entries in the table.

Address	Contents
1000	
1004	*OS return address*
1008	Int(3)
1012	
1016	1000
1020	
1024	Int(4)
1028	*return address of banana(q,r)*
1032	
1036	
1040	
1044	
1048	Int(2)
1052	
1056	
1060	
1064	
1068	*return address of apple(z,p,false)*

(b) On the previous page, clearly indicate the latest possible point in the execution of the program that could have this stack contents.