CS143 Final
Spring 2021

- Please read all instructions (including these) carefully.
- There are 5 questions on the exam + the honor code, some with multiple parts. You have 95 minutes to both finish the exam and upload it. After 95 minutes, gradescope will close your exam and automatically submit it, so make sure to submit what you have by then.
- The exam is open note. You may use laptops, phones, e-readers, and the internet, but you may not consult anyone.
- You must upload your answers to gradescope and tag each question, just like the written assignments. You may submit images of hand-written answers taken with your phone (but allow yourself time to send the image to your computer, so you can upload it to gradescope). It is your responsibility, however, to ensure they are legible. Computer typed answers as a PDF are also permitted.
- 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.

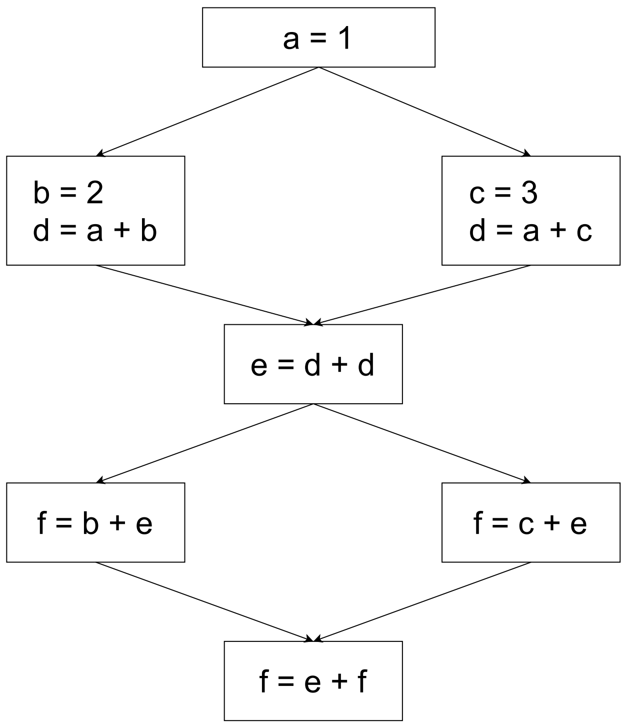
Problem	Max points	Points
1	Honor Code	
2	20	
3	10	
4	25	
5	25	
6	20	
TOTAL	100	

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

   Type your name as a signature below:

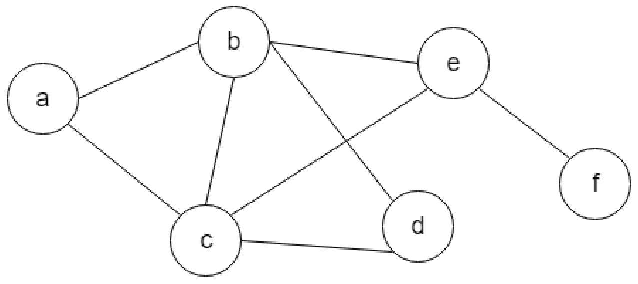
2. **Register Allocation**

Consider the following control-flow graph (CFG) where no variables are live on exit:



(a) Run the algorithm for building register inference graphs from the lectures on the CFG and draw the resulting register inference graph.

Answer:



(b) Run the register allocation algorithm from the lectures on the CFG, removing nodes with the least degree first and breaking ties alphabetically. Show the stack after it is filled with all nodes and also show the assignment of variables to registers.

Answer:

Stack (bottom to top): f, a, d, b, c, e

Assignment:
e: `r0`
c: `r1`
b: `r2`
d: `r0`
a: `r0`
f: `r1`

(c) Assume that using an uninitialized variable is undefined behavior in the programming language of the CFG. Thus, the compiler need not preserve correctness for program paths where variables are used before their initialization. Taking advantage of this observation, provide an assignment of variables to registers that uses the fewest possible number of registers.

Answer:

By taking advantage of the additional assumption, we need only 2 registers.

Assignment:
a: `r0`
b: `r1`
c: `r1`
d: `r0`
e: `r0`
f: `r1`

3. **Dataflow Analysis**

Consider the following basic blocks where live variables are listed in curly brackets before and after statements. The basic blocks form a CFG where the edges are missing. No variables are live on exit from the CFG. (We have given the blocks labels in their upper left corners for your convenience.)

```
1
{a,b}
c = a
{b}
d = b
{d}
```

```
2
{c}
a = c
{a,c}
b = 1
{a,b,c}
```

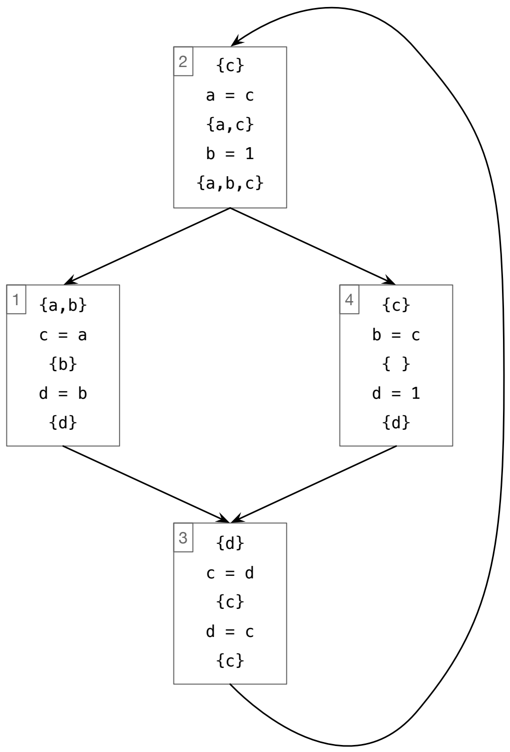
```
3
{d}
c = d
{c}
d = c
{c}
```

```
4
{c}
b = c
{ }
d = 1
{d}
```

Provide the missing control flow graph edges.

Answer:

There are multiple solutions, one of which is given below. No variables are live on exit, so every solution known to us needs a back-edge that makes c live on exit from 3.



```
2
{c}
a = c
{a,c}
b = 1
{a,b,c}
```

```
1
{a,b}
c = a
{b}
d = b
{d}
```

```
4
{c}
b = c
{ }
d = 1
{d}
```

```
3
{d}
c = d
{c}
d = c
{c}
```

4. **Cool Refactoring**

In this question, we will change the semantics of two Cool constructs and ask for new type inference rules, operational semantic rules, or both.

(a) We will change the type of a Cool if-then-else expression

$$\texttt{if}\ e_0\ \texttt{then}\ e_1 : T_1\ \texttt{else}\ e_2 : T_2\ \texttt{fi}$$

to be either the type of the then expression (T_1) or the type of the else expression (T_2), whichever is a superclass of the other. If neither $T_1 \leq T_2$ nor $T_2 \leq T_1$, then the expression does not type check. Provide the type inference rule or rules that describes the new if-then-else semantics. You may not redefine `lub` or $\leq$, nor define a new function.

Answer:

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1' = \begin{cases} \texttt{SELF\_TYPE}_c & \text{if } T_1 = \texttt{SELF\_TYPE} \\ T_1 & \text{Otherwise} \end{cases} \\ T_2' = \begin{cases} \texttt{SELF\_TYPE}_c & \text{if } T_2 = \texttt{SELF\_TYPE} \\ T_2 & \text{Otherwise} \end{cases} \\ T_1' \leq' T_2' \vee T_2' \leq T_1' \end{array}}{O, M, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi } : T_1' \sqcup T_2'}$$

Note $\texttt{SELF\_TYPE}_c \sqcup T = C \sqcup T$ and $S \leq T \implies T \sqcup S = T$.
So $T_1' \sqcup T_2' = T_1'$ or $T_1' \sqcup T_2' = T_2'$.

(b) We will change the type of the Cool while loop to have the same type as the loop body. The loop either returns the value of the loop body or, if the loop executed no iterations, the default value of its type. Give the type inference rule and the operational semantic rules for the new while loop.

Answer:

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : T_2}$$

Unfortunately there is no truly satisfactory answer within the formalism we have. The following is close. We throw away the side effect on the store caused by evaluating e_1 twice, however, there might be side effects through IO.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 \mapsto Bool(True), S_2 \\ so, S_2, E \vdash e_2 \mapsto v_2, S_3 \\ so, S_3, E \vdash e_1 \mapsto Bool(True), S_4 \\ so, S_3, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} \mapsto v_3, S_5 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} \mapsto v_3, S_5} \text{ Loop-Continue}$$

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 \mapsto Bool(True), S_2 \\ so, S_2, E \vdash e_2 \mapsto v_2, S_3 \\ so, S_3, E \vdash e_1 \mapsto Bool(False), S_4 \end{array}}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} \mapsto v_2, S_4} \text{ Loop-Last}$$

$$\frac{so, S_1, E \vdash e_1 \mapsto Bool(False), S_2}{so, S_1, E \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} \mapsto D_{T_2}, S_3} \text{ Loop-Never}$$

Where D_{T_2} is the default value of T_2.

There are betters solutions but they require either a the ability to generate fresh names (think the moral equivalent of newloc except for E) or adding a new type of expression which is generated by the compiler in the evaluation of while loops If anyone is interested in more complete explanation of these email me (Caleb) and I will type them up.

5. **Stack Data Structures**

We will extend Cool with a stack data structure that stores objects and that supports

- `push` (add an object to the top of the stack),
- `pop` (remove an object from the top of the stack), and
- `top` (return the object at the top of the stack).

For example, at the end of the following program, the stack contains the `Int` 1 at the bottom and the `Int` 2 at the top, while `v` references an object whose dynamic type is `Int` and whose value is 2.

```
a := Stack;
a.push(1);
a.push(2);
a.push(3);
a.pop();
v : Object <- a.top();
```

(a) The stack uses a linked list to store its elements, with a header object for each stack element. Provide the object memory layout for both the stack and the stack element header. (That is, list the different data items that must be stored in these two types of objects.)

Answer:
The gc tags are stored in the word before the object.

Stack
gc tag
class tag
object size
dispatch pointer
ptr to head

Element
gc tag
class tag
object size
dispatch pointer
next ptr
object ptr

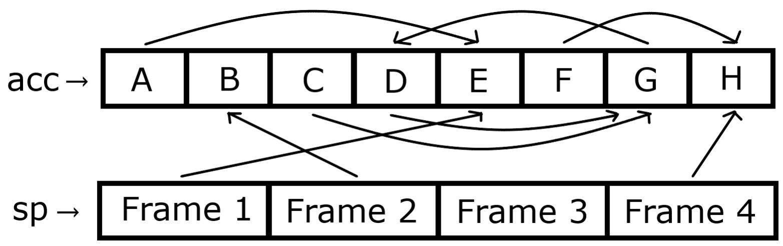
(b) Given your stack runtime design, write MIPS assembly code for the `pop` method. You may assume the address of the stack is already stored in `$r1` and you may use any other registers you need. On errors, you must jump to a pre-defined `error` label.

Answer:

```
lw $r2 12($r1) // $r2 = list.head
beq $r2 $zero error // check if null
lw $r2 12($r2) // $r2 = $r2.next
sw $r1 12($r1) // list.head = $r2
```

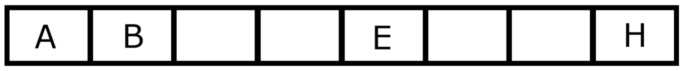
6. **Garbage Collection**

Assume the accumulator register (acc) and the stack pointer register (sp) is pointing to the following objects on the heap (top) and stack (bottom). In your answers, leave heap slots empty to indicate free memory.



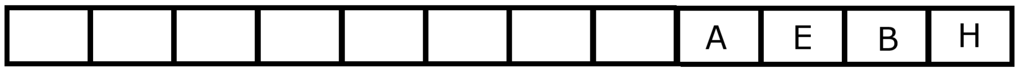
(a) Draw the heap after a pass of mark and sweep garbage collection.

Answer:



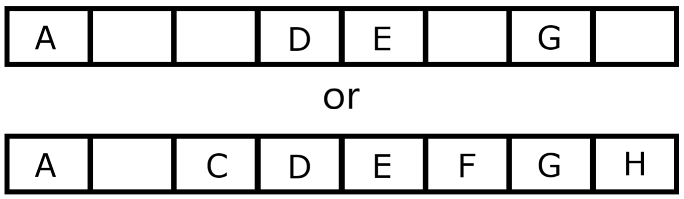
(b) Draw the full heap (both old and new space) after stop and copy garbage collection. Assume the above heap depicts old space, that new space is to the right, and that the algorithm checks acc first and then the stack frames from left to right.

Answer:



(c) Assume the objects are reference counted and draw the state of the heap after stack frames 2, 3, and 4 are popped from the stack. The acc register remains unchanged.

Answer:


A
D
E
G
or
A
C
D
E
F
G
H